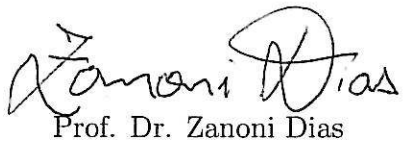
Enumeração de *traces*
e
Identificação de *breakpoints*
Estudo de aspectos da evolução

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Christian Baudet e aprovada pela Banca Examinadora.

Campinas, 13 de Dezembro de 2010.



Prof. Dr. Zanoni Dias
Instituto de Computação (IC) – Unicamp
(Orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Silvania Renata de Jesus Ribeiro Cirilo – CRB8 / 6592

Baudet, Christian
B324e Enumeração de traces e identificação de breakpoints: estudo de aspectos da evolução/Christian Baudet-- Campinas, [S.P. : s.n.], 2010.

Orientador : Zanoni Dias
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Computação Científica.

1.Evolução molecular. 2.Genomas. 3.Bioinformática. 4.Biologia computacional.. I. Dias, Zanoni. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Enumeration of traces and breakpoint identification: study of evolutionary aspects.

Palavras-chave em inglês (Keywords): 1.Molecular evolution. 2.Genomes. 3.Bioinformatics. 4.Computational biology.

Área de concentração: Ciência da Computação

Titulação: Doutor em Ciência da Computação

Banca examinadora: Prof. Dr. Zanoni Dias (IC – UNICAMP)
Prof. Dr. João Carlos Setubal (VBI– Virginia Tech)
Prof. Dr. Nalvo Franco de Almeida Jr. (FACOM-UFMS)
Prof. Dr. João Meidanis – IC – UNICAMP
Prof. Dr. Guilherme Pimentel Telles – IC - UNICAMP

Data da defesa: 13/12/2010
Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

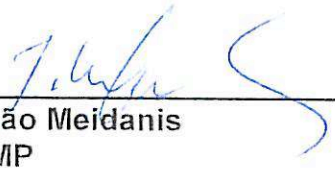
Tese Defendida e Aprovada em 13 de dezembro de 2010, pela Banca examinadora composta pelos Professores Doutores:



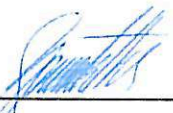
Prof. Dr. Nalvo Franco de Almeida Junior
FACOM / UFMS



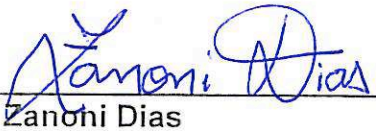
Prof. Dr. João Carlos Setubal
VBI / Virginia Tech



Prof. Dr. João Meidanis
IC / UNICAMP



Prof. Dr. Guilherme Pimentel Telles
IC / UNICAMP



Prof. Dr. Zanoni Dias
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Enumeração de *traces*
e
Identificação de *breakpoints*
Estudo de aspectos da evolução

Christian Baudet[1]

13 de Dezembro de 2010

Banca Examinadora:

- Prof. Dr. Zanoni Dias
  Instituto de Computação (IC) – Unicamp (Orientador)
- Prof. Dr. João Carlos Setubal
  Virginia Bioinformatics Institute (VBI) – Virginia Tech
- Prof. Dr. Nalvo Franco de Almeida Jr.
  Faculdade de Computação (FACOM) – UFMS
- Prof. Dr. João Meidanis
  Instituto de Computação (IC) – Unicamp
- Prof. Dr. Guilherme Pimentel Telles
  Instituto de Computação (IC) – Unicamp

[1]Suporte financeiro de: CAPES – Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Estágio de Doutorando PDEE (Processo BEX 4676/08-4) 03/2009–01/2010

Resumo

O estudo de rearranjo de genomas tem o objetivo de auxiliar o entendimento da evolução. Através da análise dos eventos de mutação como inversões, transposições, fissões, fusões, entre outros, buscamos compreender as suas influências sobre o fenômeno da diferenciação das espécies. Dentro deste contexto, esta tese ataca dois temas distintos: a Enumeração de *Traces* e a Identificação de *Breakpoints*.

Os algoritmos de ordenação de permutações por reversões orientadas produzem uma única solução ótima enquanto o conjunto de soluções é imenso. A enumeração de *traces* de soluções para este problema oferece um modo mais compacto de representar o conjunto completo de soluções ótimas. Dessa maneira, esta técnica fornece aos biólogos a possibilidade de análise de diversos cenários evolutivos.

Neste trabalho, realizamos um estudo para melhora da eficiência do algoritmo de enumeração através da adoção de uma estrutura de dados mais simples. Devido ao caráter exponencial do problema, grandes permutações não podem ser processadas em um tempo satisfatório. Assim, com o objetivo de produzir cenários evolucionários alternativos para grandes permutações, propomos e avaliamos estratégias para a enumeração parcial de traces.

Os pontos de quebra (ou *breakpoints*) são regiões que delimitam os segmentos conservados existentes nos cromossomos e denotam a ocorrência de rearranjos evolutivos. As técnicas de identificação de *breakpoints* têm a função de identificar tais pontos nas sequências dos cromossomos.

Nesta tese, implementamos um método de detecção e refinamento de pontos de quebra proposto na literatura e o disponibilizamos como um pacote que pode ser utilizado por outros pesquisadores. Além disso, introduzimos uma nova metodologia de identificação de *breakpoints* baseada na análise da cobertura de *hits* observada nos alinhamentos de sequências intergênicas, provenientes dos genomas das espécies comparadas.

Abstract

The study of genome rearrangements helps biologists understand the evolution of species. The species differentiation phenomenon are derived by analyzing mutational events (inversions, transpositions, fissions, fusions, etc) and their effects. In this context, this work aims the study of two different subjects: *Traces* Enumeration and *Breakpoint* Identification.

Algorithms that solve the problem of sorting oriented permutations through reversals output only one optimal solution, although the set of solutions can be huge. The enumeration of *traces* of solutions for this problem allows a compact representation of the set of all optimal solutions which sort a permutation. By using this technique, biologists can study many evolutionary scenarios.

We carried out a study to improve the efficiency of the enumeration algorithm by adopting a simple data structure. Due to the exponential nature of the problem, large permutations cannot be processed at a satisfactory time. Thus, in order to produce alternative evolutionary scenarios for large permutations, we proposed and evaluated strategies for partial enumeration of traces.

Breakpoints are regions that border conserved segments in the chromosomes and reflect the occurrence of evolutionary rearrangements. The techniques for *breakpoint* identification are meant to identify such points in the chromosome sequences.

In this work, we implemented a method proposed in the literature, that performs detection and refinement of *breakpoints*. The implementation is available as a package to other researchers. Additionally, we introduced a new methodology for *breakpoint* identification based on the analysis of the hit coverage observed in the alignments of intergenic sequences.

Agradecimentos

À minha mãe Ituko Yabase, ao meu pai Gérard Jacques André Baudet, aos meus irmãos Nathalie Baudet, Patrick Baudet e Jacqueline Louise Baudet dedico o trabalho realizado neste estudo. Minha família demonstrou apoio incondicional e me encheu de amor e carinho durante todos estes anos.

Ao meu orientador, Professor Zanoni Dias, deixo meus mais sinceros agradecimentos pela disposição, pela paciência e pelo incentivo dispensados durante todo o doutorado. No decorrer deste período de trabalho conjunto, uma grande amizade foi construída.

No período de Março de 2009 até Junho de 2010, realizei um estágio doutoral (Doutorado Sanduíche) nas dependências do *Laboratoire de Biométrie et Biologie Évolutive* localizado na *Université Claude Bernard* em Lyon, na França onde parte do estudo deste trabalho foi conduzido. Fui maravilhosamente acolhido por toda a equipe do laboratório liderada pela Professora Marie-France Sagot a quem deixo um agradecimento mais do que especial.

Parte do trabalho desenvolvido utilizou métodos e resultados produzidos em outros estudos. Gostaria de agradecer Marília Vieira Braga e Claire Lemaitre por terem fornecido material necessário para a realização do estudo aqui desenvolvido.

Agradeço a todos os meus amigos que me acompanharam aqui no Brasil e na França. Gostaria de lembrar os nomes daqueles que, de alguma forma, tiveram participação especial nesta jornada: André Atanásio Maranhão Almeida, Angela Marcela Perez Castañeda, Augusto Fernandes Vellozo, Cecilia Coimbra Klein, Deisy Christine Mazzini, Geny Tavares, Janice Kielbassa, Lauranne Duquenne, Lin Tzy Li, Paulo Vieira Milreu, Patrícia Simões, Vicente Acuña e Vincent Lacroix.

Finalmente, gostaria de deixar um agradecimento especial para a minha namorada Dorota Ewa Długosz que me acompanhou nos dois últimos anos de estudo e que, com seu amor e carinho, me motivou ainda mais.

Sumário

Resumo **vii**

Abstract **ix**

Agradecimentos **xi**

Apresentação **1**

I *Traces* **3**

1 Eventos de rearranjo de genoma **5**
1.1 Rearranjo de Genomas 5
1.2 Eventos de Rearranjo 7
1.2.1 Reversões 7
1.2.2 Transposições 10
1.2.3 Translocações 11
1.2.4 Combinações de Eventos 12
1.3 Ordenação de permutações por reversões orientadas 14
1.3.1 Grafo de *breakpoints* 14
1.3.2 Tipos de reversões 18
1.3.3 Cálculo da distância de reversão 19
1.3.4 Reversões seguras e inseguras 23

2 Enumeração de *traces* **25**
2.1 *Traces* 26
2.1.1 Intervalos 26

2.1.2 Classe de equivalência 27
2.1.3 Forma Normal de um *Trace* 28
2.2 Enumeração de soluções 29
2.3 Enumeração de *traces* de soluções 29
2.4 Aplicação biológica da enumeração de *traces* 31
2.4.1 Utilização de restrições biológicas 32
2.4.2 Comparação de genomas de grupos de espécies 37
2.5 Otimização do algoritmo de enumeração de *traces* 43
2.5.1 A estrutura original 44
2.5.2 A estrutura de pilha 46
2.5.3 Comparação de performance 51
2.5.4 Conclusão 55
2.5.5 Trabalhos futuros e trabalhos relacionados 56

3 Enumeração parcial de *traces* 59
3.1 Algoritmos de enumeração parcial de *traces* 60
3.1.1 Algoritmo aleatório 60
3.1.2 Algoritmo tradicional limitado por tempo 60
3.1.3 Algoritmo de janela deslizante 62
3.2 Avaliação dos algoritmos 66
3.3 Conclusão 92
3.3.1 Trabalhos futuros 93

II *Breakpoints* 95

4 Pontos de quebra no genoma 97
4.1 Rearranjos evolutivos 97
4.1.1 Modelo de quebras aleatórias 98
4.1.2 Modelo de regiões frágeis 99
4.2 Detecção de rearranjos através da comparação de genomas 101
4.2.1 Métodos experimentais 102
4.2.2 Comparação da ordem dos genes 104
4.2.3 Alinhamento de genomas completos 117

5 Identificação e refinamento de *Breakpoints* 127
5.1 Obtenção de pontos de quebra em duas etapas 128
5.1.1 Identificação de blocos de sintenia 129
5.1.2 Refinamento de *breakpoints* 139
5.2 Pacote `Cassis` 149
5.2.1 Implementação do pacote 150
5.2.2 Comparação dos genomas do homem e do camundongo 157
5.2.3 Utilizando `Cassis` com blocos de sintenia 159
5.2.4 Conclusão 171

6 Análise de regiões intergênicas 175
6.1 Definição do conjunto de dados 176
6.2 Alinhamento de sequências intergênicas 181
6.3 Metodologia para detecção de *breakpoints* 183
6.3.1 Fragmentação das regiões intergênicas 186
6.3.2 Janela deslizante 190
6.3.3 Encadeamento de *hits* 199
6.4 Conclusão 210

Considerações finais 213

Bibliografia 218

Lista de Tabelas

2.1 Permutações da família $(+2, -3, \ldots, -n, +1)$: número de soluções e de *traces* 26

2.2 Permutações entre `R1` e seus seis descendentes 39

2.3 Exemplo de cenário evolucionário 40

2.4 *Traces* de soluções obtidos para as bactérias do gênero *Rickettsia* . . 41

5.1 Resultados obtidos por Lemaitre na identificação de *breakpoints* sobre três conjuntos de genes ortólogos. 130

5.2 Resultados obtidos pelo método de construção de k-blocos sobre a lista de pares de genes ortólogos entre homem e camundongo obtidos do `Ensembl` 139

5.3 Mínimo, máximo, média e mediana dos comprimentos dos *breakpoints* identificados entre o homem e o camundongo pelo programa `Cassis` ao processar a lista de pares de genes ortólogos obtida do `Ensembl` . . 158

5.4 Mínimo, máximo, média e mediana dos comprimentos dos *breakpoints*, identificados entre o homem e o camundongo (`Ensembl`), que passaram na verificação estatística 159

5.5 Mínimo, máximo, média e mediana dos comprimentos dos 345 blocos de sintenia entre o homem e o camundongo obtidos da base de dados `Compara` do `Ensembl` 161

5.6 Número de *breakpoints* identificados pelo `Cassis` ao processar a lista de blocos de sintenia entre homem e camundongo da base de dados `Compara` do `Ensembl` 161

5.7 Mínimo, máximo, média e mediana dos comprimentos dos *breakpoints* identificados entre o homem e o camundongo pelo programa `Cassis` ao processar a lista de blocos de sintenia da base de dados `Compara` do `Ensembl` 162
5.8 Mínimo, máximo, média e mediana dos comprimentos dos *breakpoints*, identificados entre o homem e o camundongo (`Compara`), que passaram na verificação estatística 163
5.9 Mínimo, máximo, média e mediana dos comprimentos dos 672 LCBs identificados pelo programa `Mauve` nos genomas do homem e do camundongo 165
5.10 Mínimo, máximo, média e mediana dos comprimentos dos *breakpoints* identificados entre o homem e o camundongo pelo programa `Cassis` ao processar a lista de LCBs do programa `MAUVE` 166
5.11 Mínimo, máximo, média e mediana dos comprimentos dos *breakpoints*, identificados entre o homem e o camundongo (`MAUVE`), que passaram na verificação estatística 166
5.12 Mínimo, máximo, média e mediana dos comprimentos dos *breakpoints* identificados entre o homem e o camundongo pelo programa `Cassis` com os conjuntos de blocos de sintenia `GENES`, `BLOCKS` e `LCBS` 168
5.13 Mínimo, máximo, média e mediana dos comprimentos dos *breakpoints*, identificados entre o homem e o camundongo pelo programa `Cassis` com os conjuntos de blocos de sintenia `GENES`, `BLOCKS` e `LCBS`, que não foram refinados. 169
5.14 Mínimo, máximo, média e mediana dos comprimentos dos *breakpoints*, identificados entre o homem e o camundongo pelo programa `Cassis` com os conjuntos de blocos de sintenia `GENES`, `BLOCKS` e `LCBS`, que foram refinados. 170

6.1 Tamanhos e quantidade de bases mascaradas apresentados pelos cromossomos dos genomas do homem e do camundongo 177
6.2 Número e tamanho médio dos genes e regiões intergênicas do homem 179
6.3 Número e tamanho médio dos genes e regiões intergênicas do camundongo 180
6.4 Resultados dos alinhamentos de todas as sequências intergênicas humanos contra todos as sequências gênicas do camundongo 184

6.5 *Breakpoints* identificados pelo `Cassis` com base na lista de pares genes ortólogos entre homem e camundongo 188
6.6 Cobertura de *hits* média apresentada pelos fragmentos de 100 bp dentro e fora das regiões de *breakpoints* identificadas pelo `Cassis` 189
6.7 Número de regiões com cobertura abaixo da média identificadas com o algoritmo de janela deslizante, conforme o valor do parâmetro w utilizado 193
6.8 Contagem de regiões que apresentaram algum tipo de sobreposição com os *breakpoints* definidos pelo `Cassis` 194
6.9 Número de regiões identificadas com o algoritmo de janela deslizante e tamanho médio apresentado por elas 195
6.10 Número de regiões com cobertura abaixo da média identificadas, em cada cromossomo, pelo algoritmo que possui duas etapas de janela deslizante 197
6.11 Contagem de regiões, produzidas pelo algoritmo que possui duas etapas de janela deslizante, que apresentaram algum tipo de sobreposição com os *breakpoints* definidos pelo `Cassis` 198
6.12 Número de regiões identificadas com o algoritmo que possui duas etapas de janela deslizante e tamanho médio apresentado por elas 199
6.13 Tamanho médio dos *breakpoints* identificados com o método de encadeamento de *hits* 203
6.14 Contagem de *breakpoints* definidos com os algoritmos de janela deslizante que sobrepõem os pontos de quebra do `Cassis` 205
6.15 Relações entre *breakpoints* não refinados identificados pelo `Cassis` e os *breakpoints* identificados com o processo de encadeamento de *hits* de regiões identificadas com o algoritmo de janela deslizante 207
6.16 Relações entre *breakpoints* refinados identificados pelo `Cassis` e os *breakpoints* identificados com o processo de encadeamento de *hits* de regiões identificadas com o algoritmo de janela deslizante 208

Lista de Figuras

1.1 Reversão e Transposição 9
1.2 Translocação 11
1.3 Exemplo de construção passo a passo de um grafo de *breakpoints* . . . 16
1.4 Exemplo de grafos de *breakpoints* de uma permutação circular e de uma permutação identidade linear 16
1.5 Exemplo de atribuição de orientações das arestas pretas de um grafo de *breakpoints* 17
1.6 Exemplos de tipos de componentes 18
1.7 Exemplos de reversões de quebra, de junção e neutra 19
1.8 Exemplos de reversões de corte e de união 21
1.9 Exemplo de um super-obstáculo 22
1.10 Exemplo de uma fortaleza 23
1.11 Exemplos de uma reversão insegura e de uma reversão segura 24

2.1 Árvore filogenética do gênero *Rickettsia* 38
2.2 Ordem cronológica das reversões 43
2.3 Esquema da estrutura *Compressible sorted set* 44
2.4 Representação de um conjunto de *traces* em uma árvore 47
2.5 Esquema da estrutura de pilha 48
2.6 Número de *traces*, tempo de execução e memória utilizada pelas estruturas `CS` e `ST` 52
2.7 Memória utilizada e tempo de execução médios das estruturas `CS` e `ST` durante o processamento de permutações com distância de reversão $d = \lceil (n+1)/2 \rceil$ 53
2.8 Evolução do uso de memória pelas estruturas `CS` e `ST` durante o processamento de uma permutação complexa 55

3.1 Ramos mortos em uma árvore de *traces* 62
3.2 Número de *traces*, tempo de execução e memória máxima médios observados no processamento de permutações com 10 e 15 elementos (lineares e circulares) 67
3.3 Obstáculos com tamanho mínimo 70
3.4 Exemplos de permutações que possuem distância de reversão 1 71
3.5 Exemplos de permutações que possuem distância de reversão 2 72
3.6 Permutação com $d(\pi) = 3$ e dois *traces* de soluções 74
3.7 Permutações com $d(\pi) = 3$ contendo 2 e 3 componentes orientadas . . 75
3.8 Permutação com $d(\pi) = 3$ e um 4-*ciclo* orientado com três cruzamentos entre os arcos 76
3.9 Permutação com $d(\pi) = 3$ e um 4-*ciclo* orientado com 4 cruzamentos entre os arcos 77
3.10 Permutações com $d(\pi) = 3$ e uma componente orientada composta por um 2-*ciclo* orientado e um 3-*ciclo* não orientado 78
3.11 Permutação com $d(\pi) = 3$ e uma componente orientada composta por um 3-*ciclo* orientado e um 2-*ciclo* orientado que cobre uma aresta preta do ciclo maior 78
3.12 Permutação com $d(\pi) = 3$ e uma componente orientada composta por um 3-*ciclo* orientado e um 2-*ciclo* orientado que cobre duas arestas pretas do ciclo maior 79
3.13 Permutação com $d(\pi) = 3$ e uma componente orientada composta por um 3-*ciclo* orientado e um 2-*ciclo* não orientado 80
3.14 Permutação com $d(\pi) = 3$ e uma componente orientada composta por dois 2-*ciclos* não orientados que se sobrepõem e um 2-*ciclo* orientado que sobrepõe um dos ciclos não orientados 81
3.15 Porcentagem de *traces* com uma dada altura x encontrada pelos algoritmos `Random`, `Window 4`, `Window 5`, `Window 6` e `Stack` conforme o tempo limite de execução. 83
3.16 Porcentagem de *traces* com um dado tamanho médio de reversão x encontrada pelos algoritmos `Random`, `Window 4`, `Window 5`, `Window 6` e `Stack` conforme o tempo limite de execução. 84

3.17 Número de *traces* e memória máxima utilizada médios observados no processamento de permutações aleatórias com os algoritmos `Stack`, `Random`, `Window 4`, `Window 5` e `Window 6` durante um tempo limite de um minuto 86
3.18 Distribuição média dos *traces* de acordo com a altura 90
3.19 Distribuição média dos *traces* de acordo com o tamanho médio das reversões 91

4.1 Árvore filogenética do gene da insulina encontrados no homem, no rato e no camundongo 107
4.2 Esquema da duplicação do gene da insulina 107
4.3 Exemplo de um gráfico *dotplot* 113
4.4 Exemplo de conflito na construção de blocos de sintenia 116

5.1 Esquema de um *breakpoint* inter-cromossômico 129
5.2 Esquema de um *breakpoint* intra-cromossômico 130
5.3 *Breakpoints* tipos `I` e `II` 132
5.4 Exemplo de arcos que formam um conflito do tipo `I` 134
5.5 Exemplo de arcos que formam um conflito do tipo `II` 135
5.6 Exemplos de *breakpoints* sobre o genoma G_r 140
5.7 Sequências de interesse de um *breakpoint* 141
5.8 Representação esquemática das regiões ortólogas entre as sequências S_r, S_{oA} e S_{oB} 141
5.9 Representação gráfica dos *hits* dos alinhamentos de S_r com S_{oA} e S_r com S_{oB} sobre a sequência S_r. 144
5.10 Exemplo gráfico de curva da função de pontuação $s(k)$ 146
5.11 Versões estendidas das sequências S_r, S_{oA} e S_{oB} 152
5.12 Exemplo de gráfico de *dotplot* produzido pelo pacote `Cassis` 154
5.13 Exemplo de gráfico de segmentação produzido pelo pacote `Cassis` . . 155
5.14 Histograma de distribuição das diferenças de tamanhos observadas nos *breakpoints*, produzidos por `Cassis` com os pares de genes ortólogos do homem e do camundongo obtidos do `Ensembl`, antes e depois da segmentação 160

5.15 Histograma de distribuição das diferenças de tamanhos observadas nos *breakpoints*, produzidos por `Cassis` com os blocos de sintenia do homem e do camundongo obtidos da base de dados `Compara` 164
5.16 Histograma de distribuição das diferenças de tamanhos observadas nos *breakpoints*, produzidos por `Cassis` ao processar a lista de LCBs do programa `MAUVE` 167

6.1 Esquema gráfico exemplificando processo de tratamento de sobreposições em *hits* 185

Apresentação

A existência de fósseis e a grande diversidade de espécies há muito tempo sugeriam aos cientistas que animais e plantas estavam sujeitos a algum conjunto de forças evolutivas.

Em 1809, o naturalista francês Jean-Baptiste Pierre Antoine de Monet, Chevalier de Lamarck, apresentou a sua teoria dos caracteres adquiridos que se baseava em dois pontos principais. O primeiro ponto referia-se à tendência que os seres possuem de apresentar um melhoramento. Esta tendência geraria um aumento de complexidade de seres menos desenvolvidos para seres mais desenvolvidos. O segundo ponto era a lei do uso e desuso. Segundo esta lei, um órgão ou parte do corpo usado continuamente tende a se desenvolver e, no sentido oposto, um órgão ou parte do corpo menos utilizado tende a se atrofiar. Segundo Lamarck, estes dois aspectos agiriam em conjunto e as características adquiridas pelos organismos seriam transferidas às gerações seguintes.

Cinquenta anos mais tarde, a publicação do livro "A Origem das Espécies" (*The Origin of Species*), pelo naturalista britânico Charles Robert Darwin em 1859, introduziu a Teoria da Evolução que, hoje, é considerada como base da biologia moderna. Segundo esta teoria, organismos vivos se adaptam gradualmente em um processo de seleção natural. Nesse processo, organismos mais adaptados teriam mais chances de sobreviver às pressões do meio e dessa forma seriam capazes de transferir suas características a seus descendentes. Assim, as espécies se ramificam sucessivamente a partir de formas ancestrais.

O botânico austríaco Gregor Johann Mendel, considerado atualmente como o pai da genética, descreveu em 1865 os princípios da herança genética. Em seu trabalho, Mendel propõe duas leis que ditam as bases da genética tradicional. A primeira lei diz que os alelos segregam de maneira aleatória. Isso significa que se os alelos dos pais são A e a, então um membro da geração F_1 tem a mesma chance de herdar A ou

a. A segunda lei determina que pares de alelos segregam independentemente, o que significa que a herança dos alelos do gene X é independente da herança dos alelos do gene Y.

Apesar das idéias de Mendel e Darwin não terem sido relacionadas na época, hoje sabemos que os mecanismos genéticos descritos inicialmente por Mendel são fundamentais para o comportamento evolutivo observado por Darwin.

Contudo, apesar dos muitos avanços obtidos nas últimas décadas, desvendar o mecanismo da evolução das espécies ainda é um dos grandes objetivos da ciência. O estudo deste assunto envolve inúmeros ramos de pesquisa. A análise pode considerar desde aspectos sociais e comportamentais, para entender como a interação entre organismos influencia em suas evoluções, até aspectos moleculares, para compreensão de como as mudanças no DNA, nas proteínas e nas vias metabólicas trazem vantagens para que um indivíduo prevaleça em relação a outro.

Dentro deste contexto, o trabalho desenvolvido nesta tese busca estudar dois tópicos ligados ao estudo de rearranjos de genomas e à análise comparativa de genomas.

O primeiro tópico refere-se ao estudo do problema de enumeração de *traces* de soluções do problema de ordenação de permutações orientadas com o uso de reversões. A enumeração de *traces* visa fornecer aos biólogos cenários evolucionários alternativos para que estes possam determinar qual cenário realmente ocorreu durante a evolução das espécies, dentro de um modelo de parcimônia máxima.

O segundo tópico aborda a identificação de *breakpoints* através da análise comparativa dos genomas de duas espécies. O objetivo deste estudo se encontra não somente na identificação dos pontos de quebra, que definem as fronteiras dos rearranjos, como também na análise das características destas regiões. Estas informações poderão ajudar os biólogos a determinar se é possível ou não identificar regiões do genoma que são suscetíveis a quebras e que, portanto, são fundamentais para os eventos de rearranjos.

Visando uma melhor organização do texto, esta tese será dividida em duas partes, cada uma destinada a apresentar os conceitos e definições básicos e a desenvolver o assunto relacionado ao tópico associado. A Parte I abordará o estudo do problema de enumeração de *traces* e a Parte II apresentará o estudo do problema de identificação de *breakpoints*.

Parte I

Traces

Capítulo 1

Eventos de rearranjo de genoma

Mutações são eventos que provocam alterações nos materiais genéticos das espécies. Estes eventos podem ser pontuais, como a alteração de um único nucleotídeo na sequência, ou acontecer em escalas maiores, alterando a estrutura dos cromossomos. Tais eventos podem promover a ativação/desativação de genes ou, até mesmo, alterações nas propriedades dos produtos produzidos pelos genes.

Podemos ver facilmente que todo este mecanismo é fundamental para a evolução das espécies. Alterações no material genético podem trazer vantagens aos indivíduos que, dessa maneira, se tornam mais aptos a sobreviver às pressões do meio ambiente.

O estudo desenvolvido nesta tese abordará eventos de rearranjo que afetam a estrutura dos cromossomos. Neste capítulo, apresentaremos alguns conceitos e aspectos históricos sobre os eventos de rearranjos de genoma e listaremos resultados que podemos encontrar na literatura.

1.1 Rearranjo de Genomas

Os pioneiros da área de rearranjo de genomas foram Dobzhansky e Sturtevant que, em 1938, publicaram um estudo com uma árvore evolucionária que mostrava um cenário de 17 reversões para as espécies *Drosophila pseudoobscura* e *Drosophila miranda* [57]. Com o passar dos anos, outros estudos mostraram que eventos de rearranjos são comuns na evolução molecular de plantas, mamíferos, vírus e bactérias [8, 37, 63, 74–78, 90, 139, 141].

No final dos anos 80, Palmer *et al.* realizaram a comparação entre os genomas

mitocondriais das espécies *Brassica oleracea* e *Brassica campestris* e descobriram que elas são muito similares (muitos genes são entre 99 e 99,9% idênticos) [122]. Em outro exemplo, O'Brien traz em seu livro "Genetics Maps: Locus Maps of Complex Genomes" um estudo que mostra que as maiores diferenças entre os genomas das bactérias *Escherichia coli* e *Salmonella typhimurium* são as ordens de seus genes em seus cromossomos [43]. Com a evolução das técnicas de sequenciamento, que permitiu que genomas completos fossem obtidos, a área de rearranjo de genomas passou a ter uma quantidade imensa de material para estudo.

Os eventos de rearranjo mais comumente estudados são *reversões*, *transposições*, *translocações*, *fusões* e *fissões*.

Intuitivamente, uma *reversão* ocorre quando um segmento do genoma é destacado, invertido e ligado no mesmo local da sequência cromossômica. Este tipo de evento tem papel de extrema importância na diversidade das plantas e bactérias [106].

A *transposição* é um evento que envolve a troca de posição de dois blocos adjacentes no genoma. Existe uma generalização do evento de transposição chamada *block-interchange* [40]. Neste evento, não existe a restrição de que os segmentos trocados sejam adjacentes.

Ao contrário da reversão e da transposição, a *translocação* é um evento que envolve dois cromossomos. Suponha que dois cromossomos X e Y sejam divididos em duas partes cada, formando segmentos (X_1, X_2) e (Y_1, Y_2) (nenhum deles vazio). A translocação produz dois novos cromossomos combinando o prefixo ou sufixo de um cromossomo com o prefixo ou sufixo do outro. Assim, uma translocação pode produzir os cromossomos (X_1, Y_2) e (Y_1, X_2) ou (Y_1, X_1^R) e (Y_2^R, X_2), onde X^R denota o complemento reverso de X. As translocações, junto com as reversões, são os eventos mais comuns na evolução dos mamíferos [106].

Fusão e *fissão* são outros eventos observados em mamíferos. A fusão age de forma a unir dois cromossomos para formar um único novo cromossomo. A fissão, por outro lado, realiza o papel inverso ao quebrar um cromossomo em dois novos cromossomos.

No estudo de rearranjo de genomas, cada gene é representado por um identificador. Seja $\mathcal{E}$ um conjunto de identificadores. Um *cromossomo* é definido como uma permutação de identificadores do conjunto $\mathcal{E}$ e um *genoma* é uma coleção de cromossomos. Em geral, supomos que cada gene aparece uma única vez em um genoma.

Normalmente, utilizamos como identificadores números inteiros positivos. Se, adicionalmente, existir informação sobre a orientação dos genes, sinais positivos e

negativos são associados aos identificadores para formar uma permutação que é denominada *permutação orientada*.

Quando comparamos dois cromossomos, consideramos geralmente que um deles é representado pela *permutação identidade* $\iota_n = (1, 2, \ldots, n)$ e o outro é representado pela permutação $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$. O objetivo aqui é obter o número mínimo de eventos necessários (ou a própria sequência de eventos necessários) para que a permutação π seja transformada na permutação identidade.

1.2 Eventos de Rearranjo

Nesta seção detalharemos alguns eventos de rearranjo e listaremos, para cada um deles, uma série de resultados existentes na literatura. A Seção 1.2.1 tratará do evento de reversão e a Seção 1.2.2 abordará o evento de transposição. Translocações serão discutidas na Seção 1.2.3 e os estudos de eventos de rearranjo combinados serão apresentados na Seção 1.2.4.

1.2.1 Reversões

Eventos de reversão foram observados pela primeira vez por Dobzhansky e Sturtevant [56] em 1936. Neste estudo, os autores apontaram a ocorrência de reversões no terceiro cromossomo de variedades selvagens da espécie *Drosophila pseudoobscura*.

Algebricamente, uma reversão é definida como uma operação $\rho(i, j)$ sobre um intervalo $[i, j]$ de uma permutação $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ que realiza a seguinte transformação:

$$\begin{aligned}\rho(i,j)(\pi_1, \ldots, \pi_{i-1}, \underline{\pi_i, \pi_{i+1}, \ldots, \pi_{j-1}, \pi_j}, \pi_{j+1}, \ldots, \pi_n)\\ \rightarrow (\pi_1, \ldots, \pi_{i-1}, \underline{\pi_j, \pi_{j-1}, \ldots, \pi_{i+1}, \pi_i}, \pi_{j+1}, \ldots, \pi_n)\end{aligned}$$

Caso π seja uma permutação orientada, a reversão $\rho(i, j)$ também muda os sinais dos elementos $\pi_i, \ldots, \pi_j$. A Figura 1.1(a) mostra um exemplo do evento de reversão.

Uma das classes de problemas na área de rearranjo de genomas utiliza a reversão como evento base das transformações. Assim, a *distância de reversão*, denotada por $d(\pi, \sigma)$, é definida como sendo o número mínimo de reversões necessárias para transformar a permutação π na permutação σ. O problema de descobrir o número mínimo de reversões necessárias para transformar π na permutação identidade, denotado por $d(\pi)$, é conhecido como *Ordenação por Reversões*.

Kececioglu e Sankoff apresentaram o primeiro algoritmo de aproximação para o problema de ordenação de permutações não orientadas [91]. Este algoritmo utiliza uma estratégia gulosa que atinge uma aproximação de fator 2 e executa em tempo $O(n^2)$ e espaço $O(n)$ para permutações de n elementos.

Posteriormente, Bafna e Pevzner construíram um algoritmo de aproximação com fator 1,75, e tempo de execução $O(n^2)$, usando uma estrutura denominada *Grafo de Breakpoints* [10]. Para o caso de permutações orientadas, o fator de aproximação cai para 1,5 e o tempo de execução para $O(n^{1,5})$.

Em 1998, Christie aprimorou o fator de aproximação para 1,5 para o caso de permutações não orientadas, permanecendo por muito tempo como o melhor resultado conhecido [41].

Berman, Hannenhalli e Karpinski apresentaram em 2002 um algoritmo de aproximação para permutações não orientadas [23]. Este algoritmo possui fator de aproximação 1,375.

Caprara provou que o problema de ordenação por reversões para permutações não orientadas é NP-Completo [36]. Contudo, Hannenhalli e Pevzner provaram que a versão do problema que considera permutações orientadas pode ser resolvida em tempo polinomial, apresentando um algoritmo de complexidade $O(n^4)$ [78]. Em 1996, Berman e Hannenhalli apresentaram um algoritmo com complexidade $O(n^2\alpha(n))$, onde $\alpha(n)$ é o inverso da função de Ackerman, uma função praticamente constante [22]. Em 2000, Kaplan, Shamir e Tarjan construíram um algoritmo mais rápido e mais simples que resolve o problema em tempo $O(n^2)$ [88]. Neste mesmo ano, Meidanis, Walter e Dias provaram que o problema de ordenação por reversões em permutações orientadas circulares pode ser resolvido em tempo $O(n^2)$ [111].

Bergeron apresentou uma visão elementar da teoria de Hannenhalli e Pevzner [78] e forneceu um algoritmo simples com complexidade $O(n^2)$ que não necessita da construção do grafo de *breakpoints* [18].

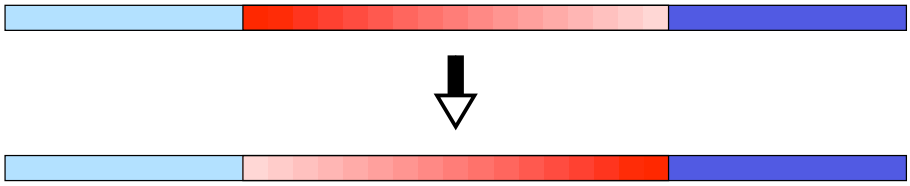
Para o problema de apenas calcular a distância de reversão de duas permutações orientadas, Bader, Moret e Yan apresentaram em 2001 um algoritmo que resolve o problema em tempo linear [6].

Em 2003, Kaplan e Verbin propuseram um algoritmo aleatório com complexidade $O(n\sqrt{n \log n})$ que produz uma sequência ótima de reversões com alta probabilidade para ordenar permutações orientadas [89].

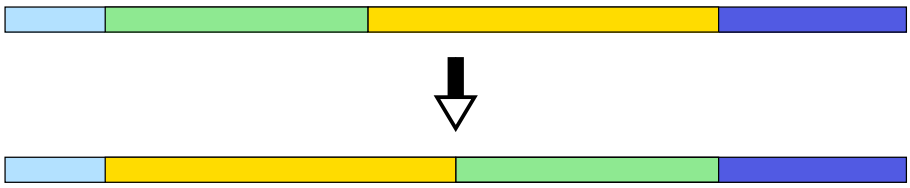
Tannier, Bergeron e Sagot propuseram em 2004 um algoritmo que contorna a necessidade de se executar o passo de detecção de reversões seguras (veja Seção 1.3.4),

que era a operação mais custosa dos algoritmos. Utilizando a estrutura de dados proposta por Kaplan e Verbin, os autores produziram o primeiro algoritmo sub-quadrático, para resolução do problema, com complexidade $O(n^{3/2}\sqrt{\log n})$ [155].

Em 2010, Swenson *et al*. apresentaram um novo algoritmo que tem complexidade de tempo $O(n \log n + kn)$, onde k é o número de "correções" sucessivas que devem ser aplicadas quando o algoritmo aplica uma reversão não segura. O valor de k não é relacionado à distância de reversão mas é limitado por ela. Swenson *et al*. exibiram uma família de permutações onde k é $\Theta(n)$ (pior caso para o valor k) e, nestas condições, o algoritmo roda em tempo quadrático. Contudo, através de testes extensivos, os autores mostraram que, em geral, k é uma constante pequena (menor do que 1) e independente do tamanho da permutação o que faz com que o tempo de execução do algoritmo seja, com alta probabilidade, $O(n \log n)$ [153].



(a) Esquema de reversão



(b) Esquema de transposição

Figura 1.1: (a) No evento de reversão, um segmento de DNA é destacado do genoma e reinserido com a sua ordem invertida. (b) No evento de transposição, dois segmentos consecutivos de DNA são destacados do genoma e reinseridos com as suas posições trocadas.

1.2.2 Transposições

Como mencionado anteriormente, transposições são eventos que trocam a posição de dois blocos adjacentes no mesmo cromossomo. É importante notar que durante o evento, nenhum dos blocos tem sua orientação alterada. A Figura 1.1(b) mostra um exemplo do evento de transposição.

Algebricamente uma transposição é definida como uma operação $\rho(i, j, k)$ sobre uma permutação $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ que realiza a seguinte transformação:

$$\begin{aligned}\rho(i,j,k)(\pi_1,\ldots,\pi_{i-1},\underline{\pi_i,\ldots,\pi_{j-1}},\underline{\pi_j,\ldots,\pi_{k-1}},\pi_k,\ldots,\pi_n)\\ \rightarrow(\pi_1,\ldots,\pi_{i-1},\underline{\pi_j,\ldots,\pi_{k-1}},\underline{\pi_i,\ldots,\pi_{j-1}},\pi_k,\ldots,\pi_n),\end{aligned}$$

onde $1 \leq i < j \leq n+1$, $j < k \leq n+1$ e $k \notin [i, j]$.

Transposições foram primeiramente estudadas por Bafna e Pevzner. Eles determinaram os limites inferiores para a distância de transposição entre cromossomos e propuseram o primeiro algoritmo de aproximação com fator 1,5 e complexidade $O(n^2)$ [9, 11].

Walter, Dias e Meidanis desenvolveram um algoritmo de implementação simples e que executa em $O(n^2)$, porém, seu fator de aproximação é de 2,25 [164].

Christie também produziu um algoritmo com fator de aproximação 1,5 e, apesar do algoritmo possuir complexidade de tempo $O(n^4)$, ele é mais simples de entender [42].

A complexidade do problema de ordenação por transposições permanece em aberto. Nenhum algoritmo polinomial exato é conhecido e nenhuma prova de que ele seja NP-completo foi produzida até a presente data.

Se restrições são impostas ao problema, algoritmos exatos eficientes ou algoritmos de aproximação com melhores fatores podem ser produzidos. Por exemplo, Jerrum apresentou um algoritmo polinomial para ordenação por transposições quando apenas pares de genes adjacentes podem ser trocados [87].

Em outro exemplo, Heath e Vergara consideraram a versão que limita o tamanho dos blocos que estão sendo trocados [80]. O problema pode ser resolvido em tempo $O(n^2)$ se um dos blocos possui um único elemento e o outro bloco não tem limite de tamanho. Para o caso em que o tamanho total dos dois blocos adjacentes é limitado por uma função proporcional a n, os autores reduziram o problema geral para o problema restrito e, assim, mostraram que este é, ao menos, tão difícil quanto o caso

geral. Limitando a 3 o tamanho total dos blocos que estão sendo trocados, Heath e Vergara produziram um algoritmo de aproximação com fator 1,33 [81].

Em 1996, Christie introduziu uma generalização do problema de ordenação por transposição [40]. O evento de *block-interchange* generaliza a transposição na medida em que ele não exige que os blocos, que estão sendo trocados, sejam adjacentes. Christie mostrou neste trabalho que a ordenação por *block-interchange* pode ser resolvida em tempo polinomial com um algoritmo de complexidade $O(n^2)$.

Resultados mais recentes conseguiram aprimorar o fator de aproximação do problema de ordenação por transposições. Em 2005, Elias e Hartman conduziram uma análise em relação ao limite superior do diâmetro de transposição (valor máximo que a distância de transposição pode alcançar) e como resultado produziram uma algoritmo de aproximação com fator 1,375 [64].

1.2.3 Translocações

O evento de translocação envolve o rearranjo de blocos em dois cromossomos distintos. A Figura 1.2 demonstra um exemplo do evento de translocação.

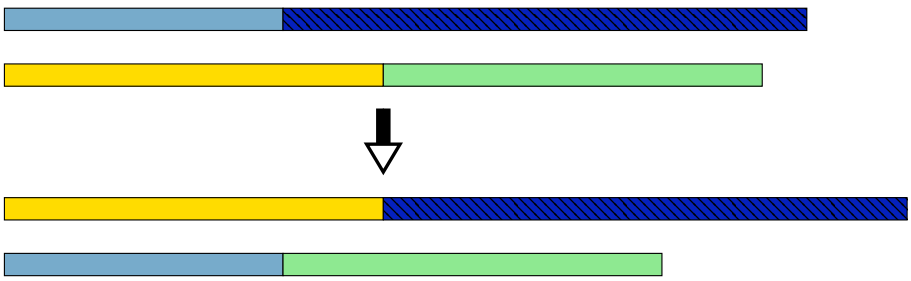


Figura 1.2: O evento de translocação envolve dois cromossomos diferentes. Cada um dos cromossomos é dividido em dois segmentos não vazios e os prefixos/sufixos de cada cromossomo combinam-se entre si, formando dois novos cromossomos.

Dadas duas permutações $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$ e $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, uma translocação prefixo-prefixo $\rho_{pp}(\pi, \sigma, i, j)$ produz duas permutações

$$(\pi_1, \pi_2, \ldots, \pi_{i-1}, \sigma_j, \sigma_{j+1}, \ldots, \sigma_n) \text{ e } (\sigma_1, \sigma_2, \ldots, \sigma_{j-1}, \pi_i, \sigma_{i+1}, \ldots, \sigma_m),$$

onde $1 < i \leq m$ e $1 < j \leq n$.

Uma translocação prefixo-sufixo $\rho_{ps}(\pi, \sigma, i, j)$ aplicada sobre π e σ produz duas permutações

$$(\sigma_n, \sigma_{n-1}, \ldots, \sigma_j, \pi_i, \ldots, \pi_m) \text{ e } (\sigma_1, \ldots, \sigma_{j-1}, \pi_{i-1}, \pi_{i-2}, \ldots, \pi_1),$$

onde $1 < i \leq m$ e $1 < j \leq n$. Se π e σ são permutações orientadas, então uma translocação prefixo-sufixo $\rho_{ps}(\pi, \sigma, i, j)$ produz duas permutações orientadas

$$(-\sigma_n, -\sigma_{n-1}, \ldots, -\sigma_j, \pi_i, \ldots, \pi_m) \text{ e } (\sigma_1, \ldots, \sigma_{j-1}, -\pi_{i-1}, -\pi_{i-2}, \ldots, -\pi_1).$$

Kececioglu e Ravi foram os primeiros a estudar o problema do cálculo da distância de translocação do ponto de vista computacional [90].

O estudo de Kececioglu e Ravi abordou a versão não orientada do problema. Eles obtiveram um algoritmo com fator de aproximação 2 e complexidade de tempo $O(n^2)$, onde n é o número total de genes distintos no genoma. Além disso, eles mostraram que quando é imposta a restrição de que todos os segmentos trocados por uma translocação têm tamanhos idênticos, existe um algoritmo que resolve o problema em tempo linear.

Hannenhalli resolveu em 1996 a versão orientada do problema de ordenação por translocação, produzindo um algoritmo que executa em tempo $O(n^3)$ [73]. Em 2006, Bergeron, Mixtacki e Stoye revisitaram este algoritmo e corrigiram um erro que ele possuía [21]. Em 2002, Zhu e Ma propuseram um algoritmo com complexidade $O(n^2 \log n)$ [175]. Em 2004, Li *et al*. publicaram um algoritmo que realiza o cálculo da distância de translocação de permutações orientadas em tempo linear [103]. Wang *et al*. apresentaram em 2005 um algoritmo que resolve o problema de ordenação por translocação em tempo $O(n^2)$ [165].

A complexidade do problema de ordenação de permutações não orientadas utilizando translocações continua em aberto.

1.2.4 Combinações de Eventos

Combinações de diferentes eventos de rearranjo também são consideradas por muitos pesquisadores. Por exemplo, Bafna e Pevzner sugeriram o problema de ordenação de permutações considerando reversões e transposições. Assim, calcular a distância de reversão e transposição de um par de permutações é obter o número mínimo de eventos de reversão e transposição que transforma uma permutação em outra. De maneira similar, a ordenação por reversões e transposições é o problema de se

encontrar a menor série de reversões e transposições que transforma uma permutação na identidade.

Walter, Meidanis e Dias apresentaram um algoritmo de aproximação com fator 3 para calcular a distância de reversão e transposição de permutações não orientadas e um algoritmo de aproximação com fator 2 para as permutações orientadas. Ambos os algoritmos possuem complexidade $O(n^2)$ [163].

Gu *et al.* apresentaram uma heurística gulosa para o problema de ordenar permutações orientadas por reversões, transposições e transposições invertidas [69]. Uma transposição invertida é um evento de rearranjo no qual, além da transposição, um dos fragmentos transpostos também sofre uma reversão. Posteriormente, eles apresentaram um algoritmo de aproximação com fator 2 e complexidade $O(n^2)$ [70].

Lin e Xue apresentaram um algoritmo unificado de aproximação com complexidade $O(n^2)$ e fator de aproximação 2 para a ordenação de permutações orientadas por reversões e transposições e para ordenação de permutações orientadas por reversões, transposições e transposições invertidas [107].

Wang e Warnow desenvolveram uma técnica chamada `IEBP` (Inverso do número esperado de *breakpoints*) para estimar o que eles chamam de *verdadeira distância evolucionária* entre dois genomas (orientados, não orientados, circulares ou lineares) [167]. A *verdadeira distância evolucionária* é, segundo os autores, o número mínimo de reversões, transposições e transposições invertidas necessárias para transformar um genoma em outro. Wang refinou a técnica e apresentou um método mais preciso, para genomas orientados, denominado `Exact-IEBP` [166].

Alguns cientistas observaram na prática que transposições, em geral, ocorrem com a metade da frequência de ocorrência de reversões [25]. Isso motivou a análise de problemas onde os eventos recebem pesos diferentes.

Com base no algoritmo exato de Hannenhalli e Pevzner para ordenação de permutações orientadas por reversões [78], Eriksen apresentou um esquema de aproximação de tempo polinomial (PTAS) para o problema, com a restrição de que as permutações sejam orientadas e circulares [65].

Dias e Meidanis estudaram o problema de ordenação por fissão, fusão e transposições onde as transposições possuem peso duas vezes maior do que os pesos dos eventos de fusão e fissão [54]. O estudo resultou em um algoritmo polinomial com complexidade de tempo $O(n^2)$ para encontrar a série de fusões, fissões e transposições de peso mínimo quando os genomas são representados como sequências de genes circulares e orientadas. Este algoritmo foi o primeiro algoritmo polinomial envolvendo

transposições.

Kececioglu e Ravi consideraram o estudo do problema de ordenação de genomas por reversões e translocações. Eles apresentaram um algoritmo de aproximação de fator 2 para a ordenação de permutações não orientadas por reversões e translocações. Eles também propuseram um algoritmo com fator 1,5 para a versão orientada do problema. Ambos os algoritmos possuem complexidade $O(n^2)$, onde n é o número de genes distintos no genoma [90].

Quando os números de cromossomos de dois genomas são diferentes, inevitavelmente é necessária a utilização dos eventos de fusão ou fissão. Esse fato abre espaço para a definição de um problema ainda mais amplo: a ordenação de genomas por reversões, translocações, fusões e fissões, que consiste em encontrar a menor série destes eventos para transformar um genoma em outro.

Hannenhalli e Pevzner consideraram a versão orientada do problema e produziram um algoritmo polinomial exato com complexidade $O(n^4)$ [76].

1.3 Ordenação de permutações por reversões orientadas

Nesta seção apresentaremos alguns conceitos básicos associados ao problema de ordenação de permutações por reversões orientadas.

1.3.1 Grafo de *breakpoints*

Produzir uma sequência ótima de reversões que resolva o problema de ordenação de permutações por reversões orientadas ou apenas calcular a distância de reversão de uma permutação orientada podem ser feitos em tempo polinomial. A abordagem clássica para a análise destes dois problemas foi desenvolvida por Hannenhalli e Pevzner [18,78] e ela se baseia numa estrutura especial denominada "Grafo de *Breakpoints*".

O grafo de *breakpoints* de uma permutação π é denotado por $G(\pi)$. Para uma dada permutação $\pi = (\pi_1, \pi_2, \ldots, \pi_{n-1}, \pi_n)$, podemos construir um grafo de *breakpoints* $G(\pi)$ da seguinte maneira:

1. Se π representa um genoma linear, devemos adicionar os valores 0 e $n+1$ para que eles representem as extremidades do cromossomo. Dessa maneira,

nós obtemos a permutação $\pi' = (0, \pi_1, \pi_2, \ldots, \pi_{n-1}, \pi_n, n+1)$. Se π é circular, devemos apenas adicionar o valor 0. Neste caso, podemos assumir, sem perda de generalidade, $\pi_n = n$ e, assim, temos $\pi' = (0, \pi_1, \pi_2, \ldots, \pi_{n-1}, n)$. Um *breakpoint* é definido por um par (π_i, π_{i+1}), tal que $(\pi_{i+1} - 1) \neq \pi_i$ com $0 \leq i \leq n$ para permutações lineares ou $0 \leq i \leq n-1$ para permutações circulares.

2. Os elementos da permutação π' definem vértices do grafo de *breakpoints*. Escrevendo os elementos de π', sequencialmente, do elemento 0 ao elemento $n+1$ (ou n para permutações circulares) podemos ligar os elementos consecutivos com *arestas pretas*. Desse modo, temos dois vértices entre cada par de elementos consecutivos de π' (um vértice em cada extremidade da aresta preta). No total, temos $2n+2$ vértices e $n+1$ arestas pretas para permutações lineares e $2n$ vértices e n arestas pretas para as permutações circulares.

3. Finalmente, a construção do grafo é concluída com a adição das *arestas cinzas* que fazem a ligação entre as extremidades das arestas pretas. Se um dado vértice v possui um valor $i \geq 0$, devemos selecionar a primeira extremidade da aresta preta situada depois do vértice v. Caso contrário ($i < 0$), devemos selecionar a última extremidade da aresta situada antes de v. Feito isso, devemos procurar o vértice u associado ao valor j, tal que $|j| = |i| + 1$. Se $j > 0$, devemos ligar a extremidade da aresta preta selecionada anteriormente com a última extremidade da aresta preta localizada antes de u. Caso contrário ($j < 0$), a ligação é feita com a primeira extremidade da aresta preta situada após u.

A Figura 1.3 exibe a construção de um grafo de *breakpoints* para a permutação linear $(-3, +2, +1, -4)$. A Figura 1.4(a) exibe a grafo *breakpoints* equivalente a permutação circular $(-3, +2, +1, -4)$. Podemos ver que, para um mesma permutação de números, os grafos de *breakpoints* podem ser bastantes diferentes se a permutação estiver associada a um cromossomo linear ou circular. No entanto, estes grafos são analisados da mesma maneira e, assim, sem perda de generalidade, as discussões sobre os grafos podem ser feitas sem a necessidade de se especificar se a permutação é linear ou circular.

Ao final do processo de construção do grafo, teremos uma coleção de ciclos que alternam entre arestas pretas e arestas cinzas. Um ciclo que contém apenas uma

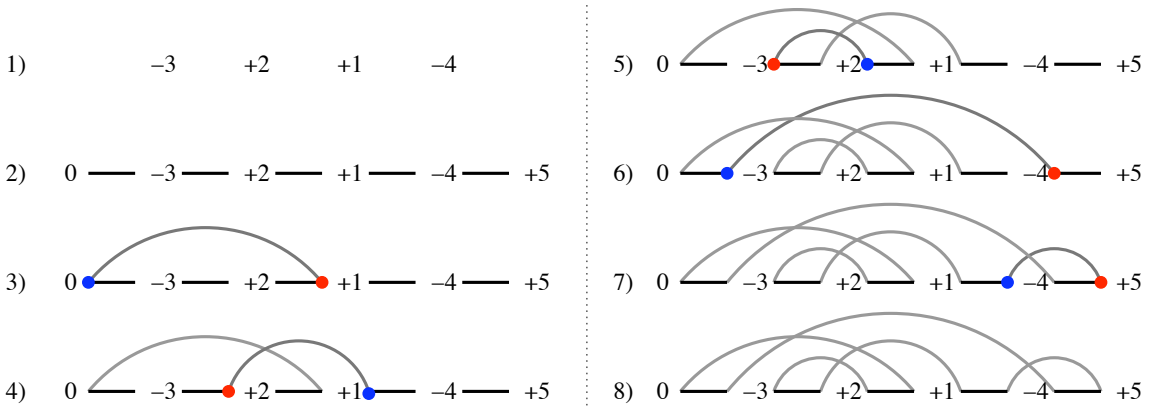


Figura 1.3: Construção passo a passo do grafo de *breakpoints* da permutação linear $(-3, +2, +1, -4)$. O primeiro passo consiste na adição dos valores 0 e $n+1$ e ligação dos pares de valores consecutivos por arestas pretas (arestas horizontais). Os passos seguintes realizam as ligações das extremidades das arestas pretas com arestas cinzas (arcos). Se um vértice v tem um valor $i \geq 0$, devemos selecionar a primeira extremidade da aresta preta situada após v. Caso contrário devemos selecionar a última extremidade da aresta preta situada antes de v. Esta seleção é representada pelos círculos azuis. Feito isso, devemos procurar o vértice u associado ao valor j, tal que $|j| = |i| + 1$. Se $j > 0$, devemos selecionar a última extremidade da aresta preta situada antes de u. Caso contrário, devemos selecionar a primeira extremidade da aresta preta localizada após u. Esta seleção é representada pelo círculo vermelho. Um arco ligando as duas extremidades selecionadas é adicionado ao grafo.



Figura 1.4: (a) Grafo de *breakpoints* da permutação circular $(-3, +2, +1, -4)$. Note que, neste caso apenas o valor 0 é adicionado e a permutação é reorganizada de modo que $\pi_4 = 4$. Para ser reorganizada, a permutação inteira foi invertida de modo que o elemento -4 ficasse com sinal positivo. Feito isso, todos os elementos foram deslocados (de forma circular) até que $\pi_4 = 4$. (b) Grafo de *breakpoints* da permutação identidade linear $\iota_3 = (+1, +2, +3)$, que é composto por 4 ciclos triviais. Note que, com o propósito de obter melhor uma visualização, ao longo do texto desta tese os ciclos triviais serão representados por uma aresta e um arco pretos.

aresta preta e uma aresta cinza é denominado *ciclo trivial* e ele cobre uma adjacência. Como uma permutação identidade $\iota_n = (+1, +2, \ldots, n-1, n)$ é composta apenas por adjacências, o seu grafo de *breakpoints* deve conter apenas ciclos triviais. A Figura 1.4(b) exibe o grafo da permutação identidade linear composta por 3 elementos ($\iota_3 = (+1, +2, +3)$).

Um ciclo que contém 4 ou mais arestas cobre pelo menos dois *breakpoints* e é denominado *ciclo longo*. Como um ciclo é sempre composto pelo mesmo número de arestas pretas e cinzas, podemos utilizar a notação de *k-ciclos* para facilitar a descrição deste elemento. Um *k-ciclo* é um ciclo composto por k arestas pretas (e, consequentemente, k arestas cinzas). Logo, um ciclo trivial é um 1-*ciclo* e um 2-*ciclo* é um ciclo longo de tamanho mínimo.

Após a construção dos ciclos, podemos impor direções às arestas pretas de acordo com um caminho arbitrário. Atribuindo uma direção positiva para uma aresta (seta para a direita), seguimos o arco até a próxima aresta preta e desenhamos uma nova seta de acordo com a extremidade de entrada: seta para a direita quando a entrada ocorre na extremidade inicial da aresta e seta para a esquerda (direção negativa) para quando a entrada ocorre na outra extremidade. Se um ciclo possui arestas pretas com direções positivas e negativas, dizemos que ele é um ciclo orientado. Caso contrário, se o ciclo possui apenas direções positivas ou apenas direções negativas, dizemos que ele é um ciclo não orientado. A Figura 1.5 mostra a atribuição passo a passo das direções da arestas pretas no grafo de *breakpoints* da permutação $(-3, +2, +1, -4)$.

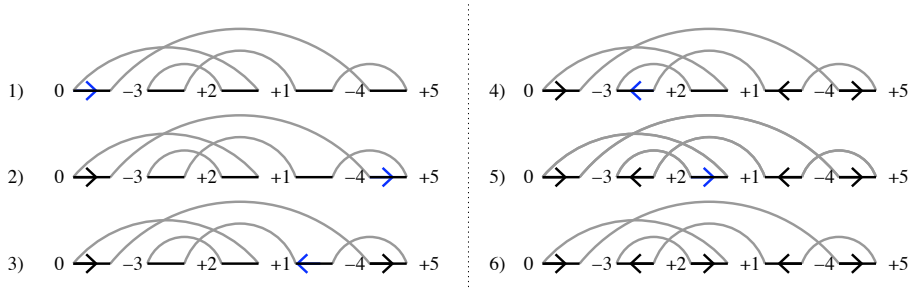


Figura 1.5: Exemplo de atribuição de orientações das arestas pretas de um grafo de *breakpoints*. Neste exemplo, como temos um ciclo com arestas pretas com as duas direções, dizemos que ele é um ciclo orientado. Se o ciclo tivesse arestas pretas com apenas uma direção, ele seria um ciclo não orientado.

Uma componente do grafo é uma coleção de ciclos que se sobrepõem da seguinte maneira: ao menos uma aresta cinza de um ciclo da componente tem uma sobreposição com uma aresta cinza de outro ciclo da componente. Adjacências são componentes triviais e uma componente não-trivial contém pelo menos dois *breakpoints*. Quando uma componente possui ao menos um ciclo orientado, ela é uma componente orientada. Caso contrário, quando todos os ciclos são não orientados, temos uma componente não orientada. A Figura 1.6 exibe exemplos destes dois tipos de componentes.



(a) Componente orientada (b) Componente não orientada

Figura 1.6: Exemplos de tipos de componentes

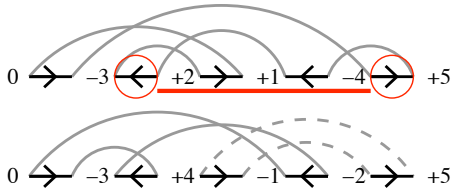
1.3.2 Tipos de reversões

Hannenhalli e Pevzner descreveram os efeitos de uma reversão ρ sobre um grafo de *breakpoints* [76, 78, 130]. Seja $c(\pi)$ o número de ciclos de um grafo de *breakpoints*. Hannenhalli e Pevzner demonstraram que um reversão ρ pode ser classificada em um dos três seguintes tipos de reversão:

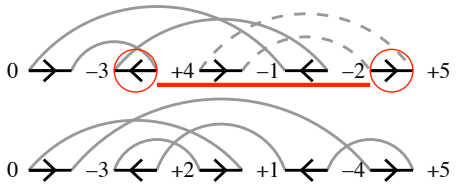
- `Reversão de quebra` (*split reversal*): Reversão que aumenta o número de ciclos do grafo de *breakpoints* em uma unidade. Neste caso, temos que $c(\pi \circ \rho) = c(\pi) + 1$. A Figura 1.7(a) exibe um exemplo de uma reversão de quebra.

- `Reversão de junção` (*joint reversal*): Reversão que diminui o número de ciclos do grafo de *breakpoints* em uma unidade. Neste caso, temos que $c(\pi \circ \rho) = c(\pi) - 1$. A Figura 1.7(b) exibe um exemplo de uma reversão de junção.

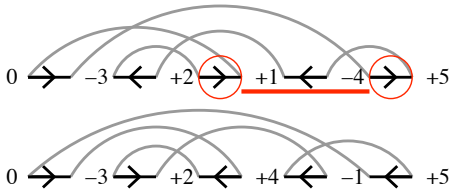
- `Reversão neutra` (*neutral reversal*): Reversão que não altera o número de ciclos do grafo de *breakpoints*. Neste caso, temos que $c(\pi \circ \rho) = c(\pi)$. A Figura 1.7(c) exibe um exemplo de uma reversão neutra.



(a) Reversão de quebra (*split reversal*)



(b) Reversão de junção (*joint reversal*)



(c) Reversão neutra (*neutral reversal*)

Figura 1.7: Exemplos de reversões de quebra, de junção e neutra

Com base nas direções das arestas pretas dos ciclos do grafo de *breakpoints*, podemos caracterizar estes três tipos de reversões. Se as extremidades de uma reversão são arestas pretas de um mesmo ciclo e elas possuem direções opostas, temos uma reversão de quebra. Se as extremidades da reversão estão em arestas pretas em diferentes ciclos, independentemente das direções, temos uma reversão de junção. Finalmente, se as extremidades da reversão estão em arestas pretas no mesmo ciclo e com mesma direção, temos uma reversão neutra.

Para entender as razões destes efeitos, devemos investigar como elas afetam a topologia do grafo. De fato, apenas as duas arestas pretas nas extremidades da reversão são modificadas. No caso das demais arestas pretas, mesmo que os elementos da permutação localizados entre elas tenham sinais trocados, elas permanecem inalteradas.

1.3.3 Cálculo da distância de reversão

Seja $b(\pi)$ o número de arestas pretas de um grafo de *breakpoints*. Quando uma permutação está ordenada, ou seja, quando ela é a permutação ι_n, temos que o número de ciclos triviais é igual ao número de arestas pretas e neste caso $b(\pi) = c(\pi)$.

Se a permutação não está ordenada, isto significa que temos pelo menos um ciclo

não trivial na permutação e que $b(\pi) > c(\pi)$. Como queremos aumentar o número de ciclos no grafo, devemos maximizar o número de reversões de quebra. Baseado nesta observação, podemos iniciar a construção de uma fórmula que calcula a distância de reversão $d(\pi)$ de uma permutação.

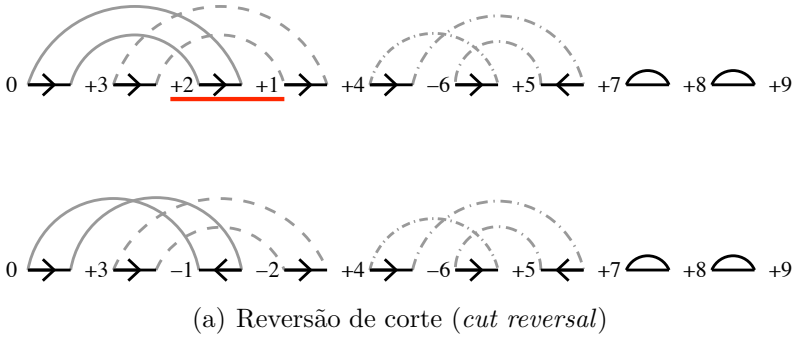
Se podemos encontrar uma sequência s de reversões que tenha apenas reversões de quebra, o tamanho de s é $b(\pi) - c(\pi)$. Contudo, uma reversão de quebra nem sempre existe. Por exemplo, se todas as arestas pretas de todos os ciclos não triviais de um grafo têm a mesma direção, não podemos realizar uma reversão de quebra. Assim, em alguns casos temos que adicionar à sequência s reversões de junção ou neutras para concluir a ordenação da permutação e, neste caso, $d(\pi) \geq b(\pi) - c(\pi)$.

Uma reversão ρ é denominada reversão de corte (*cut reversal*) quando as suas extremidades estão em um mesmo ciclo de uma componente não orientada. Uma reversão de corte é sempre neutra e transforma a componente não orientada em uma componente orientada. Assim, dizemos que uma reversão de corte elimina uma componente não orientada sem mudar o número de ciclos de um grafo de *breakpoints*. A Figura 1.8(a) exibe um exemplo de uma reversão de corte.

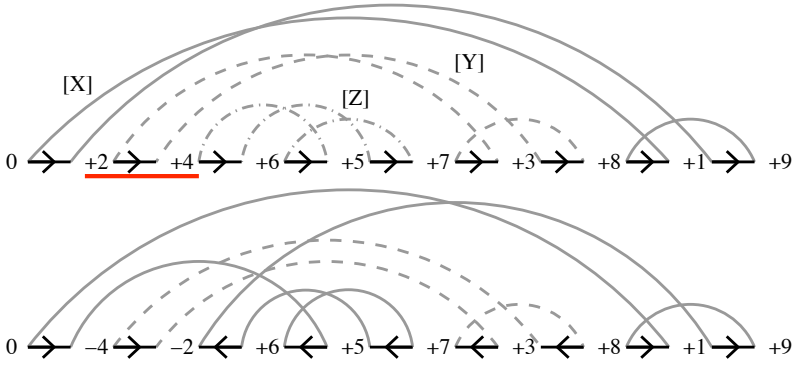
Quando um grafo de *breakpoints* possui mais de uma componente não orientada, nem sempre é necessário usar uma reversão de corte para cada componente não orientada. Uma componente não orientada Y separa duas outras componentes não orientadas X e Z quando existe uma aresta preta de Y entre qualquer aresta preta de X e qualquer aresta preta de Z. Neste caso, uma reversão que tem uma extremidade em X e uma extremidade em Z agrupará as componentes X, Y e Z em uma componente orientada. Este tipo de reversão é denominada reversão de união (*merge reversal*). A Figura 1.8(b) exibe uma exemplo de uma reversão de união.

Uma reversão de união é sempre uma reversão de junção que agrupa $i \geq 2$ componentes não orientados. Assim, uma reversão de união elimina $i \geq 2$ componentes não orientados reduzindo em uma unidade o número de ciclos do grafo de *breakpoints*.

Uma componente não orientada que não separa duas outras componentes não orientadas é denominada obstáculo (*hurdle*). O valor $h(\pi)$ representa o número de obstáculos de um grafo de *breakpoints*. Como um obstáculo não separa componentes orientadas, cada obstáculo X pode ser eliminada por uma reversão de corte cujas extremidades de corte são pontos no mesmo ciclo de X. Outra opção é eliminar X em conjunto com outro obstáculo Z através de uma reversão de união cujas extremidades estão em um ponto de X e um ponto de Z. Uma reversão de corte elimina um obstáculo e não altera o número de ciclos enquanto uma reversão de união



(a) Reversão de corte (*cut reversal*)



(b) Reversão de união (*merge reversal*)

Figura 1.8: (a) Uma reversão de corte transforma uma componente não orientada em uma componente orientada e não altera o número de ciclos do grafo de *breakpoints*. (b) A componente não orientada Y separa as componentes não orientadas X e Z. Uma reversão de união agrupa as componentes X, Y e Z em uma única componente orientada e diminui o número de ciclos do grafo de *breakpoints* em uma unidade.

elimina dois obstáculos de uma vez e diminui o número de ciclos de um grafo em uma unidade. Portanto, cada obstáculo necessita uma reversão adicional. Com base nesta informação podemos atualizar a fórmula para cálculo de distância de reversão: $d(\pi) \geq b(\pi) - c(\pi) + h(\pi)$.

Dizemos que um obstáculo Z protege uma componente não orientada Y que não é um obstáculo, se Y se transforma em um obstáculo depois da eliminação de Z com uma reversão de corte. Neste caso, o obstáculo Z é denominado *super-obstáculo* (Figura 1.9). Eliminar um super-obstáculo com uma reversão de corte não diminui o número de obstáculos neste grafo e, por isso, um super obstáculo deve ser sempre eliminado em conjunto com outro super-obstáculo com uma reversão de união que agrupa os dois super obstáculos e suas componentes não orientadas protegidas em uma única componente não orientada (Figura 1.8(b)).

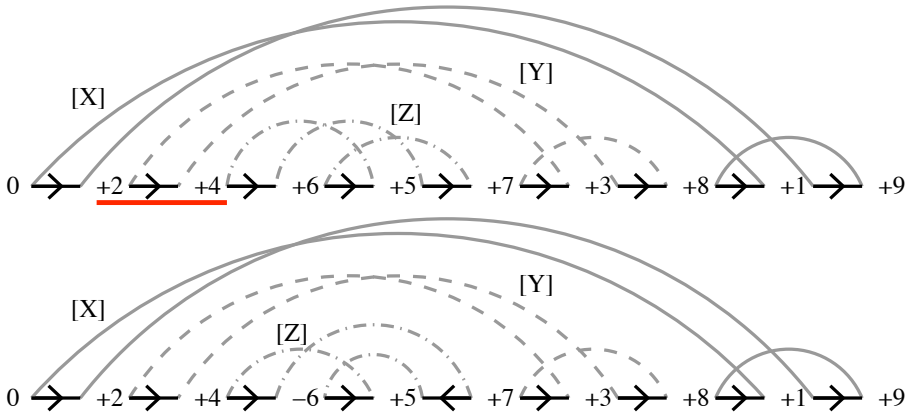


Figura 1.9: A componente orientada Y separa os dois super-obstáculos X e Z. Após eliminar o super-obstáculo Z com uma reversão de corte, o componente Y se transforma em um obstáculo e, portanto, o número de obstáculos no grafo não é reduzido.

Para completar a fórmula, precisamos analisar um último caso. Quando todas os i obstáculos de um grafo de *breakpoints* são super-obstáculos e i é um número ímpar, a permutação necessita de um esforço adicional para ser ordenada. Um grafo de *breakpoints* que tenha esta característica é denominado *fortaleza*. A Figura 1.10 exibe um exemplo de fortaleza.

Uma reverão adicional é suficiente para eliminar a fortaleza. Esta reversão deve

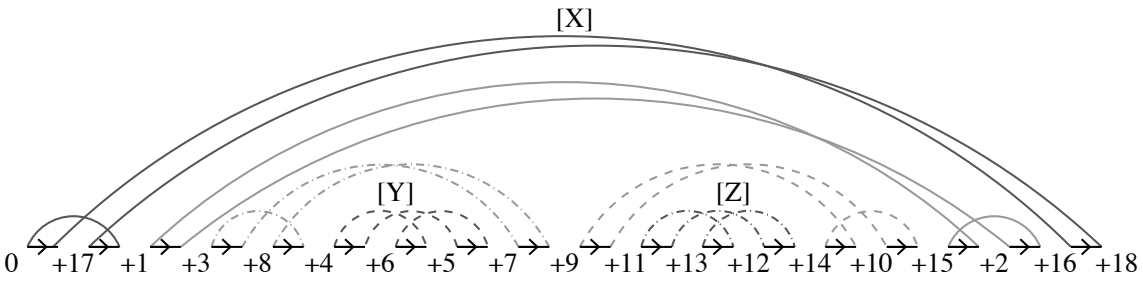


Figura 1.10: Uma fortaleza composta por 3 super-obstáculos (X, Y, Z)

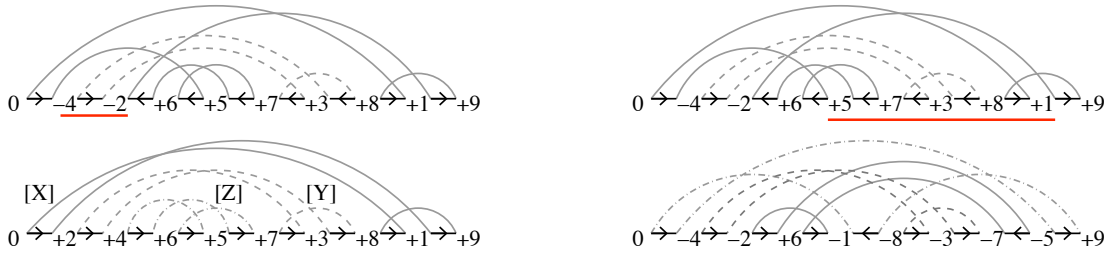
ser escolhida entre diversas possibilidades como, por exemplo, uma reversão de corte para eliminar um obstáculo ou uma reversão de união para unir dois obstáculos. Denotamos por $f(\pi)$ o valor que indica se o grafo de *breakpoints* possui uma fortaleza ou não. Se o grafo possui uma fortaleza, $f(\pi) = 1$, caso contrário, $f(\pi) = 0$. Portanto, a fórmula completa para a o cálculo da distância de reversão de uma permutação é dada por:

$$d(\pi) = b(\pi) - c(\pi) + h(\pi) + f(\pi)$$

1.3.4 Reversões seguras e inseguras

Se um grafo de *breakpoints* não possui componentes não orientadas, então ele pode ser ordenado com apenas reversões de quebra. Contudo, se nenhum cuidado é tomado na seleção das reversões aplicadas, algumas delas podem produzir novos obstáculos o que, claramente, é um efeito indesejável. Uma reversão de quebra que produz obstáculos é chamada *reversão insegura* (Figura 1.11(a)), caso contrário, ela é chamada *reversão segura* (Figura 1.11(b)).

Em geral, obstáculos são raros e fortalezas ainda mais raras em permutações que representam genomas [19]. Na prática, reversões de quebra são suficiente para ordenar a maioria das permutações e o principal problema é encontrar as reversões seguras. Uma maneira simples de encontrar estas reversões seria testar cada reversão de quebra para verificar se ela é segura ou não, até que uma reversão segura seja encontrada. Felizmente, existem maneiras mais rápidas de se selecionar uma reversão segura e, uma delas, é baseada em uma estrutura, relacionada ao grafo de *break-*



(a) Reversão insegura (*unsafe reversal*) (b) Reversão segura (*safe reversal*)

Figura 1.11: (a) Uma reversão insegura quebra um ciclo em dois, mas cria três componentes não orientadas X, Y e Z. (b) Uma reversão segura quebra o ciclo em dois sem criar componentes não orientadas.

points, denominada grafo de sobreposição (*overlap graph*) [88]. Com esta estrutura, uma reversão segura pode ser selecionada em tempo $O(n)$.

Capítulo 2

Enumeração de *traces*

No Capítulo 1 apresentamos resultados ligados ao problema de ordenação de permutações por reversões que estão presentes na literatura (Seção 1.2.1). Entre estes resultados, mencionamos a existência de algoritmos polinomiais para a resolução da versão orientada do problema. Estes algoritmos são capazes de produzir como saída uma única solução ótima que transforma uma permutação em outra.

Contudo, o espaço de soluções ótimas é frequentemente imenso e não existe garantia de que a solução produzida seja aquela que realmente ocorreu durante a evolução das duas espécies comparadas.

Uma alternativa para este problema seria a enumeração do conjunto completo de soluções ótimas. Porém, como o tamanho do conjunto de soluções ótimas cresce exponencialmente com o tamanho e com a distância de reversão da permutação, seria interessante a existência de uma representação mais compacta para o conjunto.

Para ilustrar o crescimento exponencial do conjunto de soluções, podemos observar o comportamento apresentado pela família de permutações $(2, -3, \ldots, -n, 1)$ $(n \geq 3)$. Esta família é composta por permutações que não possuem obstáculos e que possuem distância de reversão igual a n. A Tabela 2.1 mostra o número de soluções de cada membro da família quando variamos o valor de n entre 3 e 15.

Uma maneira mais compacta de se representar este imenso conjunto de soluções seria através da utilização de *traces*. A Tabela 2.1 lista o número de *traces* obtidos para cada permutação. Podemos ver claramente que apesar do número de *traces* também crescer com o valor de n, este número é muito menor do que o número de soluções.

Neste capítulo, abordaremos o problema de enumeração de *traces*. Apresentare-

Tabela 2.1: Número de soluções e de *traces* da família de permutações $(+2, -3, \ldots, -n, +1)$ para $3 \leq n \leq 15$.

n	Permutação	# Soluções	# Traces
3	+2,−3,+1	3	1
4	+2,−3,−4,+1	12	1
5	+2,−3,−4,−5,+1	72	5
6	+2,−3,−4,−5,−6,+1	490	15
7	+2,−3,−4,−5,−6,−7,+1	3764	51
8	+2,−3,−4,−5,−6,−7,−8,+1	33412	249
9	+2,−3,−4,−5,−6,−7,−8,−9,+1	329186	917
10	+2,−3,−4,−5,−6,−7,−8,−9,−10,+1	3615306	5367
11	+2,−3,−4,−5,−6,−7,−8,−9,−10,−11,+1	43519094	23253
12	+2,−3,−4,−5,−6,−7,−8,−9,−10,−11,−12,+1	571826970	150173
13	+2,−3,−4,−5,−6,−7,−8,−9,−10,−11,−12,−13,+1	8132010290	757785
14	+2,−3,−4,−5,−6,−7,−8,−9,−10,−11,−12,−13,−14,+1	124552037342	5278601
15	+2,−3,−4,−5,−6,−7,−8,−9,−10,−11,−12,−13,−14,−15,+1	2043405755112	30326675

mos um pequeno exemplo da utilidade da enumeração de *traces* e detalharemos o trabalho de otimização através da adoção de uma nova estrutura de dados.

2.1 *Traces*

Em 2002, Bergeron *et al*. introduziram o conceito de *traces* ao problema de ordenação de permutações por reversões [19]. Porém, antes de explicarmos o conceito de *traces*, devemos apresentar os conceitos de *intervalo* e de *sobreposição de intervalos*.

2.1.1 Intervalos

Dizemos que um subconjunto de números $\rho \subseteq \{1, \ldots, n\}$ é um *intervalo* de uma permutação π se existem $i, j \in \{1, \ldots, n\}$, $0 \leq i \leq j \leq n$, tais que $\rho = \{|\pi_i|, \ldots, |\pi_j|\}$. Por exemplo, se $\pi = (+1, -5, +3, +2, -4, -6)$, $i = 2$ e $j = 4$, $\rho_{i,j} = \{5, 3, 2\}$.

Note que, um intervalo lista elementos que estão contíguos em uma dada permutação. Por esta razão, a ordem em que os elementos estão listados dentro do intervalo não possui importância. Por outro lado, podemos impor à escrita dos elementos de um intervalo, a utilização da ordem lexicográfica determinada pelo al-

fabeto dos identificadores. A utilização desta representação permite a comparação de diferentes intervalos através da utilização da ordem lexicográfica.

Como as reversões afetam elementos que estão contíguos em uma permutação, podemos utilizar a notação de intervalo para representá-las. Esta notação será utilizada ao longo do texto desta tese.

Dois intervalos apresentam uma *sobreposição* se eles possuem uma interseção sem estarem contidos um no outro. Por exemplo, se $\pi = (+1, -4, +3, +2, -5, -6)$, então $\rho_1 = \{1, 3, 4\}$ e $\rho_2 = \{2, 3, 4\}$ se sobrepõem, enquanto $\rho_3 = \{2, 3, 4, 5\}$ e $\rho_4 = \{2, 3\}$ não se sobrepõem.

2.1.2 Classe de equivalência

Seja $\mathcal{A}$ o alfabeto composto por todas as reversões que podem ser aplicadas em uma permutação. Uma sequência de reversões pode ser considerada como uma palavra neste alfabeto.

Bergeron *et al.* definem uma relação de equivalência entre palavras. Se ρ e θ são reversões e não se sobrepõem, então as palavras $\rho\theta$ e $\theta\rho$ são equivalentes. Neste caso, dizemos que ρ e θ comutam.

Com base nesta relação, qualquer palavra contendo $\rho\theta$ como sub-palavra é equivalente a mesma palavra que substitui $\rho\theta$ por $\theta\rho$. Por exemplo, para a permutação $(+4, -3, -1, +2)$, a sequência de reversões $\{1, 3, 4\}\{2, 4\}\{2, 3\}\{3\}$ é uma solução. Neste caso, a reversão $\{3\}$ comuta com todas as outras reversões e nenhuma das outras reversões comutam entre si. Assim, as sequências $\{1, 3, 4\}\{2, 4\}\{3\}\{2, 3\}$, $\{1, 3, 4\}\{3\}\{2, 4\}\{2, 3\}$ e $\{3\}\{1, 3, 4\}\{2, 4\}\{2, 3\}$ são soluções equivalentes pela comutação de $\{3\}$ com todas as outras reversões.

Se considerarmos a sequência de reversões $\{1\}\{1, 2\}\{4\}\{1, 2, 3, 4\}$, temos que $\{4\}$ e $\{1, 2\}$ comutam e, por isso, $\{1\}\{1, 2\}\{4\}\{1, 2, 3, 4\}$ é equivalente à sequência $\{1\}\{4\}\{1, 2\}\{1, 2, 3, 4\}$. Na realidade, neste exemplo, todas as reversões comutam entre si e, por isso, todas as permutações destas quatro reversões também são soluções.

Uma classe de equivalência de sequências ótimas de reversões sobre a relação de equivalência descrita acima é denominada *trace*. Bergeron *et al.* propõem que para uma permutação orientada π, o conjunto de todas as reversões ótimas é uma união de *traces*.

Como consequência, temos que se o conjunto de soluções que devem ser enumeradas é muito grande, o conjunto de *traces* pode ser um resultado mais relevante para

o problema de ordenação por reversões.

Porém, para que *traces* possam realmente ser utilizados para representar conjuntos de soluções, precisamos que eles possuam uma forma de representação única.

2.1.3 Forma Normal de um *Trace*

Um elemento s de um *trace* T está em sua forma normal se ele pode ser decomposto em sub-palavras $s = u_1|\ldots|u_m$ tais que:

- todo par de elementos de uma sub-palavra u_i comuta;
- para todo elemento ρ de uma sub-palavra u_i $(i > 1)$, existe pelo menos um elemento θ da palavra u_{i-1} tal que ρ e θ não comutam;
- toda sub-palavra u_i é uma palavra crescente não vazia com relação à ordem lexicográfica induzida por $\mathcal{A}$.

A *altura* de um *trace* é definida pelo número de sub-palavras que ele possui. O tamanho de um *trace* é definido pelo número de soluções ótimas que ele representa.

Segundo um teorema de Cartier e Foata [38], para qualquer *trace* existe uma única palavra que está na forma normal. Assim, é possível representar os *traces* através das suas representações na forma normal. No caso da permutação $(+4, -3, -1, +2)$, por exemplo, temos dois *traces* de soluções ótimas:

$$\{1\}\{1,2\}\{1,2,3,4\}\{4\} \text{ e } \{1,3,4\}\{3\}|\{2,4\}|\{2,3\}.$$

O *trace* $\{1\}\{1,2\}\{1,2,3,4\}\{4\}$ possui altura 1 e tamanho 24, que é o número de total combinações que se pode obter alterando a ordem das 4 reversões que não se sobrepõem.

O *trace* $\{1,3,4\}\{3\}|\{2,4\}|\{2,3\}$ possui altura 3 e tamanho 4. Como apenas a reversão $\{3\}$ não apresenta sobreposição com as demais reversões, temos apenas 4 soluções equivalentes que são obtidas através da alteração da ordem em que $\{3\}$ é aplicada na sequência:

$$\begin{array}{ll} \{3\}\{1,3,4\}\{2,4\}\{2,3\} & \{1,3,4\}\{3\}\{2,4\}\{2,3\} \\ \{1,3,4\}\{2,4\}\{3\}\{2,3\} & \{1,3,4\}\{2,4\}\{2,3\}\{3\} \end{array}$$

Neste exemplo, com apenas duas formas normais de *traces* é possível descrever o conjunto completo de 28 soluções de uma maneira compacta.

2.2 Enumeração de soluções

Seja $d(\pi)$ a distância de reversão da permutação π em relação à permutação identidade. Uma sequência de reversões $s = \rho_1\rho_2\ldots\rho_i$ é chamada de *optimal i-sequence* quando $d(\pi \circ \rho_1\rho_2\ldots\rho_i) = d(\pi) - i$. Note que se $i = d(\pi)$, então s é uma solução ótima.

Em 2003, Siepel propôs um algoritmo capaz de enumerar todas as *optimal* 1-*sequences* de uma permutação orientada em tempo $O(n^3)$ [145]. Através da iteração deste algoritmo, podemos ver facilmente que é possível produzir a enumeração de todas as soluções que ordenam uma permutação.

Por exemplo, suponha que $s = \rho_1\rho_2\ldots\rho_i$ e que $s \in I$, onde I é a lista de todas as *optimal i-sequences* da permutação π. Se aplicarmos a sequência s em π, obteremos uma nova permutação π', tal que, $d(\pi') = d(\pi) - i$. Submetendo π' ao algoritmo proposto por Siepel, obteremos uma lista L de *optimal* 1-*sequences* de π'. Através da adição de cada uma das reversões presentes em L à sequência s, obteremos um conjunto de *optimal* $(i+1)$-*sequences*. Repetindo o processo para todas as demais *optimal i-sequences* presentes em I, ao final, teremos todas as *optimal* $(i+1)$-*sequences* de π.

Braga *et al*. mostraram que complexidade do algoritmo de enumeração de soluções proposto por Siepel é $O(n^{2n+3})$ [30].

2.3 Enumeração de *traces* de soluções

Em 2007, as idéias apresentadas por Bergeron *et al*. e Siepel foram combinadas por Braga *et al*. para a produção de um algoritmo capaz de enumerar todos os *traces* de soluções que ordenam uma permutação por reversões [30].

Para qualquer *optimal i-sequence* s, podemos construir um *i-trace* através da aplicação dos critérios que definem a forma normal de um *trace*. O *trace* obtido representa uma sequência de i reversões e pode ser denominado de *i-trace*. Além de representar a solução definida por s, este *i-trace* representa todas as soluções equivalentes a s.

Um *i-trace* é um *trace* que possui i reversões. Um novo $(i+1)$-*trace* s' pode ser construído através da adição de uma nova reversão ρ que não está contida em um *i-trace* s. Para realizar esta operação de acordo com a forma normal, devemos procurar o máximo valor de j $(1 \leq j \leq m)$ tal que a sub-palavra u_j contém uma

reversão θ que não comuta com ρ. Se tal sub-palavra u_j existe, temos que inserir a reversão ρ na sub-palavra u_{j+1}. Se $j = m$, criamos uma nova sub-palavra u_{m+1} que terá apenas a reversão ρ e a concatenamos no final do *i-trace*. Finalmente, se a sub-palavra u_j não existe, isso significa que nenhuma reversão presente em s sobrepõe ρ e, por isso, a inserção deve ocorrer na sub-palavra u_1.

Um *k-trace* T' é um prefixo de um *i-trace* T ($k \leq i$) se T' contém uma *optimal k-sequence* que é prefixo de uma *optimal i-sequence* de T. Isto é equivalente a dizer que cada *optimal k-sequence* de T' é um prefixo de uma *optimal i-sequence* de T.

Todo prefixo de tamanho k de uma *optimal i-sequence* pode ser representado por um *k-trace* de *optimal k-sequences*. Assim, ao invés de realizar a enumeração de todas as *optimal i-sequences* para posterior cálculo e comparação de todos os *traces*, torna-se mais vantajoso o processo de enumeração e comparação direta de todos os *i-traces*.

Da mesma maneira que um conjunto de *optimal* $(i+1)$*-sequences* pode ser produzido a partir de um conjunto de *optimal i-sequences*, um conjunto de $(i+1)$*-traces* pode ser produzido a partir de um conjunto de *i-traces*. Este processo pode ser aplicado de maneira incremental sem a necessidade de se computar todas as soluções.

A cada iteração i do algoritmo de enumeração de *traces*, um conjunto $\mathcal{T}$ armazena todas as formas normais dos *i-traces* calculados.

Para $i = 1$, cada 1*-trace* é obtido através da execução do algoritmo de enumeração de *optimal* 1*-sequences* de Siepel sobre a permutação π. Cada *optimal* 1*-sequence* obtida é um 1*-trace* de π.

No início de cada passo i ($2 \leq i \leq d(\pi)$), $\mathcal{T}$ contém todas as formas normais de todos os $(i-1)$*-traces* de π. Selecionando-se um $(i-1)$*-trace* t, podemos removê-lo de $\mathcal{T}$ e aplicá-lo à permutação π. Assim, como no caso da enumeração de soluções, obteremos uma nova permutação π' que será submetida ao algoritmo de Siepel para a produção de uma lista L de *optimal* 1*-sequences*. Ao adicionar cada *optimal* 1*-sequence* de L em t, construiremos um novo *i-trace* que será colocado no conjunto $\mathcal{T}$. Repetindo-se o processo para todos os $(i-1)$*-traces* de $\mathcal{T}$, produziremos o conjunto de todos os *i-traces* de π.

A cardinalidade (ou tamanho) de um *i-trace* t é a soma das cardinalidades de seus $(i-1)$ prefixos. Essa propriedade permite que a cardinalidade de cada *trace* seja computada e que assim, o número total de soluções de uma permutação π possa ser obtido.

Para a realização deste cálculo, cada *i-trace* possui um valor de tamanho associado

a ele. Inicialmente, todo 1-*trace* possui tamanho 1. À medida que cada i-*trace* t é calculado, duas possibilidades podem ocorrer:

1. $t \notin \mathcal{T}$: t é um novo i-*trace* e o tamanho dele é igual ao tamanho do $(i-1)$-*trace* que foi usado para construí-lo;

2. $t \in \mathcal{T}$: t já foi listado anteriormente. Seu novo tamanho é igual ao tamanho atual somado ao tamanho do $(i-1)$-*trace* que foi usado para construí-lo.

Por exemplo, suponha a permutação $\pi = (-1, -3, -2)$. A lista de *optimal* 1-*sequences* de π contém as reversões: $\rho_1 = \{1\}$ e $\rho_2 = \{2, 3\}$. Temos, portanto, dois 1-*traces* de tamanho 1. Após aplicarmos ρ_1 em π, a lista de *optimal* 1-*sequences* da permutação resultante terá apenas a reversão ρ_2. Ao combinarmos ρ_1 e ρ_2, teremos o 2-*trace* $\{1\}\{2, 3\}$. Como este 2-*trace* obtido é novo, a ele é atribuído o tamanho 1, que é o mesmo valor de tamanho do 1-*trace* $\{1\}$. Continuando o processo, ao aplicarmos ρ_2 em π, a lista de *optimal* 1-*sequences* da permutação resultante terá apenas a reversão ρ_1. A combinação de ρ_2 e ρ_1 gera o 2-*trace* $\{1\}\{2, 3\}$. Como este 2-*trace* já apareceu anteriormente, devemos somar o tamanho associado a ele com o tamanho do 1-*trace* $\{2, 3\}$. Esta soma tem o valor 2 que é justamente o número de soluções para a ordenação da permutação π.

A complexidade do algoritmo proposto por Braga *et al.* é $O(Nn^{k_{max}+4})$, onde N é o número de d-*traces* e k_{max} é a largura máxima de um d-*trace*. O valor da largura de um *trace* é no mínimo igual ao tamanho máximo de uma sub-palavra u_i na forma normal de um *trace* [30].

2.4 Aplicação biológica da enumeração de *traces*

Como mencionado anteriormente, a enumeração de *traces* de soluções do problema de ordenação de permutações orientadas por reversões visa fornecer suporte aos biólogos para que estes possam estudar cenários alternativos de evolução das espécies.

Um procedimento interessante a ser adotado durante a enumeração de *traces* é a utilização de filtros que descartam reversões que não atendem a determinados critérios reduzindo, desta maneira, o espaço de soluções a ser explorado. Os critérios podem ser criados livremente mas a adoção de filtros com base biológica aumenta a chance de que as soluções produzidas sejam mais relevantes. Na Seção 2.4.1 apresentaremos algumas restrições desenvolvidas por Braga *et al.* [27, 29, 31].

A enumeração de *traces* é normalmente utilizada para realizar a comparação de genomas de duas espécies. Contudo, esta aplicação pode ser estendida para a realização da comparação conjunta de diversas espécies em relação a um ancestral comum a elas. Para demonstrar esta aplicação, apresentaremos na Seção 2.4.2 a comparação que realizamos com bactérias do gênero *Rickettsia*. Este estudo foi apresentado oralmente e publicado nos anais do simpósio *ACM International Symposium on Biocomputing 2010* (`ISB 2010`) sob o título "*Chronological order of reversal events on Rickettsia genus*" [13].

2.4.1 Utilização de restrições biológicas

A utilização de restrições biológicas durante a enumeração de *traces* permite que o espaço de solução a ser explorado seja reduzido. Além disso, temos a chance de enumerar soluções que são mais relevantes do ponto de vista biológico. Braga *et al.* propuseram a implementação de diversas restrições biológicas que serão apresentadas a seguir.

Detecção de intervalos comuns

Agrupamentos de genes co-localizados são intervalos presentes em genomas diferentes e compostos pelo mesmo conjunto de genes. Contudo, estes genes não precisam estar, necessariamente, dispostos nas mesmas ordens e orientações. Tais agrupamentos são modelados como *intervalos comuns*.

Um intervalo comum de duas permutações π e π' é um intervalo de π que está presente em π'. Por exemplo, o intervalo $\{4, 5, 6, 7, 8\}$ é comum às permutações $(+2, +8, +4, -5, +6, -7, -1, -3)$ e ι_8.

Estes intervalos são utilizados para modelar grupo de genes homólogos que podem ser encontrados nas duas espécies representadas pelas permutações. A idéia aqui é que se genes se encontram juntos em ambas as espécies, então é provável que o ancestral comum a elas também possuía os mesmos genes agrupados e que, portanto, eles não foram separados durante a evolução.

Uma reversão ρ quebra um intervalo I se ρ e I apresentam sobreposição. No caso da permutação $(-5, -2, -7, +4, -8, +3, +6, -1)$, por exemplo, a reversão $\{1, 3, 6\}$ quebra o intervalo $\{2, \ldots, 8\}$. Intervalos de tamanho 1 ou n são denominados intervalos comuns triviais. Note que, uma reversão jamais quebra um intervalo comum

trivial.

Em 2001, Heber e Stoye observaram que qualquer intervalo comum pode conter inúmeros intervalos comuns menores. Baseado nesta observação, eles definiram o conceito de *intervalo comum irredutível* como sendo um intervalo comum que não contém nenhum outro intervalo comum diferente dele mesmo [82].

Qualquer intervalo comum I, existente entre duas permutações, possui uma cadeia geradora composta por intervalos comuns irredutíveis $(\gamma_1, \gamma_2, \ldots, \gamma_k)$. Nesta cadeia, $\gamma_1, \gamma_2, \ldots, \gamma_k$ são listados em ordem lexicográfica e, para cada par consecutivo γ_i e γ_{i+1}, temos que $\gamma_i \cap \gamma_{i+1} \neq \emptyset$. Um intervalo comum cuja cadeia geradora tenha tamanho maior ou igual a 2 é um *intervalo comum redutível*. Caso contrário, ele é um *intervalo comum irredutível*.

Por exemplo, a cadeia geradora do intervalo comum redutível $\{3, 4, 5\}$ existente entre as permutações $(+5, -3, +4, -1, -6, -2)$ e ι_6 é $(\{3, 4\}, \{3, 5\})$. Os intervalos comuns $\{3, 4\}$ e $\{3, 5\}$ são irredutíveis.

Braga *et al*. propõem que uma reversão ρ quebra um intervalo comum redutível I se, e somente se, ela quebra ao menos um intervalo comum irredutível da cadeia que gera I [31]. Como consequência desta proposição, se ρ não quebra nenhum intervalo comum irredutível entre π e π', então ρ não quebra nenhum intervalo comum entre π e π'. Enquanto o número de intervalos comuns entre duas permutações de n elementos é limitado por n^2, o número de intervalos comuns irredutíveis é limitado por n [82].

Uma sequência de reversões $\rho_1 \ldots \rho_k$ que ordena uma permutação π é dita "perfeita" se não existe reversão pertencente a ela que sobreponha um intervalo comum de π. Baseado neste conceito, Braga *et al*. propõem uma extensão do algoritmo de enumeração de *traces* para a produção de *traces* de sequências de reversões perfeitas [31].

A extensão do algoritmo se baseia na proposição de que todo *trace* de soluções para uma permutação por reversões com sinais contém somente soluções perfeitas ou não possui nenhuma solução perfeita. Se uma sequência $s = \rho_1 \ldots \rho_k$ que ordena uma permutação π é perfeita, então, por definição nenhum dos elementos $\rho_1, \ldots, \rho_k$ sobrepõem um intervalo comum de π. Assim, qualquer sequência com o mesmo conjunto de reversões, mas em diferentes ordens, é perfeita. Este é o caso para todas as sequências de um *trace*. Portanto, se uma sequência de um *trace* é perfeita, todas as sequências representadas por ele são perfeitas também.

Devido a esta propriedade, um *trace* que contém sequências de reversões perfeitas de tamanho $d(\pi)$ é denominado *trace* perfeito. Tal *trace* nem sempre existe o

que significa que todas as sequências ótimas de reversões devem quebrar intervalos comuns [55].

Além disso, dadas duas permutações π e π', a busca por *traces* perfeitos apresenta características de simetria. Como a lista de intervalos comuns não é alterada quando as reversões são aplicadas, se s é uma sequência perfeita de reversões que transforma π em π', a sequência inversa de s é uma sequência perfeita que transforma π' em π.

A modificação imposta no algoritmo para encontrar os *traces* perfeitos consiste primeiro na identificação dos intervalos comuns às duas permutações. Feito isso, em cada etapa a lista de *optimal* 1-*sequences* é produzida e as reversões que quebram os intervalos comuns são descartadas. No final do processo, restam apenas *traces* perfeitos ou, no caso da inexistência destes *traces*, temos uma resposta vazia.

Uma variação desta abordagem é a flexibilização da restrição de quebra de intervalos comuns. Aceitando um determinado número máximo de quebras de intervalos comuns, podemos identificar *traces* quase-perfeitos.

Ainda dentro do conceito de intervalos comuns, outra tática proposta por Braga, Gautier e Sagot é a detecção progressiva de intervalos comuns. Nesta estratégia, a identificação dos intervalos comuns acontece a cada nível da enumeração e não somente no início o que reduz substancialmente o número de *traces* gerados [29].

Os três tipos de restrições apresentadas acima podem ser aplicadas em cromossomos lineares e circulares. Contudo, apenas a restrição para geração de *traces* perfeitos é simétrica. No caso dos *traces* quase-perfeitos ou na detecção progressiva de intervalos comuns, uma solução obtida pode transformar π em π', mas isso não garante que o inverso desta solução seja capaz de transformar π' em π respeitando as mesmas restrições.

Simetria ao redor do término de replicação

Em cromossomos circulares de procariotos, três tipos de reversões diferentes podem ocorrer. O primeiro tipo é a reversão simétrica ao término de replicação que, como o próprio nome diz, ela afeta a região do término de replicação simetricamente (a reversão atinge fragmentos com aproximadamente o mesmo tamanho nos dois lados do ponto de replicação). O segundo tipo é a reversão assimétrica ao término de replicação e, finalmente, o último tipo é a reversão externa que inverte um bloco localizado entre a origem e o término de replicação.

Apesar da existência de indícios de mecanismos que favoreçam a ocorrência de

reversões simétricas [62], Mackiewicz *et al.* listam diversos argumentos que indicam que a seleção natural desempenha um papel importante neste processo. Através da observação de importantes características dos genes como a distância e a orientação dos genes em relação ao ponto de replicação, Mackiewicz *et al.* apontam que as reversões externa e assimétrica, muitas vezes, geram mutações deletérias [110].

Braga modelou este problema e o adaptou ao algoritmo de enumeração de *traces* [27]. Como uma reversão simétrica não é perfeitamente simétrica, os autores definiram a como sendo a distância entre a primeira extremidade da reversão e o término de replicação e b a distância entre o término de replicação e a outra extremidade da reversão. Baseado nestes dois valores, definiu-se a taxa de simetria que é dada pelo mínimo entre a e b dividido pelo máximo entre a e b. A taxa de simetria é portanto um valor entre 0 e 1 e a diferenciação entre reversões simétricas e assimétricas é feita através de um valor de corte sobre esta taxa.

Dada uma permutação circular $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, o algoritmo considera que os marcadores estão numerados de acordo com a origem de replicação. Assim, π_1 (π_n) representa o marcador que segue (precede) a origem de replicação. Considerando o tamanho dos marcadores, o término de replicação deve cair no meio da permutação.

A restrição de simetria ao término de replicação pode considerar apenas reversões que são simétricas ou ainda ser flexibilizada para aceitar um certo número de reversões assimétricas ou externas. Sendo assim, o algoritmo define três parâmetros: r é a taxa de simetria, p é o número máximo de reversões externas e q é o número máximo de reversões assimétricas.

Se uma reversão ρ possui uma classificação X (externa, simétrica ou assimétrica) em uma sequência de reversões s, não existe garantia que ela possua a mesma classificação X em uma sequência equivalente s'. Assim, frequentemente apenas um subconjunto de sequências de um *trace* T respeitam a restrição (p, q, r). Este subconjunto é denominado *subtrace* (p, q, r)*-induzido*.

A restrição de simetria ao término de replicação é simétrica. Isso indica que, se uma sequência s transforma π em π' respeitando os parâmetros (p, q, r), a sequência inversa de s também respeita a restrição.

Estratificação de permutações

Para analisar o cenário de evolução dos cromossomos humanos X e Y, Braga *et al.* desenvolveram um algoritmo para enumeração de soluções que estratificam uma per-

mutação [31].

Os cromossomos X e Y são muito diferentes. No caso dos humanos, particularmente, o cromossomo X possui tamanho de 155 Mbp enquanto o cromossomo Y mede apenas 58 Mbp. Contudo, acredita-se que ambos evoluíram a partir de um par de cromossomos idênticos [120]. Este processo de diferenciação deu origem a diferenciação sexual através dos pares XX para as mulheres e XY para os homens.

Estes cromossomos ainda compartilham uma região "pseudo-autossomal" em uma de suas extremidades onde a recombinação ocorre como se eles fossem autossomos. Em torno de 90% do cromossomo Y é específico do sexo masculino e contém maiores diferenças na sequência e na ordem dos genes. As teorias atuais indicam que esta região pseudo-autossomal, que na origem cobria os cromossomos inteiros, foi sucessivamente cortada por grandes reversões [98], cujas extremidades se localizam no limite da região pseudo-autossomal. Os limites sucessivos da região pseudo-autossomal no cromossomo X, da origem até posição onde ela se localiza agora, representam os limites dos "estratos evolucionários" dos cromossomos sexuais.

Diversos argumentos apontam para a existência de 5 estratos no cromossomo X [137, 147]. Os estratos são identificados na ordem em que foram criados e, assim, o estrato mais próximo da região pseudo-autossomal é identificado pelo número 5 enquanto o estrato localizado na extremidade do cromossomo X recebe o número 1.

Para estudar este problema Braga *et al.* definiram um modelo de evolução por estratos. Para uma permutação $\pi = (\pi_1, \ldots, \pi_n)$, um k-estrato é definido como uma partição de π em um conjunto ordenado $B = (I_k, I_{k-1}, \ldots, I_1)$ de k intervalos, tais que, $I_k = \{|\pi_1|, \ldots, |\pi_{n_k}|\}$, $I_{k-1} = \{|\pi_{n_k+1}|, \ldots, |\pi_{n_k+n_{k-1}}|\}$, ..., $I_1 = \{|\pi_{n_k+\ldots+n_2+1}|, \ldots, |\pi_{n_k+\ldots+n_1}|\}$, onde n_i é o tamanho do intervalo I_i.

Um sequência ótima de reversões $s = \rho_1, \rho_2, \ldots, \rho_d$ produz um k-estrato $B = (I_k, I_{k-1}, \ldots, I_1)$ em π se s possui uma subsequência $b = \theta_1, \theta_2, \ldots, \theta_k$, tal que para $1 \leq i \leq k$, a reversão θ_i contém o intervalo I_i e, para qualquer $j > i$, nenhum elemento de I_j está contido em θ_i. Além disso, para qualquer par de reversões consecutivas θ_i e θ_{i+1} em b, se ρ é uma reversão que ocorrer entre θ_i e θ_{i+1} em s, então ρ é um subconjunto de $I_1 \cup I_2 \ldots \cup I_i$. As reversões presentes em b são consideradas grandes reversões que criam estratos e as reversões que não estão em b são pequenas reversões.

Sejam uma permutação π e um k-estrato $B = (I_k, I_{k-1}, \ldots, I_1)$ para π. Se T é um *trace* de sequências ótimas de reversão de π, um B-subtrace induzido T_B é um subconjunto de T definido como $T_B = \{s | s \in T$ e s produz um k-estrato B em $\pi\}$.

A modificação feita no algoritmo de enumeração de *trace* é pequena e reside no

processo de geração e seleção de *optimal* 1-*sequences*. Após executar o algoritmo de Siepel, apenas reversões que são compatíveis com o k-estrato são selecionadas. A primeira reversão é fixa e corresponde ao primeiro estrato (é portanto uma grande reversão). Nos próximos passos, suponha que o estrato I_k foi movido por uma grande reversão e que o estrato I_{k+1} não foi. Nestas condições, é possível escolher uma grande reversão que mova o estrato I_{k+1} e nenhum elemento dos próximos estratos, ou podemos aplicar pequenas reversões existentes no conjunto $I_1 \cup \ldots \cup I_k$.

Esta restrição pode ser aplicada apenas em cromossomos lineares e é conceitualmente assimétrica, pois a análise deve ser feita do cromossomo X para o Y.

2.4.2 Comparação de genomas de grupos de espécies

A enumeração de *traces* também pode ser utilizada para a comparação de genomas de um grupo de espécies com o objetivo de se determinar padrões nos eventos de rearranjo que ocorreram durante a evolução deste grupo em relação a um ancestral comum. Para ilustrar este caso, realizamos um trabalho com um grupo de bactérias do gênero *Rickettsia*.

O gênero *Rickettsia* é um grupo de α-proteobactérias que vivem obrigatoriamente no interior de outras células e possuem genomas extremamente reduzidos, quando comparadas com genomas de bactérias similares que vivem no ambiente extra-celular. Elas costumam parasitar artrópodes e muitos membros deste gênero podem causar doenças em humanos.

Através da análise do genoma de sete membros deste gênero (*Rickettsia bellii*, *Rickettsia prowazekii*, *Rickettsia typhi*, *Rickettsia felis*, *Rickettsia massiliae*, *Rickettsia africae* e *Rickettsia conorii*), Blanc *et al*. produziram resultados interessantes sobre a evolução redutiva que as espécies deste grupo sofreram [24].

Blanc *et al*. apresentaram evidências sobre a ocorrência de reversões durante a evolução dos membros deste grupo. Além disso, eles construíram a árvore filogenética das bactérias estudadas e indicaram o número de reversões que teriam acontecido dentro de cada ramo da árvore (a partir do ancestral `R1`). A árvore filogenética construída e o número de reversões em cada ramo podem ser vistos na Figura 2.1.

A árvore filogenética mostra que a espécie *Rickettsia bellii* foi a primeira a divergir durante a evolução. O ancestral `R1` é comum aos seis membros remanescentes: *Rickettsia prowazekii* – `rpr`, *Rickettsia typhi* – `rty`, *Rickettsia felis* – `rfe`, *Rickettsia massiliae* – `rma`, *Rickettsia africae* – `raf` e *Rickettsia conorii* – `rco`.

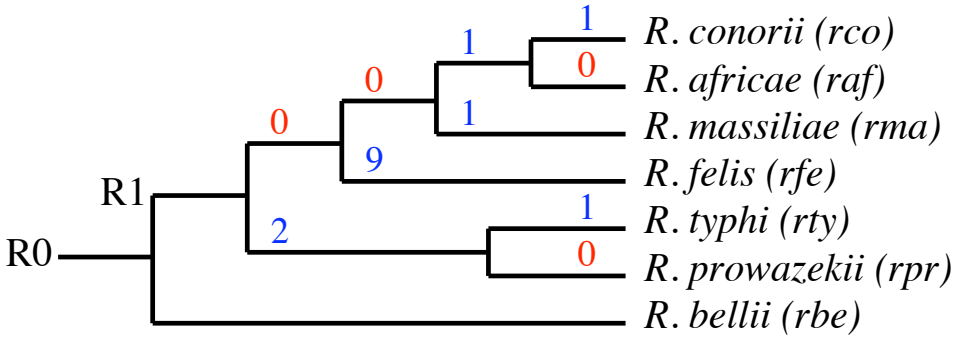


Figura 2.1: Árvore filogenética do gênero *Rickettsia* com o número de reversões que ocorreram em cada ramo, de acordo com o trabalho de Blanc *et al*. [24].

A organização do genoma (ordem dos genes) do ancestral `R1` foi reconstruída por Blanc *et al*. através da observação dos genes ortólogos presentes nas bactérias estudadas. Estes dados, que podem ser obtidos na página `RicBase` – *Rickettsia* Genome Database [136], listam os genes ou pseudo-genes presentes nos genomas dos seis descendentes que são correspondentes aos genes do genoma de `R1`. Neste conjunto de dados, os genomas das seis espécies e do ancestral `R1` estão linearizados da mesma forma.

O ancestral `R1` possui um total de 1.248 genes. Deste total, os números de genes de `R1` que possuem correspondências com genes ou pseudo-genes de `rco`, `raf`, `rma`, `rfe`, `rty` e `rpr` são, respectivamente, 1.034, 1.041, 1.088, 1.129, 816 e 826.

Para nossa análise, consideramos apenas os genes de `R1` que tinham correspondentes em todos os seis descendentes. Um total de 796 genes respeitam este critério. Para definir a direção de um gene em `R1`, adotamos a direção mais representada entre os seis genes ortólogos.

Após a definição dos genes e de suas direções em `R1`, identificamos todos os blocos de genes consecutivos que fossem comum aos seis membros do gênero *Rickettsia*. Um total de 20 blocos foram identificados e utilizados para a construção das permutações entre `R1` e seus descendentes. A Tabela 2.2 lista as permutações que foram construídas.

Calculamos as distâncias de reversão para transformar cada uma das permutações na permutação identidade (`R1`). Os valores obtidos para as α-proteobactérias `rco`, `raf`, `rma`, `rfe`, `rty` e `rpr` foram, respectivamente, 2, 1, 1, 9, 3 e 2. Este valores são idênticos aos obtidos por Blanc *et al*.

Se um algoritmo tradicional que resolve o problema de ordenação de permutações orientadas por reversões fosse utilizado para transformar as permutações da Tabela 2.2 na permutação identidade, obteríamos apenas uma solução para cada permutação. A Tabela 2.3 exibe um cenário que um destes algoritmos poderia produzir.

Tabela 2.2: Permutações entre `R1` (permutação identidade) e seus seis descendentes: `rco`, `raf`, `rma`, `rfe`, `rty` e `rpr`.

rco – Distância de reversão: 2 +1,+2,+3,+4,+5,+6,+7,+8,+9,+10,+11,+13,−12,+14,+15,+16,+17,+18,+19,+20
raf – Distância de reversão: 1 +1,+2,+3,+4,+5,+6,+7,+8,+9,+10,+11,+12,−13,+14,+15,+16,+17,+18,+19,+20
rma – Distância de reversão: 1 +1,+2,+3,+4,+5,+6,+7,+8,+9,+10,+11,+12,+13,+14,+15,+16,+17,+18,−19,+20
rfe – Distância de reversão: 9 +1,+2,+4,−3,−17,+7,−15,−16,+10,+11,+12,+13,+14,+8,−9,−6,−5,+18,+19,+20
rty – Distância de reversão: 3 +1,−5,−4,−3,−2,+6,+7,+8,+9,+10,+13,−11,−12,+14,+15,+16,+17,+18,+19,+20
rpr – Distância de reversão: 2 +1,+2,+3,+4,+5,+6,+7,+8,+9,+10,+13,−11,−12,+14,+15,+16,+17,+18,+19,+20

Neste caso, temos permutações simples e podemos verificar algumas relações entre suas sequências ótimas de reversões. Contudo, não podemos dizer que, por exemplo, o cenário apresentado pela permutação `rfe` corresponde ao que realmente ocorreu durante a evolução da espécie.

Além disso, quando observamos as sequências de reversões de `rty` e `rpr`, podemos ver que elas possuem alguma relação. Neste caso temos a reversão {2,−,5} que aparece na segunda posição da sequência de reversões de `rty` e que, na realidade, aconteceu após a reversão {12,13}, como veremos adiante.

Para obter o conjunto de todos *traces* de soluções do problema de ordenação de permutações orientadas por reversões, utilizamos o programa `analyzeTraces` [28] que implementa o algoritmo proposto por Braga *et al.* [31].

Para cada permutação, produzimos o conjunto de todos *traces* de soluções entre ela e a permutação identidade. A Tabela 2.4 lista o conjunto de *traces* produzido e exibe para cada permutação, o número de soluções representadas pelos seus *traces*.

Quando observamos a árvore filogenética exibida na Figura 2.1, podemos ver que `R1` evoluiu em dois ramos. Um ramo agrupa `rty` e `rpr` e mostra evidências que

Tabela 2.3: Exemplo de cenário evolucionário que poderia ser obtido por um algoritmo tradicional que resolva o problema de ordenação de permutações orientadas por reversões.

rco – Distância de reversão: 2 {12,13}{13}
raf – Distância de reversão: 1 {13}
rma – Distância de reversão: 1 {19}
rfe – Distância de reversão: 9 {5,−,17}{7,−,16}{8,−,14,16}{8,−,15}{8,10,−,14}{9,−,14}{3,4}{8}{4}
rty – Distância de reversão: 3 {11,13}{2,−,5}{12,13}
rpr – Distância de reversão: 2 {11,13}{12,13}

estas espécies são próximas entre si. O outro ramo mostra que, entre os 4 membros remanescentes, durante a evolução `rfe` foi a primeira a divergir, seguida então por `rma`. Finalmente, podemos ver que `rco` e `raf` são próximas entre si.

A Tabela 2.4 mostra que, exceto no caso da bactéria `rfe`, as outras 5 espécies foram afetadas por poucas reversões. Este fato pode indicar que `rfe` sofreu alguma pressão ambiental distinta que fez com que ela aceitasse mais reversões durante a sua evolução.

Os *traces* de `rfe` mostram a primeira vantagem de se enumerar o conjunto de todos os *traces* que ordenam uma permutação. Enquanto os algoritmos tradicionais produziriam um único cenário, a enumeração de *traces* permitiu a obtenção de 13 classes distintas de cenários que representam um total de 546.840 soluções ótimas diferentes. Podemos ver também que as reversões {3,4}, {4} e {5,−,17} estão presentes independentemente do cenário observado. Apenas analisando as reversões restantes, não podemos dizer qual cenário aconteceu. Contudo, estes *traces* podem fornecer informações ao pesquisadores sobre quais blocos devem ser considerados no estudo.

A forma normal de um *trace* impõe uma ordenação às reversões que estão na solução. Se temos mais que uma sub-palavra no *trace*, sabemos que as reversões que aparecem em qualquer sub-palavra u_i $(i > 1)$ sobrepõem pelo menos uma reversão da sub-palavra u_{i-1} e, por causa disso, elas só podem ser aplicadas na permutação após

Tabela 2.4: *Traces* de soluções obtidos para as *alpha*-proteobactérias `rco`, `raf`, `rma`, `rfe`, `rty` and, `rpr` com relação ao ancestral `R1`.

`rco` – Distância de reversão: 2 – # *traces*: 1 – # soluções representadas: 2 {12,13}{13}
`raf` – Distância de reversão: 1 – # *traces*: 1 – # soluções representadas: 1 {13}
`rma` – Distância de reversão: 1 – # *traces*: 1 – # soluções representadas: 1 {19}
`rfe` – Distância de reversão: 9 – # *traces*: 13 – # soluções representadas: 546.840 {3,4}{4}{5,−,17}{7}{7,10,−,16}{9}{10,−,14,16}|{7,−,9}{10,−,15} {3,4}{4}{5,−,17}{7}{8,10,−,16}|{7,8,10,−,14,16}|{7,9,15}|{8,9,15}|{10,−,15} {3,4}{4}{5,−,17}{7,−,16}{8}{8,−,14,16}{8,10,−,14}|{8,−,15}{9,−,14} {3,4}{4}{5,−,17}{7,−,16}{8}{8,10,−,16}{10,−,14,16}|{9,−,16}{10,−,15} {3,4}{4}{5,−,17}{7,−,16}{8,−,14,16}{9}{15}|{8,9,15}|{10,−,15} {3,4}{4}{5,−,17}{7,−,16}{8,10,−,14}{15}{16}|{8,15,16}|{9,−,16} {3,4}{4}{5,−,17}{7,−,16}{9}{10,−,14,16}{16}|{8,9,16}|{8,−,15} {3,4}{4}{5,−,17}{7,10,−,16}{7,15,16}{9}|{7,−,9,15,16}|{8,−,14,16}|{8,−,15} {3,4}{4}{5,−,17}{7,10,−,16}{8}{10,−,14,16}|{7,8,10,−,15}|{7,9,−,15}|{8,−,15} {3,4}{4}{5,−,17}{7,15,16}{8,10,−,14}{16}|{7,8,15}|{7,9,−,15}|{8,−,15} {3,4}{4}{5,−,17}{7,15,16}{9}{15}{16}|{7,10,−,14}|{7,−,9} {3,4}{4}{5,−,17}{8}{8,10,−,16}|{7,8,10,−,14,16}|{7,8,10,−,15}|{7,−,14}{9,−,14} {3,4}{4}{5,−,17}{8,10,−,14}|{8,15,16}{9,−,14}|{7,8,16}|{7,8,15}|{7,−,14}
`rty` – Distância de reversão: 3 – # *traces*: 1 – # soluções representadas: 3 {2,−,5}{11,13}|{12,13}
`rpr` – Distância de reversão: 2 – # *traces*: 1 – # soluções representadas: 1 {11,13}|{12,13}

todas as reversões conflitantes serem aplicadas. De fato, qualquer reversão ρ_j, que aparece em um *trace*, pode ser aplicada após uma reversão conflitante ρ_i e antes de uma reversão conflitante ρ_k, onde i (k) é o índice máximo (mínimo) de uma reversão conflitante que aparece antes (depois) de ρ_j.

Os *traces* das permutações `rty` e `rpr` mostram que estas duas espécies são próximas. De todos o membros analisados, apenas estas duas compartilham o mesmo 2-*trace* {11,13}|{12,13}. Este 2-*trace* indica que a reversão {11,13} deve acontecer antes da reversão {12,13}.

A única diferença entre os *traces* das bactérias `rty` e `rpr` é a presença da reversão {2,−,5} no *trace* de `rty`. Como esta reversão não sobrepõe as reversões {11,13} e {12,13}, ela pode ser aplicada em qualquer momento da evolução de `rty`. Contudo, quando analisamos os dois *traces* juntos, podemos inferir que a reversão {2,−,5} ocorreu após `rty` e `rpr` terem divergido de seu ancestral comum.

A mesma observação pode ser feita para as bactérias `rco` e `raf` que compartilham o 1-*trace* {13}, que não aparece em outras espécies. A única diferença entre os seus *traces* é a presença da reversão {12,13} no *trace* de `rco`. Assim, como nas espécies `rty` e `rpr`, esta reversão ocorreu após `rco` e `raf` terem divergido do seu ancestral comum.

A bactéria `rma` foi afetada apenas pela reversão {19}, que não aparece nos *traces* das outras espécies. Assim, podemos inferir que ela aconteceu após `rma` ter divergido do ancestral comum a ela, a `rco` e a `raf`.

A Figura 2.2 exibe a ordem cronológica das reversões que foi inferida a partir da observação dos *traces* das permutações.

Através da análise dos *traces* e da árvore filogenética, conseguimos reconstruir o cenário observado por Blanc *et al*. Apesar de ser um exemplo simples, este estudo mostra que *traces* podem ser utilizados para o estudo da evolução das espécies.

A enumeração de *traces* permite a verificação de correspondências entre diferentes espécies e oferece um ponto inicial para que biólogos possam realizar análises mais detalhadas.

O desenvolvimento de um algoritmo para a análise de *traces* de espécies para a detecção de cenários de evolução parcimoniosos seria uma interessante extensão deste estudo.

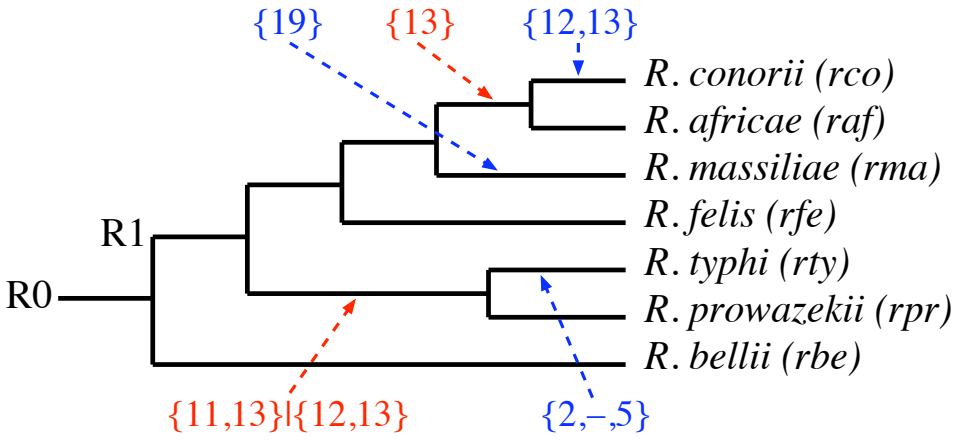


Figura 2.2: Árvore filogenética com a ordem cronológica de reversões inferida a partir da análise dos *traces* de bactérias do gênero *Rickettsia*.

2.5 Otimização do algoritmo de enumeração de *traces*

O algoritmo proposto por Braga *et al*. utiliza uma estrutura de dados especial que mantém os *k-traces* intermediários de forma ordenada. Ela realiza a transferência de dados da memória principal para o disco com o objetivo de reduzir o consumo de memória e, assim, tornar possível a enumeração do conjunto de *traces* de permutações maiores.

Devido à rotina de escrita em disco, a performance desta estrutura é extremamente prejudicada à medida que as permutações crescem e passam a exigir uma quantidade maior de memória para armazenamento dos *k-traces* intermediários.

Com o objetivo de melhorar a performance do algoritmo, desenvolvemos uma nova estrutura de dados que permite que o espaço de soluções seja explorado de maneira diferente. Os dados intermediários permanecem o tempo todo na memória principal proporcionando a diminuição do uso de memória e do tempo de execução.

O estudo produzido durante a criação da estrutura foi apresentado oralmente e publicado nos anais do 25^{th} *Symposium on Applied Computing* (`ACM SAC 2010`) sob o título "*An Improved Algorithm to Enumerate All Traces that Sort a Signed Permutation by Reversals*" [14].

A Seção 2.5.1 apresentará a estrutura utilizada originalmente e a Seção 2.5.2 apre-

sentará a nova estrutura, incluindo resultados de testes e a discussão das vantagens e desvantagens que ela proporciona.

2.5.1 A estrutura original

O algoritmo de enumeração de *traces* proposto por Braga *et al.* foi implementado em Java utilizando uma estrutura de dados especial para o armazenamento dos *traces* intermediários [27, 28]. Denominada "*Compressible sorted set*" (Conjunto ordenado compressível), esta estrutura é composta por um conjunto ordenado de subconjuntos de *traces*. Cada subconjunto mantém sua lista de *traces* ordenada em ordem crescente (de acordo com a ordem lexicográfica). Para facilitar a manipulação, cada subconjunto registra as informações sobre o menor e o maior *traces* contidos em seu interior.

Graças à organização da estrutura, um operação de inserção é feita através de uma busca binária dupla. A primeira busca localiza o subconjunto onde o novo *trace* será inserido e a segunda busca identifica a posição em que ele será colocado dentro do subconjunto para que a ordem crescente seja mantida. A Figura 2.3 exibe um esquema da estrutura.

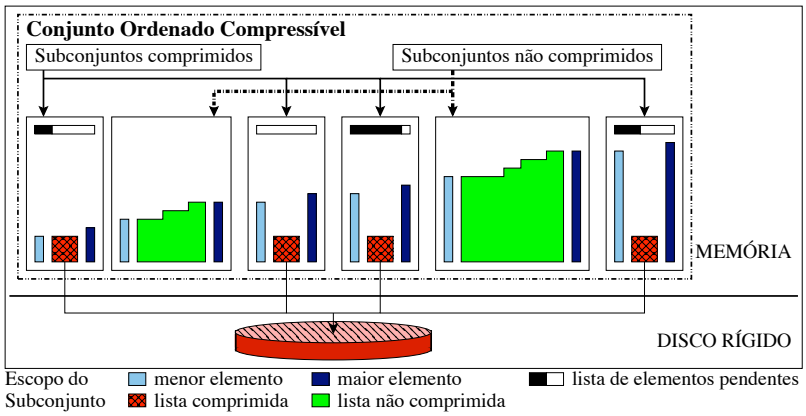


Figura 2.3: Esquema da estrutura "*Compressible sorted set*". Esta estrutura foi implementada por Braga para gerenciar os dados produzidos durante a execução do algoritmo de enumeração de *traces* [27, 28].

Os subconjuntos possuem um número máximo x de elementos. Quando este número é alcançado, o subconjunto é divido em dois novos subconjuntos segundo

um fator de balanceamento b (normalmente $b = 0.5$). Os primeiros nb elementos do subconjunto original são colocados em um novo subconjunto e os $n(1 - b)$ elementos restantes são colocados em outro subconjunto. Na implementação de Braga, o parâmetro x possui o valor padrão de 3.000.

Como podemos ver na Figura 2.3, um subconjunto pode possuir dois estados distintos: *comprimido* e *não comprimido*. Um subconjunto não comprimido mantém sua lista de elementos na memória principal. Por outro lado, quando um subconjunto está comprimido, sua lista de elementos é compactada e registrada em um arquivo no disco rígido e ele possui uma referência para este arquivo.

Para comprimir uma lista, primeiro todos os elementos são concatenados em uma matriz de bytes bidimensional. Feito isso, cada linha da matriz é comprimida e os dados comprimidos são escritos em um arquivo no disco. Finalmente, o espaço ocupado na memória principal pelos elementos recém-compactados pode ser liberado.

Como mencionado anteriormente, a inserção de um novo elemento é feita através de uma busca binária dupla. Após a primeira busca, precisamos verificar se o subconjunto está comprimido ou não comprimido. Se ele não está comprimido, precisamos apenas realizar a segunda busca binária para identificar a posição de inserção. Se o subconjunto estiver comprimido, o novo elemento é colocado em uma lista de espera.

Após a inserção de um número z de elementos na lista de espera de um subconjunto, este é descomprimido e todos os elementos da lista são inseridos nele. A lista de espera possui o papel de diminuir o custo do processo de leitura e descompactação dos dados de um arquivo. O número de subconjuntos não comprimidos é limitado por um valor y.

Para permutações que possuem um grande número de *traces*, o número de subconjuntos destinados a manter os dados intermediários pode ser muito grande. Neste caso, a descompactação de um subconjunto pode forçar a compactação de outro pois o algoritmo está trabalhando no limite dos parâmetros y e z cujos valores padrões são, respectivamente, 400 e 200.

Ao final de cada iteração i do algoritmo, uma operação de "congelamento" é executada. Ela consiste na escrita de todos os i-*traces* em disco. Na iteração seguinte $(i + 1)$, os i-*traces* são lidos sequencialmente e inseridos na estrutura que está na memória principal.

Esta estrutura funciona bem para pequenas permutações. Contudo, quando o número de elementos na permutação e/ou a distância de reversão aumentam, a performance degrada rapidamente devido ao número de acessos realizados ao disco.

Além disso, dependendo dos valores dos parâmetros x, y e z podemos encontrar dois problemas. No caso de valores altos serem escolhidos para x, y e z, o programa pode rapidamente esgotar a memória principal disponível. Por outro lado, se valores baixos são aplicados ao parâmetro x, o número de subconjuntos pode ser tão grande que o seu gerenciamento pode ser muito custoso.

Uma versão estendida da estrutura foi implementada por Braga para que o algoritmo de enumeração possa considerar regras de restrição de reversões. Nesta versão, cada *trace* armazenado é associado a uma sequência de reversões representativa. Como nem sempre a ordem de reversões imposta pela forma normal de um *trace* respeita as restrições impostas, a sequência representativa é armazenada para registrar uma solução válida. Isto garante que o algoritmo possa enumerar *traces* aplicando restrições simétricas ou assimétricas.

2.5.2 A estrutura de pilha

Quando um *trace* $s = u_1|\ldots|u_m$ está em sua forma normal, ele respeita um conjunto de regras que foram apresentadas na Seção 2.1.3. A organização imposta por estas regras permite que um conjunto de *traces* possa ser representado através de uma estrutura de árvore ordenada similar àquela exibida na Figura 2.4, que exibe os *traces* que ordenam a permutação $\pi = (-3, +2, +1, -4)$.

Cada nó da árvore representa um conjunto de reversões ordenadas em ordem lexicográfica. O nó raiz contém a lista de *optimal* 1-*sequences* da permutação origem. Associada a cada reversão ρ de um nó que não é folha, temos uma sub-árvore que agrupa reversões que são lexicograficamente maiores do que ρ ou que devem ser aplicadas após ρ. Na Figura 2.4, o nó A contém todas as *optimal* 1-*sequences* da permutação π. A reversão $\{1\}$ do nó A contém uma sub-árvore, cuja raiz é o nó B. O nó B, por sua vez, contém todas as *optimal* 1-*sequences* da permutação π' que é obtida após a aplicação da reversão $\{1\}$ em π.

Todo caminho da raiz da árvore a um nó no nível i produz um i-*trace*. Assim, um caminho da raiz até um nó no nível $i = d(\pi)$ corresponde a um *trace* que ordena a permutação π.

Para melhorar o consumo de memória do algoritmo de enumeração de *traces*, podemos utilizar o fato de que não necessitamos calcular todos os i-*traces* antes de calcular o primeiro $(i+1)$-*trace*. Isso significa que ao invés de explorarmos a árvore em largura, podemos explorá-la em profundidade.

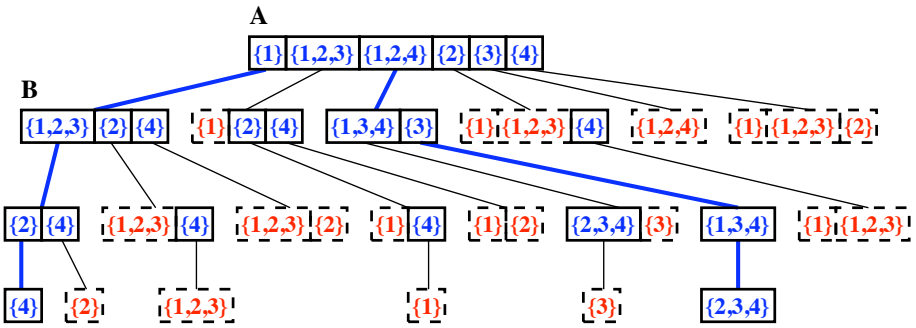


Figura 2.4: Estrutura em forma de árvore que contém os *traces* que ordenam a permutação $\pi = (-3, +2, +1, -4)$. Nesta representação, apenas os valores que estão dentro de caixas com bordas sólidas são reversões que efetivamente estão na estrutura. Os valores que estão dentro de caixas com bordas tracejadas são reversões que são *optimal* 1-*sequences* mas que, quando adicionadas ao i-*trace* pai, formam *traces* que já foram inseridos em outros ramos da árvore. Os dois caminhos com arestas mais largas indicam os *traces* $\{1\}\{1,2,3\}\{2\}\{4\}$ e $\{1,2,4\}\{3\}|\{1,3,4\}|\{2,3,4\}$ que ordenam a permutação.

Quando exploramos a árvore em profundidade, estamos olhando apenas para um ramo da árvore. Portanto, ao invés de utilizar a representação em forma de árvore, uma estrutura de pilha pode ser utilizada para armazenar o ramo que está sendo explorado.

O nível inferior da pilha é iniciado com a lista de *optimal* 1-*sequences* calculadas para a permutação origem π. Para criar cada nível i $(1 < i \leq d(\pi))$, primeiro construímos um i-*trace* T, percorrendo a pilha de baixo para cima (isto é, do nível inferior ao topo), adicionando em T a primeira (menor) reversão presente em cada nível. Feito isso, aplicamos T à permutação π para obter uma nova permutação $\pi' = \pi \circ T$. A partir de π', obtemos a sua lista L de *optimal* 1-*sequences*. Para cada reversão ρ presente em L, verificamos se ao adicioná-la em T, ela aparecerá na última posição do novo $(i+1)$-*trace* T'. Em caso afirmativo, adicionamos ρ ao novo topo da pilha. Caso contrário, ρ pode ser descartada.

A Figura 2.5 mostra um esquema da estrutura de pilha no momento em que o algoritmo está enumerando os *traces* da permutação $\pi = (-3, +2, +1, -4)$, explorando o ramo que possui raiz na reversão $\{1, 2, 3\}$. Neste exemplo, a pilha possui dois níveis e {1,2,3}{2} é o 2-*trace* corrente. Ao aplicamos este 2-*trace* em π, obtemos a permutação $\pi' = (-1, +2, +3, -4)$ que possui duas *optimal* 1-*sequences*:

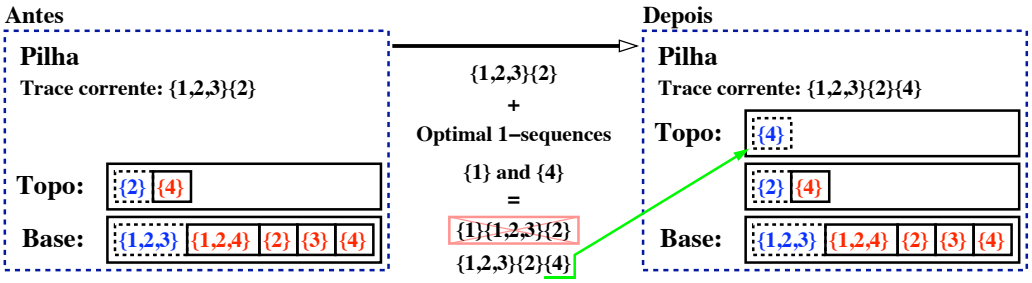


Figura 2.5: Esquema da estrutura de pilha durante o processamento da permutação $(-3, 2, 1, -4)$. Quando combinada com o 2-*trace* $\{1, 2, 3\}\{2\}$, apenas a reversão $\{4\}$ aparece na última posição quando adicionada ao 2-*trace* corrente e, por causa disso, apenas ela é inserida no novo topo da pilha.

$\{1\}$ e $\{4\}$. Quando adicionamos $\{1\}$ e $\{4\}$ ao 2-*trace* corrente, obtemos os 3-*traces* $a = \{1\}\{1, 2, 3\}\{2\}$ e $b = \{1, 2, 3\}\{2\}\{4\}$. Como a reversão $\{1\}$ não se localiza na última posição de a, ela é descartada. Por outro lado, a reversão $\{4\}$ aparece na última posição de b e, por isso, ela é inserida no novo topo da pilha.

Quando o nível $i = d(\pi)$ é atingido, temos um *trace* que ordena a permutação π. Neste caso, o *trace* pode ser impresso e o topo da pilha é removido. Além disso, removemos também a primeira reversão do novo topo (nível $d(\pi) - 1$) e começamos a explorar o próximo ramo. Toda vez que o topo fica vazio, ele deve ser removido em conjunto com a primeira reversão do nível inferior.

O Algoritmo 2.5.1 descreve os passos necessários para se enumerar todos os *traces* de uma permutação π utilizando-se uma pilha como estrutura de dados. Este algoritmo baseia-se na possibilidade de que podemos descartar reversões que, quando inseridas no i-*trace* corrente, não aparecem na última posição do $(i+1)$-*trace* criado. Precisamos provar, portanto, que este procedimento é seguro e que não estamos perdendo dados durante o processo de enumeração. Para isso, devemos analisar o que acontece quando adicionamos uma nova reversão ρ em um i-*trace* T para construir um novo $(i+1)$-*trace* T'.

Seja u_m a última sub-palavra de um i-*trace* T que possui altura igual a m. A sub-palavra u_m possui ao menos uma reversão. Seja θ_i a última reversão de u_m. Quando adicionamos uma nova reversão ρ em T, uma das seguintes situações podem ocorrer:

1. A reversão ρ sobrepõe ao menos uma reversão da sub-palavra u_m. Neste caso,

Algoritmo 2.5.1: Algoritmo de enumeração de *traces* utilizando pilha

Entrada: Permutação origem π
Saída: Lista de *traces* que ordenam π.

```
1  início
2      Crie uma pilha vazia P;
3      L ← lista ordenada contendo as optimal 1-sequences de π;
4      Insira L em P;
5      enquanto Altura(P) > 0 faça
6          Construa um novo i-trace T com a primeira reversão de cada nível da
           pilha caminhando da base em direção ao topo;
7          se Altura(P) = d(π) então
8              Imprima T;
9              Remova a primeira reversão do topo da pilha;
10         senão
11             π' ← π ∘ T;
12             L ← lista ordenada contendo as optimal 1-sequences de π';
13             L' ← ∅;
14             para cada ρ ∈ L faça
15                 T' ← trace resultante da adição de ρ em T;
16                 se ρ está na última posição de T' então
17                     L' ← L' ∪ ρ;
18                 fim
19             fim
20             Insira L' (ordenado) P;
21             enquanto Topo de P estiver vazio e Altura(P) > 0 faça
22                 Remova o topo de P;
23                 Remova a primeira reversão do novo topo da pilha;
24             fim
25         fim
26     fim
27 fim
```

uma nova sub-palavra u_{m+1} é criada com a reversão ρ e ela é concatenada ao final de T.

2. A reversão ρ não sobrepõe nenhuma reversão em u_m ($m \geq 1$) e ela sobrepõe uma reversão na sub-palavra u_{m-1} ($m > 1$). Neste caso, ρ deve ser inserida em u_m e temos duas possibilidades:

   (a) A reversão ρ é lexicograficamente maior do que θ_i. Neste caso, inserimos ρ imediatamente após θ_i.

   (b) A reversão ρ é lexicograficamente menor do que θ_i. Neste caso, ρ é inserida em alguma posição antes da reversão θ_i.

3. A reversão ρ não sobrepõem nenhuma reversão em u_m e em u_{m-1}. Neste caso, ρ será inserida em alguma sub-palavra u_i tal que $1 \leq i < m$.

Temos, portanto, dois casos onde ρ se torna a última reversão do novo $(i+1)$-*trace* T': 1 e $2(a)$. Nestes casos, é fácil ver que obtemos um novo $(i+1)$-*trace* que descende diretamente do i-*trace* original (T é prefixo de T').

Nos outros dois casos, $2(b)$ e 3, a reversão ρ é inserida em alguma posição diferente da última posição. Sem perda de generalidade, podemos definir as sequências de reversões $S = \theta_1 \dots \theta_k \theta_{k+1} \dots \theta_i$ e $S' = \theta_1 \dots \theta_k \rho \theta_{k+1} \dots \theta_i$, respectivamente, como as sequências equivalentes aos *traces* T e T'.

Podemos ver que S e S' possuem um prefixo comum $\mathcal{P} = \theta_1 \dots \theta_k$. Se ρ e θ_{k+1} podem ser inseridas imediatamente após o prefixo comum $\mathcal{P}$, isso significa que estas reversões são *optimal* 1-*sequences* da permutação π', que é criada através da aplicação de cada reversão de $\mathcal{P}$ em π.

A lista de *optimal* 1-*sequences* de π' é processada em ordem lexicográfica crescente. Logo, se ρ é menor do que θ_{k+1}, temos que todos os *traces* que possuem prefixo $\theta_1 \dots \theta_k \rho$ já foram explorados (lembrem-se, que no momento, estamos explorando o ramo do prefixo $\theta_1 \dots \theta_k \theta_{k+1}$). Se ρ é maior do que θ_{k+1}, o ramo do prefixo $\theta_1 \dots \theta_k \rho$ será explorado no futuro.

Observando S', vemos que a reversão θ_{k+1} aparece depois da reversão ρ, o que indica que θ_{k+1} é uma *optimal* 1-*sequence* da permutação que obtemos quando aplicamos a sequência de reversões $\theta_1 \dots \theta_k \rho$ em π. Independentemente de ser explorado antes ou depois, o ramo $\theta_1 \dots \theta_k \rho$ será capaz de gerar T'. Isso nos garante que não estamos perdendo informações quando não incluímos ρ no topo da pilha e que o algoritmo é seguro.

2.5.3 Comparação de performance

A estrutura original (`CS`) utilizada por Braga e apresentada na Seção 2.5.1 e a estrutura de pilha (`ST`) proposta na Seção 2.5.2 foram avaliadas através da realização de testes para medição de tempo e consumo de memória.

Utilizando como base o código fonte original implementado em Java por Braga, implementamos nossa estrutura de modo que ela utilizasse o mesmo conjunto de objetos. Os testes foram realizados em um Intel Pentium 4 HT 3.0 GHz com 2.0 GB de RAM e sistema operacional Ubuntu. Para evitar a influência de operações de *swap* na performance dos programas, limitamos a memória máxima da máquina virtual Java em 1.0GB (parâmetro `-Xmx1024m`). A estrutura `CS` foi testada com os seus parâmetros padrões.

Permutações orientadas aleatórias de n elementos foram criadas ($5 \leq n \leq 14$). Para cada valor de n, criamos permutações com distâncias de reversão $d_1 = \lceil (n + 1)/2 \rceil$, $d_2 = \lceil 3n/4 \rceil$ e, $d_3 = n$. Um total de 100 permutações aleatórias foram criadas para cada par (n, d). As permutações criadas possuem $n+1$ *breakpoints* e, portanto, não possuem adjacências. Isso garante que elas não podem ser transformadas em permutações com menor número de elementos.

Como o pacote desenvolvido por Braga não trabalha com permutações que possuem obstáculos, as permutações foram construídas respeitando este critério. A decisão de ignorar obstáculos baseia-se no fato de que a probabilidade de encontrá-los em permutações aleatórias é muito pequena [152].

Cada conjunto de permutações representado um par (n, d) foi processado com as duas diferentes estruturas. Registramos o tempo total de execução e valor máximo de memória utilizada durante o processamento de cada permutação. A memória foi mensurada com a utilização de uma *thread* separada que, em tempos regulares, coletava a memória utilizada pela máquina virtual Java (Objeto `Runtime`: métodos `totalMemory()` and `freeMemory()`).

A Figura 2.6(a) mostra o número médio de *traces* observados em cada um dos conjuntos de 100 permutações utilizados nos testes. Para um dado valor de n o número de *traces* que ordenam a permutação cresce exponencialmente à medida que a distância de reversão d se aproxima do valor de n.

Para cada par (n, d), as Figuras 2.6(b) e 2.6(c) exibem, respectivamente, a média da memória máxima utilizada pelas estruturas e a média do tempo total de execução para processar cada permutação.

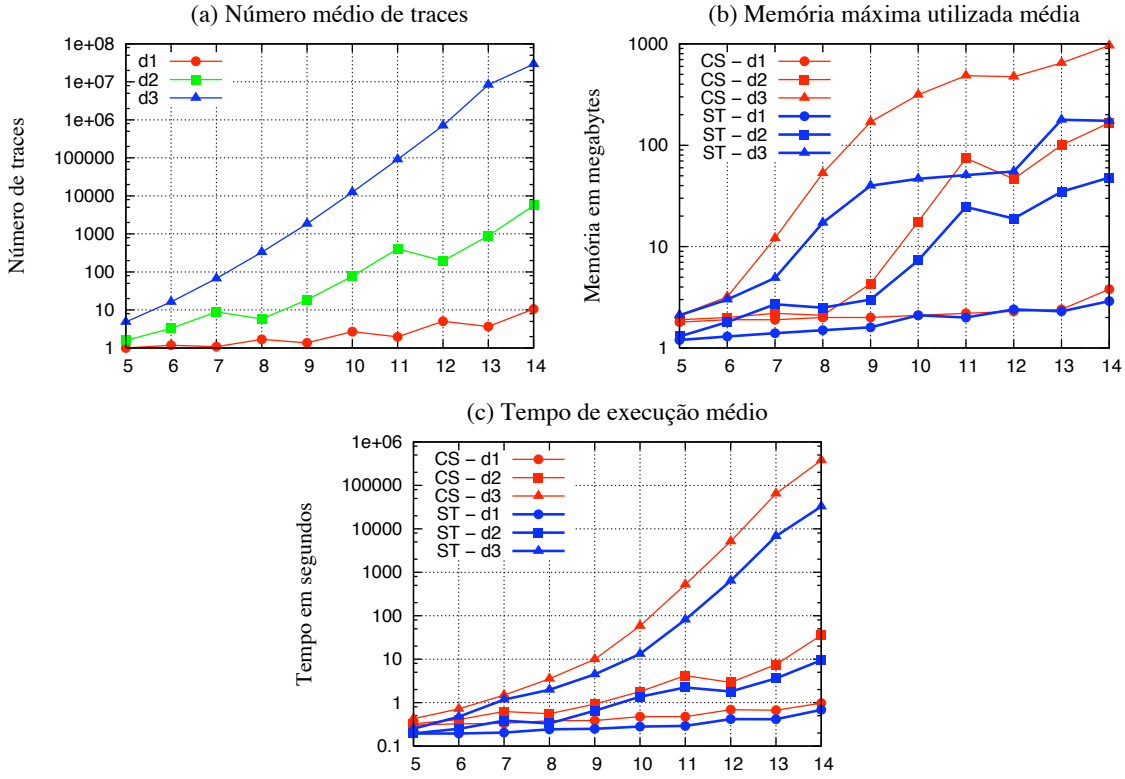


Figura 2.6: As estruturas `CS` e `ST` foram testadas com 100 permutações aleatórias para cada par (n, d): $d_1 = \lceil (n+1)/2 \rceil$, $d_2 = \lceil 3n/4 \rceil$ e, $d_3 = n$. Os gráficos exibem as médias dos números de *traces*, tempos de execução e memória utilizada observados em cada conjunto.

Permutações com distância de reversão d_1 são menos complexas e, por isso, são mais fáceis de se ordenar do que permutações que possuem o mesmo número de elementos n e valores de distância maiores.

Em nossos testes, apesar do aumento do valor de n, o tempo médio necessário para enumeração dos *traces* das permutações com distância de reversão d_1 é menor do que 1 segundo. Este tempo médio aumenta para alguns segundos no caso das permutações com distância d_2 e para algumas horas no caso das permutações com distância d_3. A mesma observação pode ser feita sobre o consumo de memória que cresce com a complexidade das permutações.

As Figuras 2.6(b) e 2.6(c) mostram claramente que a estrutura `ST` apresenta

performance melhor do que a estrutura `CS` tanto em consumo de memória como em tempo de execução. Na maioria dos casos, ela apresentou os menores valores para esses dois parâmetros e estes resultados confirmam a vantagem de se manter apenas o ramo que está sendo explorado na memória principal.

Através da redução do número de objetos que a estrutura tem que gerenciar, eliminamos a penalização causada pela escrita e leitura de dados no disco rígido. Por exemplo, quando processamos o conjunto de permutações $(14, 14)$, a estrutura `CS` trabalha perto do limite de memória mesmo utilizando o recurso de compactação e congelamento dos dados não utilizados. Para o mesmo conjunto de permutações, a estrutura `ST` usa 5,53 vezes menos memória e é 11,54 vezes mais rápida do que a estrutura `CS`.

No estudo de cenários evolucionários, frequentemente estudamos espécies que são próximas. Normalmente, esperamos que as permutações entre elas sejam mais simples do que aquelas que possuem distância de reversão d_2 e d_3. Assim, decidimos avaliar o consumo de memória e o tempo de execução apresentado pelas estruturas quando elas são usadas para processar permutações menos complexas. Para isso, criamos conjuntos de 100 permutações aleatórias com n elementos $(5 \leq n \leq 35)$ e $d = d_1$.

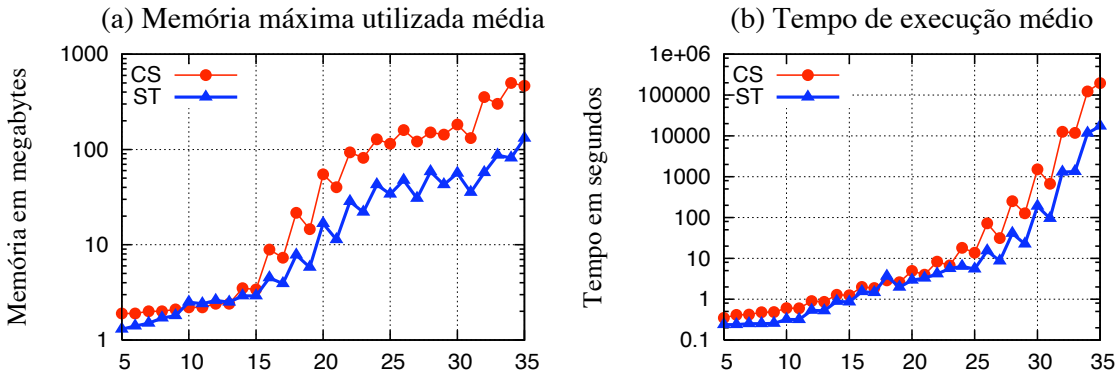


Figura 2.7: Memória utilizada e tempo de execução médios das estruturas `CS` e `ST` durante o processamento de permutações de n elementos $(5 \leq n \leq 35)$ com distância de reversão $d = \lceil (n+1)/2 \rceil$.

As Figuras 2.7(a) e 2.7(b) mostram a evolução do consumo de memória e do tempo de execução de cada estrutura durante o processamento de permutações de n

elementos e distância de reversão d_1. As curvas indicam que a estrutura `ST` apresenta aumento dos valores destes dois parâmetros em uma taxa menor do que a da estrutura `CS`.

Para avaliar a evolução do consumo de memória ao longo do tempo, decidimos avaliar as estruturas utilizando uma permutação mais complexa construída com dados reais. Para isso, escolhemos duas bactérias do gênero *Wolbachia*: *Wolbachia endosymbiont of Culex quinquefasciatus* e *Wolbachia endosymbiont of Drosophila melanogaster*. A escolha deste gênero justifica-se pelo alto grau de rearranjo dos genes que as suas bactérias apresentam [96].

Para construir a permutação entres as duas bactérias, realizamos um `BLAST` recíproco entre as sequências protéicas dos genes. Apenas os *hits* recíprocos que possuiam *e-value* menor o que 10^{-6} e cobertura de pelo menos 50% foram considerados. Baseado na ordem dos genes selecionados no genoma das duas bactérias, construímos uma permutação de 252 elementos e distância de reversão 245.

Uma permutação com $n = 252$ e $d = 245$ exige um tempo muito grande para ser processada. Assim, decidimos avaliar o consumo de memória das estruturas ao longo de 48 horas, anotando os valores de memória utilizada de 15 em 15 segundos. A Figura 2.8 exibe as curvas dos valores obtidos para cada estrutura.

Observando as curvas das estruturas `CS` e `ST` na Figura 2.8, podemos ver um padrão interessante. O consumo de memória aumenta e diminui em ondas. Para a estrutura `CS`, este padrão é explicado pelas rotinas de compactação e descompactação (ondas menores) e congelamento de dados (onda maior ao final das 48 horas). Para a estrutura `ST`, este movimento está relacionado à inclusão e remoção de listas de reversões na pilha enquanto o algoritmo explora os ramos da árvore de *traces*. O gráfico reforça a observação que a estrutura `ST` consome menos memória do que a estrutura `ST`.

Como a permutação é extremamente complexa, após 48 horas de execução, nenhum dos programas produziu algum *trace*. Contudo, mesmo assim podemos ver vantagem da estrutura `ST` em relação à estrutura `CS`. Como a estrutura de pilha está processando a árvore de *traces* em profundidade, assim que ela chegar ao nível $i = d(\pi)$ ela escreverá uma solução. A estrutura `CS`, por sua vez, depende da exploração de todos os níveis antes de escrever o primeiro resultado do processamento. No nosso teste, após 48 horas o programa que usa `CS` iniciava o processamento do terceiro nível da árvore de *traces* (a grande redução de memória ao final das 48 horas refere-se ao congelamento de todos os 2-*traces* calculados).

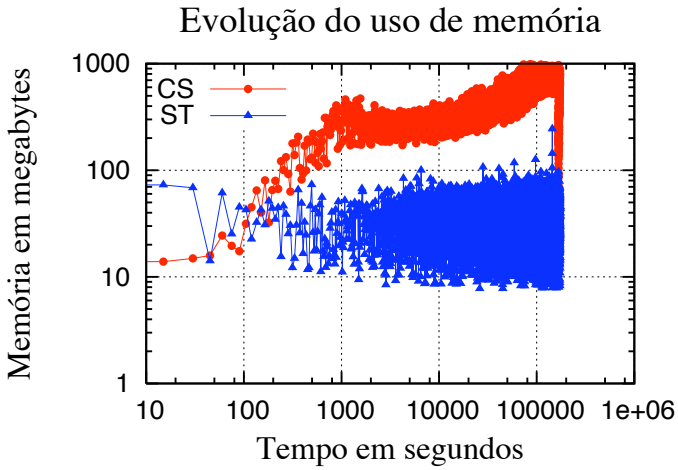


Figura 2.8: Evolução do uso de memória pelas estruturas `CS` e `ST` ao longo de 48 horas durante o processamento da permutação entre as bactérias *Wolbachia endosymbiont of Culex quinquefasciatus* e *Wolbachia endosymbiont of Drosophila melanogaster*. A permutação possui $n = 252$ e $d = 245$. Os valores de memória foram coletados de 15 em 15 segundos.

2.5.4 Conclusão

Nossos testes mostraram que a estrutura de pilha possui performance melhor do que a exibida pela estrutura original.

Através da manipulação destes parâmetros da estrutura `CS`, poderíamos obter uma estrutura que conserva a memória principal e realiza mais acessos a disco ou uma estrutura que reduz o número de acessos a disco e consome mais memória principal. Devido à necessidade de se realizar acessos a discos, a estrutura `CS` não seria capaz de superar a performance da estrutura `ST` e, por isso optamos por testá-la somente com os parâmetros padrões definidos na implementação de Braga [27, 28].

A melhora no tempo de execução apresentada pela pilha se explica pela eliminação da necessidade de se acessar o disco rígido para registro e leitura de dados intermediários. A redução do consumo de memória ocorre porque, se imaginamos o conjunto de *traces* como uma árvore, a estrutura `ST` mantém apenas um ramo na memória enquanto a estrutura `CS` gerencia um nível completo.

Como um ramo da árvore é independente dos outros ramos, o algoritmo de enumeração de *traces* que utiliza a pilha pode ser estendido para um algoritmo para-

lelo/distribuído onde cada processo/nó trabalha com um ramo diferente.

Outra vantagem apresentada pela pilha é que, através de um único *trace*, podemos reconstruí-la. No caso da execução do programa ser interrompida, podemos pegar o último *trace* produzido, reconstruir a pilha e continuar com o processamento.

Uma desvantagem apresentada pela estrutura de pilha é a eliminação da possibilidade de se calcular o número de soluções que cada *trace* representa. Como estamos explorando ramos e reversões são eliminadas durante o processo, não podemos efetuar a etapa de soma dos tamanhos dos *traces* para acumular o número total de soluções.

A estrutura de pilha possui outra desvantagem relacionada ao uso de restrições para a eliminação de reversões. Como o algoritmo considera apenas i-*traces* que são prefixos de $(i+1)$-*traces*, ele não pode manter uma solução representativa como no caso da estrutura `CS`. Para construir uma solução representativa, a pilha não poderia descartar reversões que ao serem inseridas em um i-*trace* não aparecem na última posição. Nesse caso, o algoritmo exploraria uma grande porcentagem do espaço de soluções, ao invés de explorar o espaço de *traces*, aumentando dessa maneira o tempo de execução.

Por este motivo, a estrutura `ST` é capaz apenas de trabalhar com restrições que geram *traces* perfeitos como, por exemplo, no caso da estratégia de detecção inicial de intervalos comuns entre as permutações origem e alvo. Restrições como a detecção progressiva de intervalos comuns, simetria ao redor do término de replicação e estratificação de permutações não são aplicáveis com o uso desta estrutura.

A estrutura de pilha gerou uma melhora significativa no que se refere ao uso de memória. O tempo de execução foi reduzido mas ainda é muito alto porque o número de *traces* também cresce exponencialmente com a complexidade das permutações.

2.5.5 Trabalhos futuros e trabalhos relacionados

Uma extensão deste trabalho seria a realização de um estudo para se descobrir padrões que permitam a eliminação de ramos que não geram *traces*. Outra possibilidade seria o desenvolvimento de outra forma de representação de soluções ainda mais compacta.

Recentemente, em Setembro de 2010, Swenson, Badr e Sankoff propuseram um algoritmo que produz a lista de todas as *optimal* 1-*sequences* de uma permutação em tempo quadrático [151]. Outro trabalho futuro seria a substituição do algoritmo de

Siepel por este novo algoritmo para diminuição da complexidade total dos algoritmos que utilizam tanto a estrutura `CS` como a estrutura `ST`.

Em Outubro de 2010, Badr, Swenson e Sankoff publicaram um estudo que realiza uma análise de nosso algoritmo e propõem uma nova adaptação para o algoritmo de Braga *et al.* [30] e para o nosso algoritmo que utiliza estrutura de pilha [7].

A análise de complexidade realizada por eles mostrou que o tempo necessário para que o algoritmo de enumeração de *traces* que utiliza estrutura de pilha produza um único *trace* é $O(n^4 2^n)$. Assim, se N é o número total de *traces*, a complexidade para se gerar todos os *traces* de soluções de uma permutação de tamanho n é $O(Nn^4 2^n)$.

A complexidade do algoritmo de Braga *et al.* é $O(Nn^{k_{max}+4})$, onde k_{max} é a largura máxima de um *d-trace*. O valor da largura de um *trace* é no mínimo igual ao tamanho máximo de uma sub-palavra u_i na forma normal de um *trace*. Por exemplo, em um *trace* de altura 2, a largura de uma sub-palavra é pelo menos $\lceil n/2 \rceil$. Neste caso, o algoritmo teria complexidade $O(Nn^4 n^{n/2})$.

A adaptação proposta por Badr, Swenson e Sankoff consiste na realização do agrupamento dos *i-traces* em função das permutações que eles produzem, partindo do nível 0 (permutação origem) até o nível i. Como vários *i-traces* podem chegar a uma mesma permutação, o algoritmo proposto pelos autores economiza um grande número de operações ao realizar a geração da lista de *optimal* 1-*sequences* uma vez para cada permutação do nível i. Por exemplo, no caso do nosso algoritmo, a lista de *optimal* 1-*sequences* é gerada para cada *i-trace* do nível i.

Em seus testes que utilizaram permutações com n variando entre 5 e 26 e distâncias de permutações $d_1 = \lceil (n+1)/2 \rceil$, $d_2 = \lceil 3n/4 \rceil$ e, $d_3 = n$, as comparações mostraram que entre o algoritmo de Braga *et al.* e sua versão adaptada o ganho de tempo foi de até 70% e entre o nosso algoritmo e a sua versão adaptada o ganho foi de até 50%. Dessa maneira, a proposta de Badr, Swenson e Sankoff se mostra muito interessante e indica pontos que podem ser atacados para melhora da performance do algoritmo.

Uma ressalva que precisa ser feita sobre o estudo apresentado por eles é em relação ao consumo de memória que não é mencionado. Durante o processamento, as permutações do nível i corrente são armazenadas como chaves de uma tabela *hash* enquanto os valores das tabelas são os grupos de *i-traces*. Como todos estes valores são mantidos em memória, permutações com tamanhos ou distâncias de reversão grandes podem exigir uma imensa quantia de memória principal.

Capítulo 3

Enumeração parcial de *traces*

O algoritmo de enumeração de *traces* apresentado no Capítulo 2 (Seção 2.3) permite a listagem de todos o *traces* de soluções de permutações. Contudo, sua utilidade prática se limita a permutações mais simples, com valores pequenos de distância de reversão ($d(\pi) \leq 12$). Permutações mais complexas normalmente geram um número imenso de *traces* e, além disso, necessitam de uma grande quantidade de tempo para serem processadas.

Uma maneira de produzir uma enumeração mais "racional" dos *traces* é a adoção de restrições biológicas para que reversões sejam escolhidas de maneira mais seletiva e que o espaço de *traces* de soluções seja reduzido. Além de diminuir o tamanho do conjunto de soluções, esta estratégia produz *traces* mais relevantes para a análise biológica.

Com exceção da restrição de detecção de intervalos comuns (não progressiva), as restrições apresentadas na Seção 2.4.1 não podem ser utilizadas com o algoritmo de enumeração que utiliza a estrutura de pilha (Seção 2.5.2). Assim, o algoritmo original, que utiliza a estrutura *Conjunto ordenado compressível* (Seção 2.5.1), deve ser utilizado.

O algoritmo original percorre a árvore de *traces* em largura e ele imprime o resultado do processamento apenas após processar o último nível. Essa característica faz com que a sua utilização seja impraticável, mesmo com o uso de restrições, para o processamento de permutações com grande número de elementos e/ou grande distância de reversão.

Diante deste problema, uma abordagem alternativa seria a enumeração *parcial* do conjunto de *traces*. Esta estratégia permite que um conjunto de *traces* seja

obtido e utilizado por biólogos para a análise de cenários evolutivos. Neste capítulo apresentaremos algoritmos para a enumeração parcial de *traces* e discutiremos as vantagens e desvantagens desta abordagem.

3.1 Algoritmos de enumeração parcial de *traces*

Os algoritmos de enumeração parcial são uma alternativa para o processamento de grandes permutações pois podem produzir um conjunto de *traces* em um tempo muito menor do que o necessário para se enumerar todos os *traces*.

Para a realização de uma enumeração parcial, o algoritmo deve ter um critério de parada específico. Esse critério pode ser o tempo de execução, o número de *traces* produzidos, etc.

No caso deste estudo, utilizaremos o limite de tempo como critério de parada. Assim, o objetivo é avaliar o desempenho dos algoritmos dentro de um limite fixo de tempo.

3.1.1 Algoritmo aleatório

Um solução muito simples para a realização da enumeração parcial de *traces* é através da adoção de uma estratégia que constrói os *traces* de maneira aleatória.

Sejam π_0 a permutação origem, π_d a permutação alvo, onde d é a distância de reversão entre π_0 e π_d. Esta estratégia consiste em gerar um *trace* que representa uma sequência de reversões construída passo a passo através da seleção ao acaso de uma reversão do conjunto de *optimal* 1-*sequences* de cada permutação π_i existente entre π_0 e π_d ($0 \leq i < d$). O Algoritmo 3.1.1 lista os passos necessários para a produção de um *trace* aleatório.

Sejam t_{max} o tempo máximo de execução e t_{ac} o tempo acumulado desde o início da execução, o Algoritmo 3.1.2 gera um conjunto de *traces* aleatórios dentro de um determinado intervalo de tempo.

3.1.2 Algoritmo tradicional limitado por tempo

Outra opção simples para a geração parcial de *traces* é a utilização do algoritmo de enumeração que utiliza a estrutura de pilha (Algoritmo 2.5.1) com uma pequena

Algoritmo 3.1.1: Algoritmo para geração de um *trace* aleatório

Entrada: π_0 e π_d
Saída: *Trace* aleatório que transforma π_0 em π_d

início
  $T_0 \leftarrow$ *trace* vazio;
  para $i \leftarrow 0$ **até** $d-1$ **faça**
    $\rho \leftarrow$ reversão aleatória da lista de *optimal* 1-*sequences* de π_i;
    $T_{i+1} \leftarrow T_i + \rho$;
    $\pi_{i+1} \leftarrow \pi_i \circ \rho$;
  fim
  Retorne T_d;
fim

Algoritmo 3.1.2: Algoritmo para geração de um conjunto de *traces* aleatórios

Entrada: π_0, π_d e t_{max}
Saída: Conjunto de *traces* aleatórios que transformam π_0 em π_d

início
  $\mathcal{T} \leftarrow \emptyset$;
  $t_{ac} \leftarrow 0$;
  enquanto $t_{ac} < t_{max}$ **faça**
    $t_{inicio} \leftarrow$ tempo corrente;
    $T \leftarrow$ *trace* aleatório produzido pelo Algortimo 3.1.1 que transforme π_0 em π_d;
    $\mathcal{T} \leftarrow \mathcal{T} \cup T$;
    $t_{fim} \leftarrow$ tempo corrente;
    $t_{ac} \leftarrow t_{ac} + (t_{fim} - t_{inicio})$;
  fim
  Retorne $\mathcal{T}$;
fim

modificação para a introdução de uma verificação de tempo decorrido. Com esta alteração, a execução pode ser interrompida quando o tempo limite é alcançado.

3.1.3 Algoritmo de janela deslizante

Durante a geração de *traces* aleatórios pelo Algoritmo 3.1.2, toda vez que o Algoritmo 3.1.1 é chamado, estamos apenas explorando um único ramo da árvore de *traces* do início (raiz) até o final (folha).

O algoritmo de enumeração de *traces* utilizando pilha (Algoritmo 2.5.1) tem uma abordagem diferente pois, a cada vez que ele sobe para um nível $i+1$ da árvore, ele explora toda a sub-árvore do i-*trace* corrente. Assim, ele explora diversos ramos da árvore antes de passar para o próximo i-*trace* no nível i.

Quando exploramos a árvore de *traces* em profundidade, um aspecto importante é que quanto maior o nível i da árvore maior será o número de "*ramos mortos*". Um ramo é considerado morto quando ele não é estendido, ou seja, quando uma reversão é adicionada a um i-*trace* e é descartada porque ela não aparece na última posição do $(i+1)$-*trace* gerado. De fato, para grandes permutações a maior parte do tempo de processamento é gasto na exploração de ramos mortos. Na Figura 3.1, os ramos mortos são aqueles que terminam em reversões marcadas por caixas tracejadas.

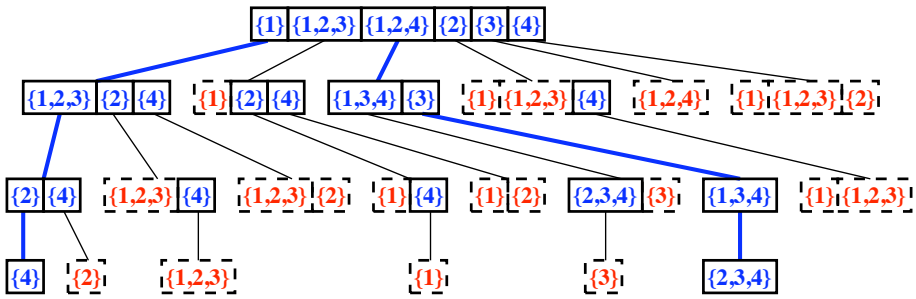


Figura 3.1: Estrutura em forma de árvore que contém os *traces* que ordenam a permutação $\pi = (-3, 2, 1, -4)$. As reversões que estão em caixas tracejadas indicam a ocorrência de um ramo morto. Um ramo morto é uma ramo que não é estendido porque a reversão adicionada ao i-*trace* não aparece na última posição do $(i+1)$-*trace* gerado.

Sejam π_0 a permutação origem, π_d a permutação alvo e π_k uma permutação intermediária dada por uma sequência ótima de reversões após a aplicação das k

primeiras reversões ($1 \leq k < d(\pi)$). Neste contexto, podemos definir o *k-trace* A e o *l-trace* B, onde $l = d(\pi) - k$, como os *traces* que, respectivamente, representam conjuntos de soluções que transformam π_0 em π_k e π_k em π_d.

Se pegarmos as reversões de B e adicioná-las uma a uma em A, obteremos um novo *trace* C que converte π_0 em π_d. Este *trace* representará um conjunto de soluções que utilizam as reversões listadas em A e B, respeitando as relações de sobreposições existente entre elas. Note que este conjunto poderá ser mais amplo do que aquele formado pela simples concatenação de cada sequência de soluções de B ao final de cada sequências de soluções de A.

Suponha que A represente um conjunto de x soluções que transformam π_0 em π_k e que B represente um conjunto de y soluções que transformam π_k em π_d. A simples concatenação das soluções de A e B geraria um conjunto de $x \times y$ soluções que convertem π_0 em π_d e passam por π_k. Contudo, no conjunto de soluções representadas por C, reversões pertencentes a B podem aparecer em posições inferiores a k e, quando isso acontece, π_k não é uma permutação intermediária.

Esta estratégia de combinar *k-traces* menores para construir um *trace* maior pode ser utilizada para a criação de um algoritmo de janela deslizante. Neste algoritmo, um conjunto de permutações intermediárias produzido por uma sequência ótima de reversões é utilizado como um "caminho" para guiar a criação de *k-traces* que serão unidos para a construção de um *trace* maior.

O primeiro passo deste algoritmo consiste em gerar aleatoriamente um caminho que será explorado. Para isso, podemos modificar o Algoritmo 3.1.1 para que, ao invés de gerar um *trace* que transforme π_0 em π_d, ele produza a lista de permutações intermediárias gerada pela sequência de reversões utilizadas e o conjunto de *i-traces* que convertem π_0 em π_i. O Algoritmo 3.1.3 lista os passos da versão modificada do Algoritmo 3.1.1.

De posse de uma sequência de permutações que vão de π_0 até π_d, podemos produzir conjuntos de *traces* menores para formação de *traces* de soluções que transformam π_0 em π_d. Para isso, podemos definir uma janela de tamanho w que será utilizada para a construção de *w-traces* que convertem π_i em $\pi_{(i+w)}$. Esta janela é deslizada ao longo da sequência de permutações e os *w-traces* gerados são combinados para a construção de *traces* maiores.

A cada passo i da janela, o Algoritmo 2.5.1 é usado para enumerar todos os *w-traces* que transformam π_i em $\pi_{(i+w)}$. Feito isso, este conjunto de *w-traces* é combinado ao conjunto de *i-traces* que convertem π_0 em π_i. Com essa operação,

Algoritmo 3.1.3: Algoritmo para geração de um conjunto de permutações intermediárias referentes a uma sequência ótima e aleatória de reversões que transforma π_0 em π_d e do conjunto associado de *i-traces* que transformam π_0 em π_i.

Entrada: π_0 e π_d
Saída: Dois vetores de tamanho $d+1$ contendo a lista de permutações intermediárias entre π_0 em π_d (P) e a lista de *i-traces* que transformam π_0 em π_i (I).

```
1  início
2      T_0 ← trace vazio;
3      I ← Vetor de tamanho d + 1;
4      P ← Vetor de tamanho d + 1;
5      I[0] ← ∅;
6      para i ← 0 até d − 1 faça
7          P[i] ← π_i;
8          ρ ← reversão aleatória da lista de optimal 1-sequences de π_i;
9          T_{i+1} ← T_i + ρ;
10         I[i + 1] ← T_{i+1};
11         π_{i+1} ← π_i ∘ ρ;
12     fim
13     P[d] ← π_d;
14     Retorne (P, I);
15 fim
```

obtemos um conjunto de $(i+w)$-*traces* que transformam π_0 em $\pi_{(i+w)}$.

Sejam π_0 a permutação origem, π_d a permutação alvo, w o tamanho da janela, t_{max} o tempo máximo de execução e t_{ac} o tempo acumulado desde o início da execução. O Algoritmo 3.1.4 descreve os passos para a geração de um conjunto aleatório de *traces* com o auxílio de uma janela deslizante dentro de um determinado intervalo de tempo.

No Algoritmo 3.1.4, o *trace* armazenado na posição d do vetor I produzido pelo Algoritmo 3.1.3 (linha 6) é o "*trace* base" de todo o caminho de permutações que será explorado pela etapa de janela deslizante (linhas 8 até 18). Ao término da etapa de janela deslizante, o conjunto de $\mathcal{T}$ receberá, além deste *trace* base, todo o conjunto

Algoritmo 3.1.4: Algoritmo para geração de um conjunto de *traces* aleatórios através da utilização de uma janela deslizante de tamanho w.

Entrada: π_0, π_d, t_{max} e w
Saída: Conjunto de *traces* aleatórios que transformam π_0 em π_d

```
1  início
2      T ← ∅;
3      t_ac ← 0;
4      enquanto t_ac < t_max faça
5          t_inicio ← tempo corrente;
6          (P, I) ← Conjuntos de permutações e de i-traces produzidos pelo
           Algoritmo 3.1.3 para as permutações π_0 e π_d;
7          se I[d] ∉ T então
8              para i ← 0 até d − 1 faça
9                  k ← i + w;
10                 se k > d então k ← d;
11                 T ← conjunto de todos os traces de soluções que transformam
                   P[i] em P[k] produzido pelo Algortimo 2.5.1;
12                 para cada i-trace x ∈ I[i] faça
13                     para cada (k − i)-trace y ∈ T faça
14                         z ← k-trace construído a partir da adição das reversões
                           de y em x; I[k] ← I[k] ∪ z;
15                     fim
16                 fim
17                 I[i] ← ∅; /* Liberar memória */
18             fim
19             T ← T ∪ I[d];
20         fim
21         t_fim ← tempo corrente;
22         t_ac ← t_ac + (t_fim − t_inicio);
23     fim
24     Retorne T;
25 fim
```

de *traces* que foram explorados em torno deste caminho de permutações.

Um *trace* pode representar inúmeras soluções e, por isso, diversos caminhos diferentes podem dar origem a um mesmo *trace*. O *trace* base é construído aleatoriamente e, eventualmente, ele pode fazer parte de um conjunto de *traces* gerados anteriormente durante o processamento de outro caminho de permutações.

Com o objetivo de se maximizar o número de *traces* gerados, a verificação da linha 7 do Algoritmo 3.1.4 evita que caminhos que já possuem *traces* bases presentes no conjunto de *traces* calculados sejam explorados.

Trabalhando com o tamanho da janela, podemos encontrar um compromisso entre a produção de um maior número de w-*traces* e a redução do número de ramos mortos explorados.

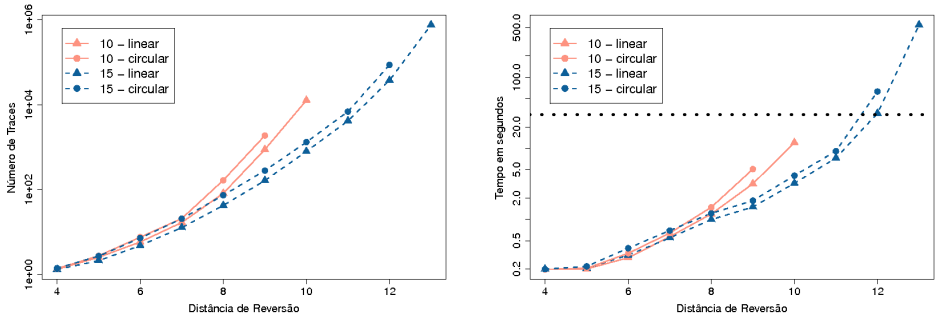
3.2 Avaliação dos algoritmos

Antes de realizar testes comparativos com os algoritmos propostos, procuramos avaliar qual o tempo médio necessário para se enumerar todos os *traces* de uma permutação quando utilizamos o Algoritmo 2.5.1 (Algoritmo de enumeração de *traces* com estrutura de pilha).

Para isso, criamos permutações aleatórias, lineares e circulares, com 10 e 15 elementos e sem obstáculos. Utilizando $n = 10$, definimos permutações com distância de reversão variando entre 4 e 10 para as permutações lineares e entre 4 e 9 para as permutações circulares. No caso de $n = 15$, os intervalos de valores de distância de reversão variaram entre 4 e 13 para as permutações lineares e entre 4 e 12 para as permutações circulares. Para cada trinca $(tipo, n, d(\pi))$, onde $tipo \in \{linear, circular\}$, geramos um conjunto de 500 permutações aleatórias.

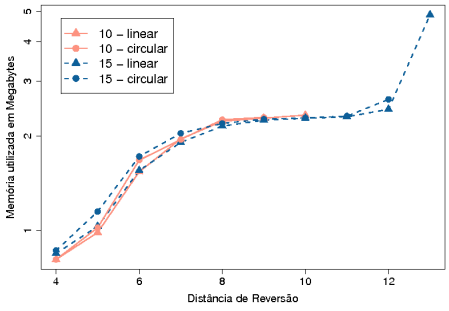
O Algoritmo 2.5.1 foi utilizado para processar as permutações criadas. Durante o processamento de cada permutação, anotamos o número de *traces* produzidos, o tempo total de execução e a memória máxima utilizada. Feito isso, calculamos o número de *traces* médio, o tempo de execução médio e a memória utilizada média para cada conjunto de 500 permutações. Os gráficos da Figura 3.2 mostram o comportamento destes valores médios conforme aumentamos o valor da distância de reversão para as permutações lineares e circulares com 10 e 15 elementos.

Analisando o gráfico da Figura 3.2(a) podemos ver como o número de *traces* cresce exponencialmente conforme aumentamos o número de elementos e a distância de



(a) Número médio de *traces*

(b) Tempo de execução médio



(c) Memória máxima média

Figura 3.2: Grupos de 500 permutações aleatórias, lineares e circulares, com 10 e 15 elementos e diversas distâncias de reversão foram criados e processados com o Algoritmo 3.1.2. Para cada grupo, foram calculados o número médio de *traces* enumerados, o tempo de processamento médio e a memória utilizada máxima média. A linha horizontal pontilhada no gráfico 3.2(b) indica a faixa de 30 segundos.

reversão das permutações. Para um mesmo valor de distância de reversão, podemos ver que o número de *traces* cresce à medida que a razão $d(\pi)/n$ se aproxima de 1. Por exemplo, considerando uma distância de reversão 9, as permutações de 10 elementos apresentam um número maior de *traces* do que as permutações de 15 elementos.

O gráfico mostra também que a quantidade de *traces* das permutações circulares é levemente maior do que a exibida pelas permutações lineares para uma mesma configuração de n e $d(\pi)$. Esse resultado é esperado pois as permutações circulares oferecem mais opções de movimentos do conjunto de marcadores, já que podemos mover elementos do início e do final da permutação com um único movimento.

As mesmas observações feitas para o gráfico da Figura 3.2(a) podem ser feitas para o gráfico Figura 3.2(b). Isso indica, que o tempo de execução é proporcional ao número de soluções que devem ser enumeradas.

O gráfico de utilização de memória da Figura 3.2(c) mostra que o consumo aumenta à medida que as permutações ficam mais complexas e possuem números maiores de *traces*. O padrão da curva, no entanto, é diferente dos padrões apresentados nos outros dois gráficos. Esta diferença se deve ao fato de que a quantia de memória utilizada está associada ao número de reversões que estão presentes em cada nível da pilha e não ao número total de *traces* produzidos durante a execução.

Para avaliar o comportamento dos algoritmos propostos, decidimos escolher um conjunto de permutações que tivesse um tempo médio de processamento nem demasiado pequeno nem demasiado grande. Além disso, como os comportamentos das permutações lineares e circulares são parecidos, optamos por realizar testes apenas com permutações lineares para simplificar a análise dos dados.

Utilizando estes critérios, escolhemos o conjunto composto por permutações lineares com 15 elementos e distância de reversão 12. Este conjunto, que apresenta um tempo médio de 30 segundos para a enumeração do conjunto completo de *traces*, foi processado pelos seguintes algoritmos:

- `Random`: Algoritmo de geração de *traces* aleatórios (Algortimo 3.1.2);
- `Stack`: Algoritmo de enumeração de *traces*, que utiliza a estrutura de pilha, limitado por tempo (Algoritmo 2.5.1 limitado por tempo);
- `Window X`: Algoritmo de enumeração parcial de *traces* com janela deslizante de tamanho X (Algoritmo 3.1.4).

Para o algoritmo de janela deslizante, optamos por testar os valores 4, 5 e 6 para os tamanhos de janelas. Assim, para facilitar a descrição, as execuções do Algoritmo 3.1.4 com estes parâmetros serão denominados, respectivamente, `Window 4`, `Window 5` e `Window 6`.

Sabemos que quanto maior for o tamanho w da janela, maior será o número de w-*traces* produzidos. Por outro lado, o aumento do valor w acarreta na geração de um número maior de ramos mortos. Assim, os valores 4, 5 e 6 foram escolhidos para avaliação de como os diferentes tamanhos de janela afetam o número de *traces* gerados pelo algoritmo.

No entanto, devemos também esclarecer a razão pela qual não testamos janelas com valores menores do que 4.

Teorema 3.2.1. *O valor mínimo de distância de reversão para que uma permutação possua obstáculos é* 3.

Prova Uma componente não orientada (obstáculo) de tamanho mínimo, que possui um único ciclo, é composta por um 3-*ciclo* não orientado. A Figura 3.3(a) exibe um exemplo de uma componente deste tipo.

Para transformar uma componente não orientada em orientada necessitamos aplicar uma reversão de corte. Esta reversão apenas transforma o ciclo e não altera o número de ciclos total do grafo de *breakpoints*. A aplicação de uma reversão de corte em um 3-*ciclo* não orientado gera um 3-*ciclo* orientado.

Até aqui, temos que uma reversão foi aplicada na permutação e, agora, precisamos saber quantas permutações a mais serão necessárias para eliminar este 3-*ciclo* orientado e obter apenas ciclos triviais.

A distância de reversão de uma permutação é dada pela fórmula $d(\pi) = b(\pi) - c(\pi) + h(\pi) + f(\pi)$, onde $b(\pi)$ é o número de arestas pretas no grafo de *breakpoints*, $c(\pi)$ é o número de ciclos, $h(\pi)$ é o número de obstáculos (*hurdles*) presentes na permutação e $f(\pi)$ tem o valor 1 caso a permutação tenha fortaleza e 0, caso contrário. O valor de $b(\pi)$ é igual a $n+1$ para permutações lineares e igual a n para permutações circulares.

Um 3-*ciclo* orientado não contém obstáculos e tão pouco fortalezas. Logo, $h(\pi) = f(\pi) = 0$ e $d(\pi) = b(\pi) - c(\pi)$. Supondo, sem perda de generalidade, que estamos lidando com uma permutação linear, temos que $b(\pi) = n+1$. O valor de $c(\pi)$ é igual ao número de ciclos triviais mais um 3-*ciclo*.

Em uma permutação orientada, o número de ciclos triviais é igual ao número de arestas pretas. Como o 3-*ciclo* ocupa três arestas pretas, temos que o número de ciclos triviais, na permutação corrente, é dado por $n+1-3 = n-2$. Logo, temos que $d(\pi) = (n+1) - (n-2+1) = 2$. Portanto, um total de 3 reversões são necessárias para se eliminar um 3-*ciclo* não orientado.

Uma componente não orientada formada por mais de um ciclo, tem seu tamanho mínimo, quando dois 2-*ciclos* não orientados se sobrepõem. A Figura 3.3(b) mostra um exemplo de uma componente deste tipo.

Uma reversão de corte transforma esta componente não orientada em uma componente orientada composta por dois 2-*ciclos*, um orientado e outro não. Como a

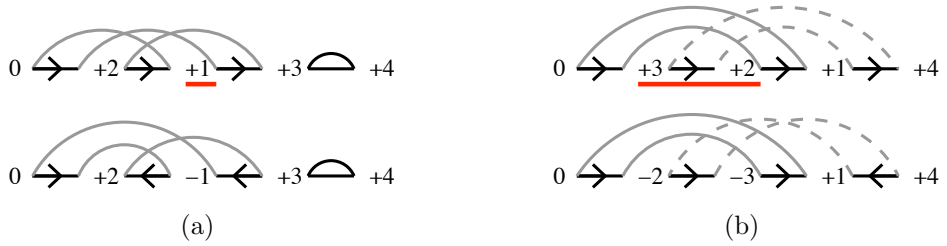


Figura 3.3: Obstáculos de tamanho mínimo. (a) A permutação $(+2, +1, +3)$ possui uma componente não orientada composta por um 3-*ciclo* não orientado. Após a reversão $\{1\}$, obtemos a permutação $(+2, -1, +3)$ que possui um 3-*ciclo* orientado. (b) A permutação $(+3, +2, +1)$ possui uma componente não orientada composta por dois 2-*ciclos* não orientados. Após a reversão $\{2,3\}$, obtemos a permutação $(-2, -3, +1)$ que possui dois 2-*ciclos*, um orientado e outro não.

componente resultante é orientada, a permutação não possui nenhum obstáculo.

Repetindo a análise feita anteriormente, temos que $h(\pi) = f(\pi) = 0$ e $d(\pi) = b(\pi) - c(\pi)$. Neste caso, para obter o número de ciclos triviais da permutação corrente devemos subtrair as 4 arestas pretas ocupadas pelos dois 2-*ciclos* do número total de arestas pretas $(n + 1 - 4 = n - 3)$.

O valor de $c(\pi)$ é dado pelo número de ciclos triviais mais os dois 2-*ciclos* da componente orientada. Assim, temos que $d(\pi) = (n+1) - (n-3+2) = 2$. Portanto, para eliminar uma componente não orientada composta por dois 2-*ciclos* precisamos de três reversões.

Em ambos os casos, necessitamos de uma reversão para transformar a componente não orientada em orientada e duas reversões para eliminar esta componente orientada. Se olharmos para o cálculo da distância da permutação original (com a componente não orientada), esta reversão a mais é dada por $h(\pi) = 1$, que é o número de obstáculos presentes na permutação.

A adição de novos ciclos não triviais ao grafo ou o aumento do número de arestas de um ciclo não trivial acarretará na diminuição de ciclos triviais no grafo de *breakpoints*. Para recuperar os ciclos triviais utilizados, necessitamos de mais reversões.

Como os obstáculos analisados tem tamanho mínimo, não existe alternativa para a construção de uma componente não orientada menor que exija menos reversões. Assim, podemos concluir que a distância de reversão mínima para que uma permutação tenha obstáculo é igual a 3. □

Lema 3.2.1. *Uma permutação que possui distância de reversão igual a* 1 *possui apenas um* 1-trace *de soluções ótimas.*

Prova Como consequência do Teorema 3.2.1, uma permutação com distância de reversão igual a 1 não possui obstáculos. Assim, temos que $c(\pi) = b(\pi) - 1$. Nestas condições, além dos ciclos triviais, temos um único 2-*ciclo* orientado no grafo de *breakpoints*. Os grafos da Figura 3.4 mostram dois exemplos de permutações lineares que possuem distância de reversão 1.

Existe uma única reversão de quebra que divide um 2-*ciclo* orientado em dois ciclos triviais. Logo existe um único 1-*trace* referente a esta única reversão possível. □



(a) $(-1, 2)$ (b) $(-2, -1)$

Figura 3.4: Exemplos de permutações que possuem distância de reversão 1. A única reversão possível transforma o 2-*ciclo* orientado em dois ciclos triviais.

Lema 3.2.2. *Uma permutação que possui distância de reversão igual a* 2 *possui apenas um* 2-trace *de soluções ótimas.*

Prova Como consequência do Teorema 3.2.1, uma permutação com distância de reversão igual a 2 não possui obstáculos. Neste caso, os ciclos não triviais podem assumir três configurações possíveis:

1. *Dois* 2-ciclos *orientados que não se cruzam formando duas componentes orientadas.* (Figura 3.5(a))

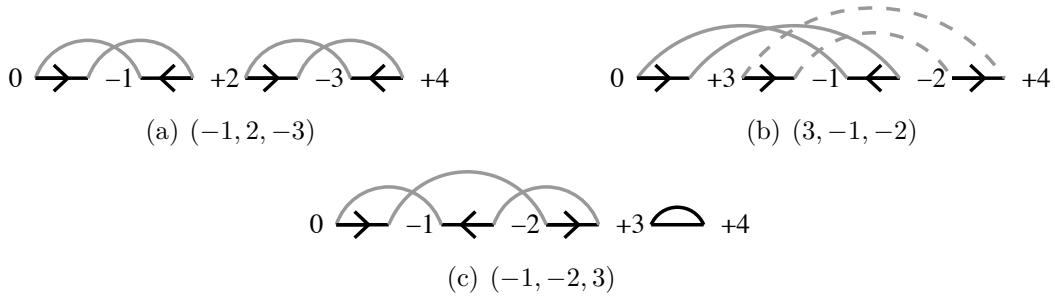
   Neste caso, cada 2-*ciclo* pode ser quebrado em dois ciclos triviais por uma reversão de quebra. Como cada ciclo é independente um do outro, a ordem das reversões não importa.

2. *Um* 2-ciclo *orientado que cruza com um* 2-ciclo *não orientado formando uma componente orientada.* (Figura 3.5(b))

O ciclo orientado é quebrado por uma reversão de quebra em dois ciclos triviais e o ciclo não orientado se transforma em um ciclo orientado. Esta reversão deve ser obrigatoriamente a primeira. Uma nova reversão de quebra é aplicada para transformar o ciclo restante em dois ciclos triviais.

3. *Um* 3-ciclo *orientado que forma uma componente orientada.* (Figura 3.5(c))

   Uma reversão de quebra transforma o 3-*ciclo* em um ciclo trivial e um 2-*ciclo* orientado. Uma nova reversão transforma o ciclo não trivial restante em dois ciclos triviais. A ordem das reversões não importa.



(a) $(-1, 2, -3)$ (b) $(3, -1, -2)$ (c) $(-1, -2, 3)$

Figura 3.5: Exemplos de permutações que possuem distância de reversão 2. (a) As reversões {1} e {3} podem ser aplicadas em qualquer ordem. (b) A reversão {1,3} deve ser aplicada antes da reversão {2,3}. (c) As reversões {1} e {2} podem ser aplicadas em qualquer ordem.

No primeiro e terceiro caso, como a ordem das reversões não importam, a combinação delas forma um 2-*trace* de altura 1. No outro caso, como a primeira reversão deve ser aplicada antes da primeira, isso significa que uma sobrepõe a outra e que, portanto, temos um 2-*trace* de altura 2.

Cada 2-*ciclo* orientado possui apenas uma reversão que o elimina. Assim, quando temos dois 2-*ciclos*, apenas duas reversões são possíveis. Logo, neste caso, existe apenas um 2-*trace* possível.

Em um 3-*ciclo* orientado, temos que duas arestas pretas estão em um mesmo sentido e uma aresta preta que está em sentido oposto. Para transformar este ciclo em um ciclo trivial e um 2-*ciclo* orientado, devemos efetuar uma reversão que age sobre elementos que estão nas extremidades de duas arestas pretas com sentidos opostos. Caso os elementos revertidos estejam em extremidades de duas arestas

pretas com mesmo sentido, o resultado da reversão será um 3-*ciclo* não orientado (estaremos aumentando em uma unidade a distância de reversão pois criaremos um obstáculo).

Quando temos um 3-*ciclo* orientado, temos duas opções para selecionar elementos que estão na extremidade de arestas pretas com sentidos opostos. Quando uma opção é selecionada, após a reversão, a outra opção se transforma em um 2-*ciclo* orientado. Assim, neste caso, temos que apenas duas reversões são possíveis e, portanto, apenas um 2-*trace* é possível. □

Teorema 3.2.2. *Uma permutação possui dois* traces *distintos quando sua distância de reversão é no mínimo igual a* 3*.*

Prova Os Lemas 3.2.1 e 3.2.2 mostram que permutações com distâncias de reversão menores do que 3 possuem apenas um *trace*. Assim, para provar este teorema, precisamos mostrar apenas um exemplo de uma permutação com distância de reversão igual a 3 que possui dois *traces*.

A permutação $(+3, +2, +1)$ possui um grafo de *breakpoints* composto por dois 2-*ciclos* não orientados que se sobrepõem. Como discutimos na prova do Teorema 3.2.1, esta configuração exige três reversões para ser resolvida e a primeira delas serve para transformar a componente não orientada em uma componente orientada.

A reversão {1,2} gera a permutação $(+3, -1, -2)$ que possui uma componente orientada composta por um 2-*ciclo* não orientado e um 2-*ciclo* orientado. A reversão {1,3} quebra o ciclo orientado e gera a permutação $(+1, -3, -2)$. Esta permutação possui um 2-*ciclo* orientado que é eliminado com a reversão {2,3}.

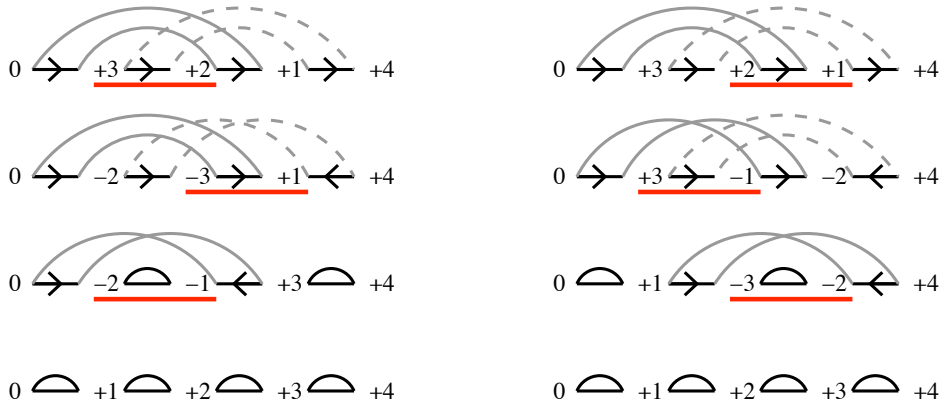
Contudo, a permutação $(+3, +2, +1)$ pode ser ordenada de outra maneira. A reversão {2,3} pode ser aplicada para transformar um 2-*ciclo* não orientado em um 2-*ciclo* orientado gerando a permutação $(-2, -3, +1)$. A reversão {1,3} quebra o 2-*ciclo* orientado e produz a permutação $(-2, -1, +3)$. Finalmente, a reversão {1,2} conclui a ordenação.

Assim, a permutação $(+3, +2, +1)$ possui dois *traces*:

{1,2}|{1,3}|{2,3} e {2,3}|{1,3}|{1,2}

A Figura 3.6 mostra os grafos de *breakpoints* produzidos durante a ordenação da permutação $(+3, +2, +1)$ segundo os dois possíveis *traces* de soluções.

□



(a) *Trace* $\{2,3\}|\{1,3\}|\{1,2\}$ (b) *Trace* $\{1,2\}|\{1,3\}|\{2,3\}$

Figura 3.6: Permutação com $d(\pi) = 3$ e dois *traces* de soluções

Teorema 3.2.3. *Uma permutação sem obstáculos possui dois* traces *distintos quando sua distância de reversão é no mínimo igual a* 4*.*

Prova Pelo Teorema 3.2.1 sabemos que permutações com distâncias de reversão menores do que 3 não possuem obstáculo. Adicionalmente, os Lemas 3.2.1 e 3.2.2 nos garantem que permutações com distâncias de reversão menores do que 3 possuem apenas um *trace*.

Assim, para provar este teorema, precisamos mostrar que não existe permutação sem obstáculos e com distância de reversão igual a 3 que possua mais de um *trace*.

As componentes orientadas de um grafo de *breakpoints* de uma permutação que não possui obstáculos e possui distância de reversão igual a 3 podem assumir as seguintes configurações:

1. *Três componentes orientadas*

   Neste caso, temos três 2-*ciclos* que não se sobrepõem e, por isso, são independentes entre si. Uma reversão de quebra deve ser aplicada sobre cada componente. Como as componentes são independentes, as reversões não se sobrepõem e, por isso, temos um único 3-*trace* de altura 1. A Figura 3.7(a) mostra um exemplo de permutação com distância de reversão igual a 3 e composta por três componentes orientadas.

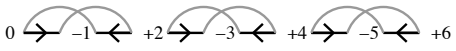
2. *Duas componentes orientadas*

   Neste caso, temos que uma componente orientada é responsável por uma reversão e outra componente orientada é responsável pelas outras duas reversões.

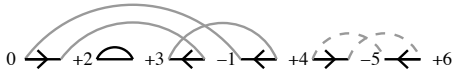
   A componente que possui apenas uma reversão é, obrigatoriamente, um 2-*ciclo* orientado. A segunda componente é maior e pode ser formada por um 3-*ciclo* orientado (Figura 3.7(b)) ou por dois 2-*ciclos*, um orientado e outro não, que se sobrepõem (Figura 3.7(c)).

   Se a componente orientada maior for um 3-*ciclo*, temos que ele gera um único 2-*trace* de altura 1. Se ele for composto por um 2-*ciclo* orientado que sobrepõe um 2-*ciclo* não orientado, teremos um único 2-*trace* de altura 2.

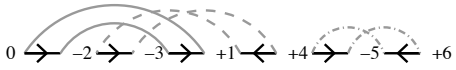
   A combinação da reversão da menor componente com o 2-*trace* da componente maior gerará um único 3-*trace* que poderá ter altura 1 ou 2 conforme o tipo da componente maior. No caso do 3-*trace* com altura 2, como a reversão da componente menor é independente das reversões do outro componente, ela ficará posicionada na primeira sub-palavra do 3-*trace*.



(a) *Trace* $\{1\}\{3\}\{5\}$



(b) *Trace* $\{1,-,3\}|\{2,3\}\{5\}$



(c) *Trace* $\{1,3\}\{5\}|\{1,2\}$

Figura 3.7: Permutações com $d(\pi) = 3$ contendo 2 e 3 componentes orientadas. (a) Permutação $(-1,+2,-3,+4,-5)$ possui três componentes orientadas. (b) Permutação $(+2,+3,-1,+4,-5)$ possui um 3-*ciclo* orientado e um 2-*ciclo* orientado. (c) Permutação $(-2,-3,+1,+4,-5)$ possui uma componente composta por um 2-*ciclo* orientado e uma componente composta por dois 2-*ciclos*, um orientado e outro não.

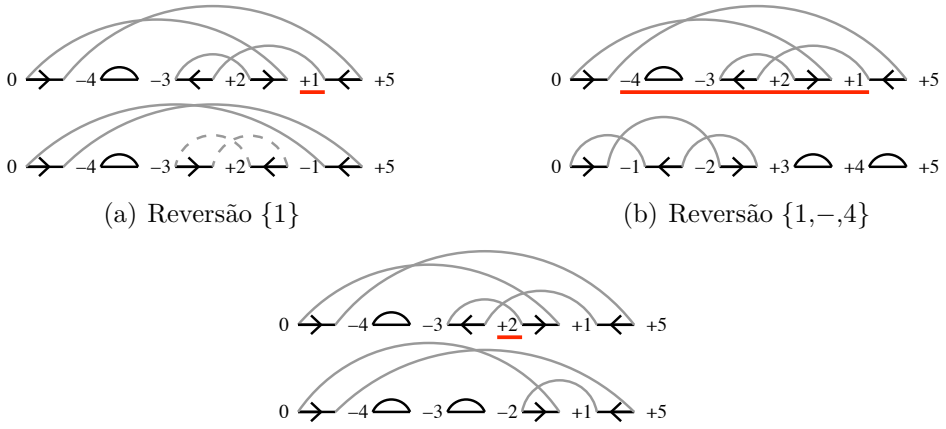
3. *Uma componente orientada*

   As seguintes configurações são possíveis quando temos uma única componente orientada:

(a) *Um* 4-ciclo *orientado*

Quando temos um 4-*ciclo* orientado, duas configurações são possíveis. Em uma configuração, os arcos que formam o ciclo cruzam três vezes entre si. Na segunda configuração, os arcos se cruzam quatro vezes.

Se o ciclo apresenta três cruzamentos, três reversões que não se sobrepõem podem ser aplicadas para quebrá-lo. Elas quebram o 4-*ciclo* em um ciclo trivial e um 3-*ciclo* orientado ou em dois 2-*ciclos* orientados que não se sobrepõem. Após a seleção de uma das reversões possíveis, as outras duas reversões são utilizadas para quebrar o(s) ciclo(s) restante(s). Nesta situação, temos um 3-*trace* de altura 1. A Figura 3.8 exibe um exemplo de permutação que apresenta estas características.



(a) Reversão {1}

(b) Reversão {1,−,4}

(c) Reversão {2}

Figura 3.8: A permutação $(-4,-3,+2,+1)$ possui um 4-*ciclo* orientado com três cruzamentos entre os arcos. O *trace* {1}{1,−,4}{2} agrupa as três reversões que podem ser aplicadas nesta permutação.

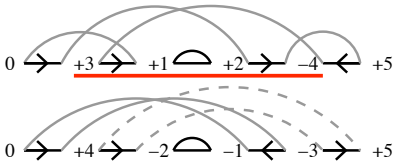
No caso do ciclo possuir 4 cruzamentos, apenas duas reversões podem ser aplicadas para quebrá-lo. Ao aplicarmos uma delas, o 4-*ciclo* pode ser dividido em um ciclo trivial e um 3-*ciclo* orientado ou em uma componente orientada composta por dois 2-*ciclos*, um orientado e outro não.

Se um 3-*ciclo* orientado foi formado, ele pode ser eliminado com duas reversões que não se sobrepõem. Contudo, uma destas duas reversões terá sobreposição com a reversão aplicada anteriormente. Neste caso, teremos

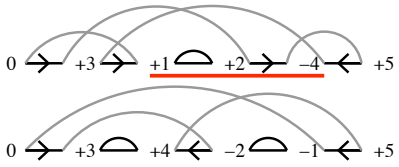
a formação de um 3-*trace* com altura 2, sendo que, a primeira sub-palavra possui duas reversões.

Se uma componente orientada composta por dois 2-*ciclos*, um orientado e outro não, foi criada, sabemos que podemos eliminá-la com duas reversões que se sobrepõem. Como a primeira reversão aplicada não sobrepõe a primeira reversão que elimina o componente formado pelos dois ciclos, teremos a formação de um 3-*trace* com altura 2 e primeira sub-palavra contendo duas reversões.

A Figura 3.9 exibe uma permutação que possui um 4-*ciclo* orientado que apresenta 4 cruzamentos entre seus arcos.



(a) Reversão $\{1, -, 4\}$



(b) Reversão $\{1, 2, 4\}$

Figura 3.9: A permutação $(+3, +1, +2, -4)$ possui um 4-*ciclo* orientado com 4 cruzamentos entre os arcos. A primeira sub-palavra do *trace* $\{1, -, 4\}\{1, 2, 4\}|\{3, 4\}$ agrupa as duas reversões que podem ser aplicadas nesta permutação.

(b) *Um* 2-ciclo *orientado que sobrepõe um* 3-ciclo *não orientado*

Quando temos esta configuração, apenas uma reversão pode ser aplicada. Esta reversão age de forma a eliminar o 2-*ciclo* orientado e transformar o 3-*ciclo* não orientado em um 3-*ciclo* orientado.

As reversões que eliminam o 3-*ciclo* orientado não se sobrepõem entre si. Contudo, ambas sobrepõem a reversão aplicada inicialmente. Assim, temos como resultado um 3-*trace* de altura 2 em que a segunda sub-palavra possui duas reversões.

A Figura 3.10 exibe dois exemplos de permutações que apresentam um 2-*ciclo* orientado e um 3-*ciclo* não orientado que se sobrepõem.

(c) *Um* 2-ciclo *orientado que sobrepõe um* 3-ciclo *orientado*

Quando um 2-*ciclo* orientado sobrepõem um 3-*ciclo* orientado, duas situações podem ocorrer. Na primeira, o 2-*ciclo* cobre apenas uma aresta

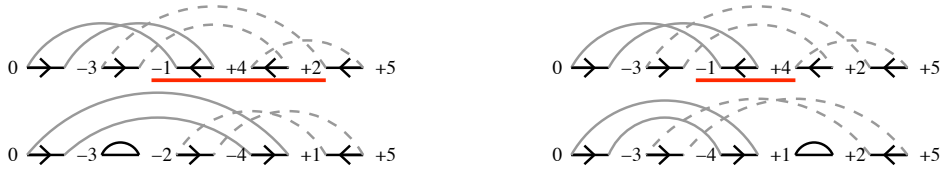


(a) Reversão {1,3} (b) Reversão {1,2,4}

Figura 3.10: Permutações com $d(\pi) = 3$ e uma componente orientada composta por um 2-*ciclo* orientado e um 3-*ciclo* não orientado. A primeira reversão elimina o 2-*ciclo* orientado e transforma o 3-*ciclo* não orientado em orientado. (a) A permutação $(-3, -1, +4, -2)$ possui o *trace* $\{1, 3\}|\{2, -, 4\}\{3, 4\}$. (b) A permutação $(-2, +4, -1, +3)$ possui o *trace* $\{1, 2, 4\}|\{2, 3\}\{2, -, 4\}$.

preta do 3-*ciclo*. Na segunda, o ciclo menor cobre 2 arestas pretas do ciclo maior.

Na primeira situação, duas reversões podem ser aplicadas na permutação e ambas geram uma nova componente orientada formada por dois 2-*ciclos*, um orientado e outro não. Ao selecionarmos uma reversão, a outra reversão se torna a primeira reversão que desfaz a componente orientada composta por dois 2-*ciclos*. Como sabemos que a terceira reversão irá sobrepor a segunda reversão aplicada, temos que o resultado é um 3-*trace* de altura 2 com a primeira sub-palavra contendo duas reversões. A Figura 3.11 exibe um exemplo de permutação que possui esta estrutura.



(a) Reversão $\{1, 2, 4\}$ (b) Reversão $\{1, 4\}$

Figura 3.11: A permutação $(-3, -1, +4, +2)$ possui uma componente orientada composta por um 3-*ciclo* orientado e um 2-*ciclo* orientado que cobre uma aresta preta do ciclo maior. A primeira sub-palavra do *trace* $\{1, 2, 4\}\{1, 4\}|\{1, -, 3\}$ contém as duas reversões que podem ser aplicadas.

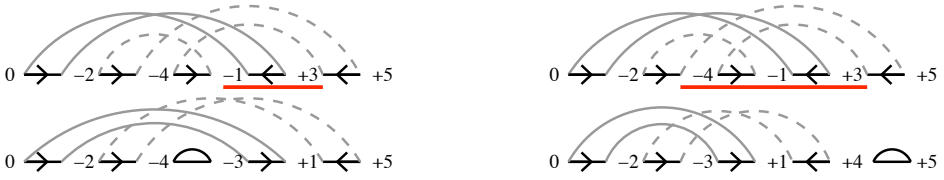
Na segunda situação, novamente, temos duas reversões que podem ser aplicadas na permutação. Uma das reversões transforma a componente

em um ciclo trivial e um 3-*ciclo* orientado. A outra reversão transforma a componente em dois 2-*ciclos* que se sobrepõem, um orientado e outro não.

Quando um 3-*ciclo* é formado, temos que uma das duas reversões, que o elimina, sobrepõe a primeira reversão aplicada. Assim, o resultado é um 3-*trace* de altura 2 com duas reversões na primeira sub-palavra.

Quando os dois 2-*ciclos* são formados, a primeira reversão que deve ser aplicada para eliminá-los não sobrepõem a primeira reversão que foi aplicada na permutação. Como sabemos que a terceira e última reversão terá sobreposição com a segunda reversão, podemos concluir que obteremos também um 3-*trace* de altura 2 com duas reversões na primeira sub-palavra.

A Figura 3.12 exibe um exemplo de permutação que possui um 3-*ciclo* orientado e um 2-*ciclo* orientado que cobre duas arestas pretas do ciclo maior.



(a) Reversão $\{1, 3\}$ (b) Reversão $\{1, 3, 4\}$

Figura 3.12: A permutação $(-2, -4, -1, +3)$ possui uma componente orientada composta por um 3-*ciclo* orientado e um 2-*ciclo* orientado que cobre duas arestas pretas do ciclo maior. A primeira sub-palavra do *trace* $\{1, 3\}\{1, 3, 4\}|\{1, 2\}$ contém as duas reversões que podem ser aplicadas.

(d) *Um* 2-ciclo *não orientado que sobrepõe um* 3-ciclo *orientado*

Nesta situação, apenas uma reversão pode ser aplicada na permutação. Ela gera uma componente orientada composta por dois 2-*ciclos* que se sobrepõem, um orientado e outro não.

A primeira reversão aplicada na permutação sobrepõe a primeira reversão necessária para eliminar a componente composta pelos dois 2-*ciclos*. Logo, nestas condições, obtemos um 3-*trace* de altura 3.

A Figura 3.13 exibe um exemplo de uma permutação que possui estas características.

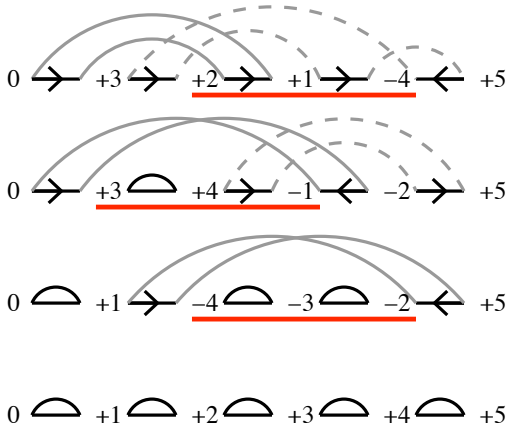


Figura 3.13: A permutação $(+3, +2, +1, -4)$ possui uma componente orientada composta por um 3-*ciclo* orientado e um 2-*ciclo* não orientado. O *trace* $\{1, 2, 4\}|\{1, 3, 4\}|\{2, -, 4\}$ lista as reversões que ordenam esta permutação.

(e) *Dois* 2-ciclos *não orientados que se sobrepõem e um* 2-ciclo *orientado que sobrepõe um dos ciclos não orientados*

Nestas condições, apenas uma reversão pode ser aplicada na permutação. Ela age sobre o ciclo orientado eliminando-o e transforma os dois ciclos restantes em uma componente orientada composta por dois 2-*ciclos*, um orientado e outro não.

A primeira reversão aplicada na permutação sobrepõe a primeira reversão necessária para decompor a componente composta pelos dois 2-*ciclos*. Portanto, nestas condições, obtemos um 3-*trace* de altura 3.

A Figura 3.14 exibe uma permutação que possui dois 2-*ciclos* não orientados que se sobrepõem e um 2-*ciclo* orientado que sobrepõe um dos ciclos não orientados.

Analisando os casos listados, percebemos que todos sempre possuem um único *trace* de soluções. Logo, concluímos que não é possível produzir dois *traces* de soluções quando a permutação sem obstáculos tem distância de reversão igual a 3.

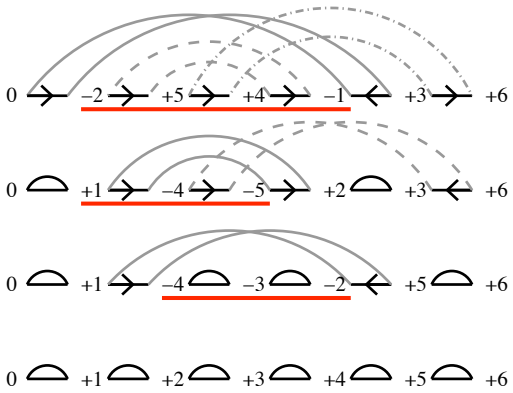


Figura 3.14: A permutação $(-2, +5, +4, -1, +3)$ possui uma componente orientada composta por dois 2-*ciclos* não orientados que se sobrepõem e um 2-*ciclo* orientado que sobrepõe um dos ciclos não orientados. O *trace* $\{1, 2, 4, 5\}|\{2, 3, 5\}|\{2, -, 4\}$ lista as reversões que ordenam esta permutação.

Para $d = 4$, podemos mostrar uma instância em que temos dois *traces* diferentes para uma permutação. Por exemplo, se $\pi = (+1, +4, +2, +6, -3, -5)$ temos dois *traces* que transformam π em ι_6:

$$\{2, -, 6\}\{2, 3, 5, 6\}\{3, 6\}|\{4, -, 6\} \text{ e } \{2, 4, 6\}\{2, 6\}|\{3, 4, 6\}|\{5, 6\}$$

□

Como consequência do Teorema 3.2.2, uma permutação tem que ter no mínimo distância de reversão igual a 3 para gerar mais de um *trace* de soluções. Contudo, se adicionamos a condição de que as permutações não podem ter obstáculos, o Teorema 3.2.3 aumenta a distância de reversão mínima em uma unidade.

Quando todas as janelas de um caminho geram apenas um *w-trace*, a combinação de todos os *w-traces* gerados resultará em um único *trace*. Como a idéia do algoritmo de janela deslizante é potencializar a geração de *traces*, o ideal é que se evite esta situação.

Como a nossa implementação do algoritmo de enumeração de *traces* trabalha apenas com permutações sem obstáculos, adotamos o tamanho mínimo 4 para os testes com as janelas.

Os algoritmos `Random`, `Stack`, `Window 4`, `Window 5` e `Window 6` foram utilizados para processar o conjunto de 500 permutações aleatórias que possuem 15 elementos

e distância de reversão 12. Considerando que o tempo médio para se enumerar todos os *traces* de uma permutação deste conjunto é 30 segundos, testamos os seguintes intervalos de tempo: 6, 12, 18, 24, 30 e 36 segundos.

Durante a execução dos algoritmos, para cada *trace* produzido, verificamos qual a sua altura e qual o tamanho médio das reversões que o compõem (por exemplo, o tamanho médio das reversões do *trace* {1,2}{3} é $(2 + 1)/2 = 1{,}5$). Os valores de altura observados entre todas as permutações ficaram entre 2 e 12 e os valores de tamanho médio de reversão ficaram entre 2,33 e 11,50.

Baseados nos valores de altura (tamanho médio de reversão) observados, definimos intervalos $(x - 1/2, x + 1/2]$ para $2 \leq x \leq 12$ ($2 \leq x \leq 11$). Para cada trinca (algoritmo A, permutação P, tempo limite T) contamos o número de *traces* que possuíam altura (tamanho médio de reversão) dentro de um dado intervalo I. A mesma contagem foi feita sobre o conjunto completo de *traces* da permutação.

Calculamos para cada intervalo I a porcentagem de *traces* do conjunto completo que foi produzida pelo algoritmo A dentro do tempo limite T para uma dada permutação P. Finalmente, para cada trinca (algoritmo A, tempo limite T, intervalo I), calculamos a média das porcentagens observadas no intervalo I ao longo das 500 permutações processadas pelo algoritmo A sob um tempo limite T.

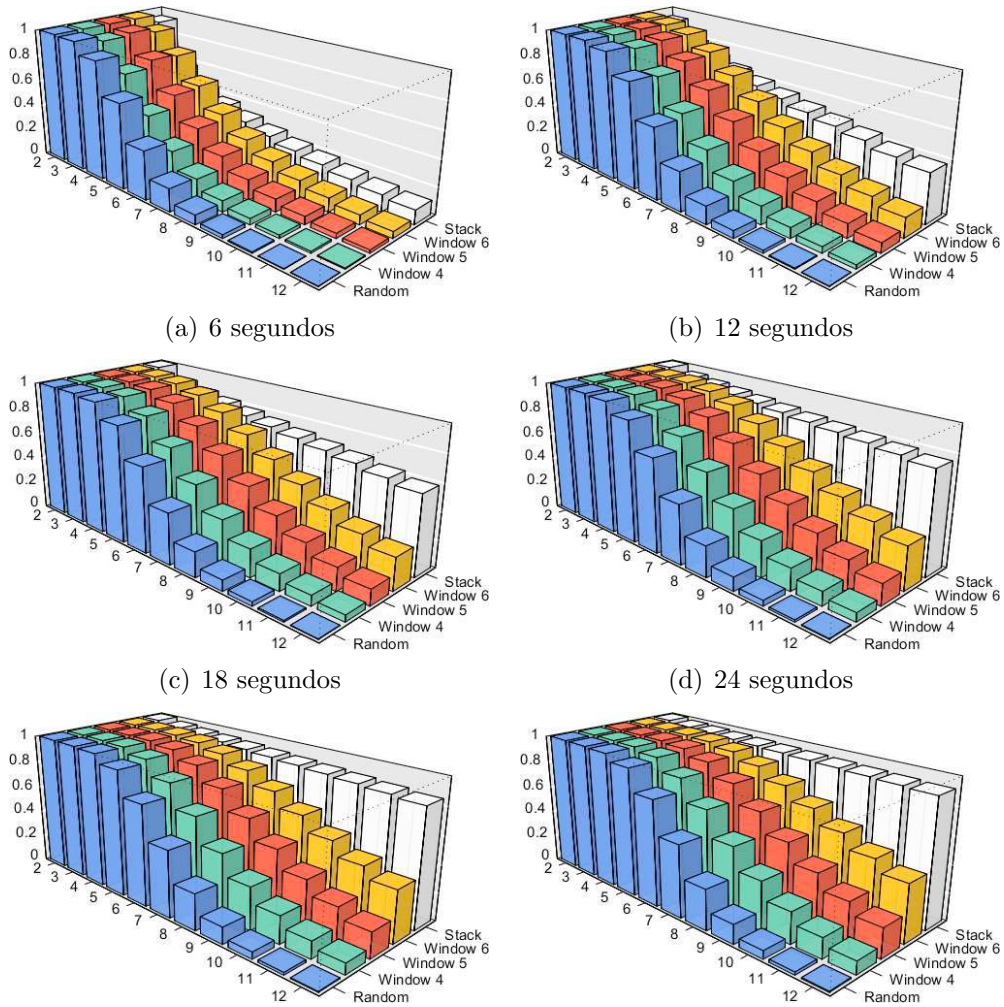
Os gráficos da Figura 3.15 (3.16) mostram o comportamento exibido por cada algoritmo em relação a porcentagem de *traces* com uma determinada altura (tamanho médio de reversão) que foram produzidos conforme o tempo limite de execução que foi aplicado.

A análise dos gráficos da Figura 3.15 mostra que a medida que o tempo limite de execução aumenta, o número de *traces* produzidos cresce.

Neste conjunto de permutações, o algoritmo `Stack` é o que mais se aproxima de encontrar o conjunto completo de *traces*. Como ele é um algoritmo não aleatório, o aumento do tempo limite de execução resulta em um aumento gradual do número de *traces* produzidos.

Em relação aos demais algoritmos, também é possível observar um aumento do número de *traces* conforme o crescimento do tempo limite. Contudo, o padrão exibido pelos algoritmos aleatórios é distinto do mostrado pelo algoritmo `Stack`.

Traces com alturas menores representam conjuntos maiores de soluções ótimas. Devido a isso, a probabilidade de se gerar aleatoriamente um *trace* com menor altura é maior do que a probabilidade de produzir um *trace* com maior altura. Assim, apesar dos algoritmos aleatórios serem capazes de enumerar praticamente 100% dos *traces*



(a) 6 segundos

(b) 12 segundos

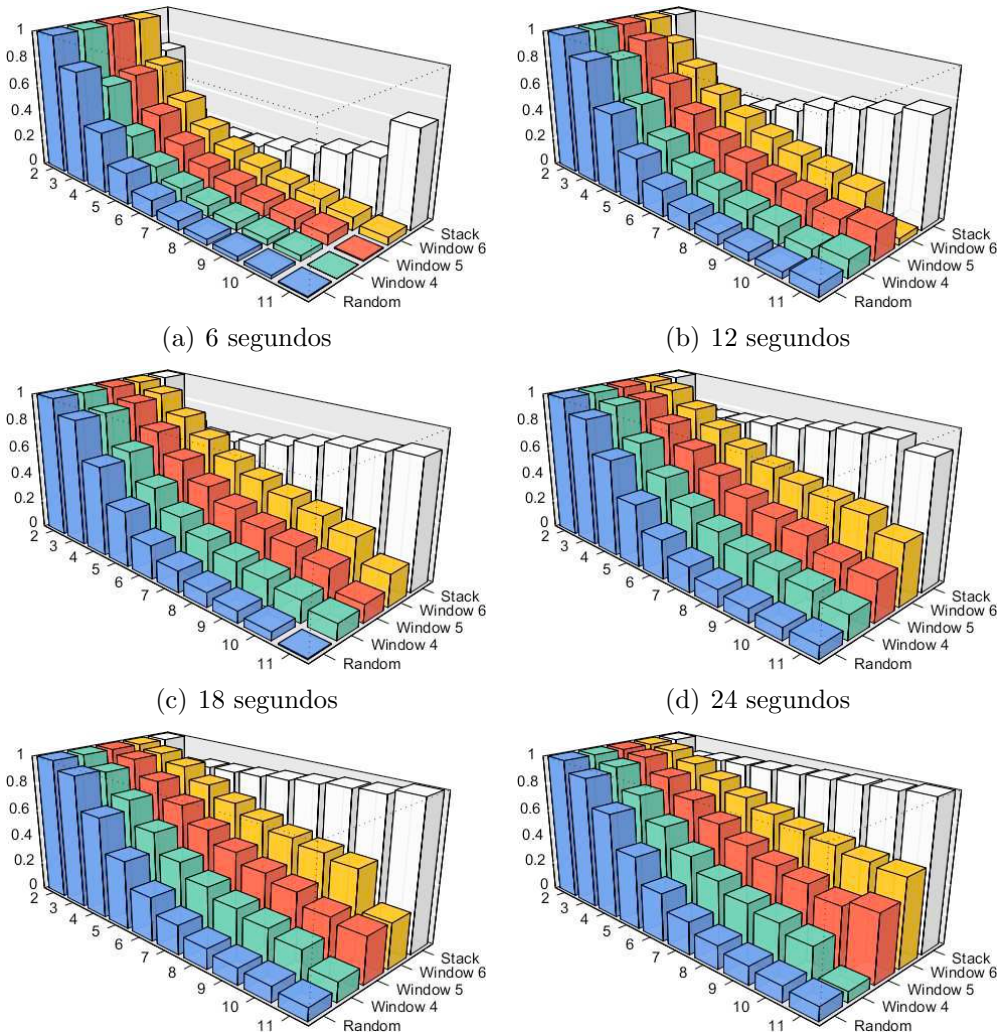
(c) 18 segundos

(d) 24 segundos

(e) 30 segundos

(f) 36 segundos

Figura 3.15: Um conjunto de 500 permutações aleatórias com $n = 15$ e $d(\pi) = 12$ foram processadas com os algoritmos `Random`, `Window 4`, `Window 5`, `Window 6` e `Stack` utilizando-se diferentes valores de tempo limite (6, 12, 18, 24, 30 e 36 segundos). Para cada trinca (permutação, algoritmo, altura do *trace*), anotamos a porcentagem de *traces* que foram encontrados. As barras nos gráficos mostram a média das porcentagens obtidas para cada trinca.



(a) 6 segundos (b) 12 segundos

(c) 18 segundos (d) 24 segundos

(e) 30 segundos (f) 36 segundos

Figura 3.16: Um conjunto de 500 permutações aleatórias com $n = 15$ e $d(\pi) = 12$ foram processadas com os algoritmos `Random`, `Window 4`, `Window 5`, `Window 6` e `Stack` utilizando-se diferentes valores de tempo limite (6, 12, 18, 24, 30 e 36 segundos). Para cada trinca (permutação, algoritmo, tamanho médio das reversões contidas no *trace*), anotamos a porcentagem de *traces* que foram encontrados. As barras nos gráficos mostram a média das porcentagens obtidas para cada trinca.

com alturas 2, 3 e 4, a porcentagem de *traces* encontrados é cada vez menor à medida que aumentamos o valor de altura.

A comparação entre os algoritmos `Random`, `Window 4`, `Window 5` e `Window 6` mostra que o algoritmo `Random` é o que possui a pior capacidade de produzir *traces* com alturas maiores. Em relação aos algoritmos de janela deslizante, os resultados melhoram à medida que aumentamos o tamanho da janela.

Os gráficos da Figura 3.16 mostram um resultado similar. O algoritmo `Stack` novamente é o que mais se aproxima do conjunto completo de *traces* e os algoritmos aleatórios exibem o mesmo padrão observado nos gráficos da Figura 3.15.

O tempo médio para se processar as permutações lineares com $n = 15$ e $d(\pi) = 12$ é de aproximadamente 30 segundos. Trata-se portanto de um conjunto de permutações que pode ter seu conjunto completo de *traces* facilmente enumerado. Contudo, a idéia dos algoritmos, propostos neste capítulo, é produzir uma enumeração parcial de *traces* para conjuntos que demandem mais tempo de processamento.

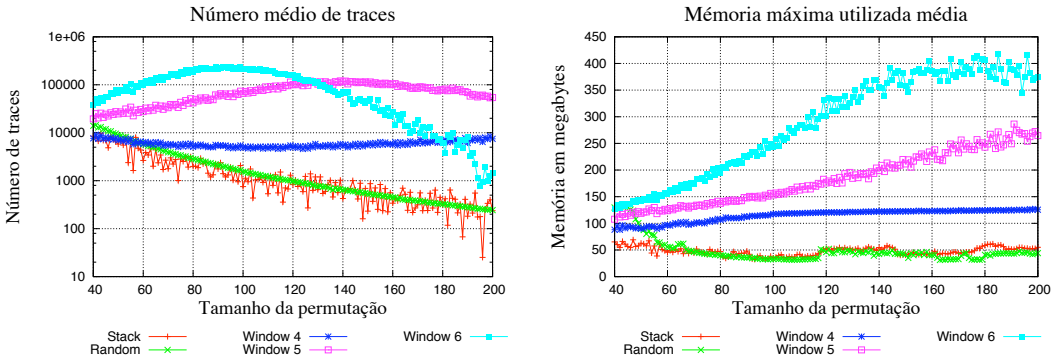
Para testar este aspecto, criamos conjuntos compostos por 100 permutações aleatórias com número de elementos variando entre 40 e 200 e $d(\pi) = \lceil (n+1)/2 \rceil$. Os algoritmos `Stack`, `Random`, `Window 4`, `Window 5` e `Window 6` foram utilizados para processar as permutações com um tempo limite de execução um minuto para cada uma delas. O número de *traces* e a memória máxima utilizada foram anotados para cada permutação processada. Finalmente, os valores médios da quantidade de *traces* e da memória utilizada foram calculados para cada conjunto de 100 permutações.

A Figura 3.17(a) mostra as curvas do número médio de *traces* observados para cada valor de n nos resultados produzidos pelos algoritmos `Stack`, `Random`, `Window 4`, `Window 5` e `Window 6` durante um intervalo limite de um minuto. A Figura 3.17(b) mostra a memória máxima utilizada média observada durante a execução destes algoritmos.

Analisando o gráfico da Figura 3.17(a) podemos verificar que o número de *traces* produzidos pelo algoritmo `Stack` cai conforme os tamanhos das permutações aumentam. Este fenômeno está associado ao maior tempo que o algoritmo `Stack` gasta na exploração de ramos mortos da árvore de *traces*.

O algoritmo `Random` apresenta uma curva similar àquela do algoritmo `Stack`. À medida que os tamanhos das permutações aumentam, o algoritmo `Random` também reduz o número de *traces* produzidos.

Os algoritmos de janela deslizante `Window 4`, `Window 5` e `Window 6` mostram uma capacidade de gerar mais *traces* que os algoritmos `Stack` e `Random`. Para $n < 55$, os



(a) Número médio de *traces* (b) Memória máxima utilizada média

Figura 3.17: Conjuntos de 100 permutações aleatórias com $40 \leq n \leq 200$ e $d(\pi) = \lceil (n+1)/2 \rceil$ foram criados e processados com os algoritmos `Stack`, `Random`, `Window 4`, `Window 5` e `Window 6`. Cada permutação foi submetida a cada algoritmo por um tempo limite de um minuto. O número de *traces* produzidos e a memória máxima utilizada foram anotados e os valores médios foram calculados para cada conjunto de 100 permutações.

algoritmos `Window 5` e `Window 6` produzem mais *traces* que os algoritmos `Stack` e `Random`. Para $n \geq 55$, os três algoritmos de janela deslizante produzem mais *traces* que os algoritmos `Stack` e `Random`.

Para cada *trace* gerado pelo algoritmo `Random`, são selecionadas $d(\pi)$ reversões aleatórias. Estas reversões são selecionadas da lista de *optimal* 1-*sequences* que é obtida para a permutação corrente em relação à permutação alvo. Assim, da primeira à última reversão, estamos trabalhando com permutações que possuem distância de reversão que vão de $d(\pi)$ até 1. Quanto maior a distância de reversão, maior o tempo necessário para se analisar os ciclos do grafo de *breakpoints* em busca de uma reversão segura. Este aumento do tempo de geração das reversões reflete numa menor quantidade de *traces* produzidos à medida que o valor de n cresce e o tempo limite é mantido o mesmo.

Enquanto o algoritmo `Random` trabalha com a produção de reversões para permutações com valores de distância de reversão entre 1 e $d(\pi)$, os algoritmos de janela deslizante utilizam uma estratégia diferente. Se w é o tamanho da janela, em nenhum momento o algoritmo de janela deslizante produzirá listas de *optimal* 1-*sequences* para permutações que tenham distâncias de reversão maiores do que w. Como w

tende a ser muito menor do que $d(\pi)$, o tempo gasto para se produzir as listas de *optimal* 1-*sequences* de permutações com distâncias de reversão igual w é muito menor do que o observado em permutações com distâncias maiores.

Outra vantagem do algoritmo de janela deslizante, é que ele produz todos os w-*traces* entre as permutações π_i e $\pi_{(i+w)}$. Assim, todas as estruturas de grafos de *breakpoints* criadas durante o processo são aproveitadas ao máximo. No caso do algoritmo `Random`, a estrutura de grafo de *breakpoints*, mesmo que ela seja parcialmente criada, é sub-utilizada pois apenas uma reversão será considerada por grafo criado.

Normalmente, quanto maior o número de w, maior será o número de w-*traces* produzidos e, consequentemente, maior será o número de w-*traces* a serem combinados. Devido a isso, podemos ver que o número médio de *traces* produzidos cresce mais rapidamente para o algoritmo `Window 6` e mais lentamente para o algoritmo `Window 4` para valores de n menores do que 80.

A curva apresentada pelo algoritmo `Window 6` possui fórmula de parábola. Durante o intervalo de valores de n entre 40 e 80, o algoritmo `Window 6` apresenta um rápido crescimento no número de *traces* produzidos. Uma patamar pode ser visto para $80 < n < 110$. Para $n \geq 110$, o número de *traces* enumerados passa a ser cada vez menor conforme o valor de n aumenta. A partir de $n = 130$, o algoritmo `Window 6` é superado pelo algoritmo `Window 5` e a partir de $n = 180$, ele também é superado pelo algoritmo `Window 4`.

O algoritmo `Window 5` também apresenta uma curva em forma de parábola. Contudo, apesar de apresentar uma taxa de crescimento menor do que a apresentada por `Window 6`, o intervalo de valores de n onde o número de *traces* produzidos cresce com o tamanho da permutação se estende até aproximadamente $n = 120$. Um patamar pode ser observado no intervalo de valores entre 120 e 160 e, finalmente, para $n > 160$ o algoritmo `Window 5` começa a apresentar redução no número de *traces* produzidos.

O formato de parábola da curva está associado ao processo de combinação de i-*traces* com w-*traces* ao final da enumeração dos w-*traces* entre as permutações π_i e $\pi_{(i+w)}$. Quando combinamos x i-*traces* que transformam π_0 em π_i com y w-*traces* que convertem π_i em $\pi_{(i+w)}$, temos a formação de até $x \times y$ $(i+w)$-*traces* (na combinação, alguns *traces* podem se repetir). Assim, quanto maior a distância de reversão, maior será o número de combinações necessárias e consequentemente, maior o tempo gasto com estas operações.

Em geral, um conjunto de 4-*traces* é menor do que um conjunto de 5-*traces* e muito menor do que um conjunto de 6-*traces*. Consequentemente, o número de

combinações de 4-*traces* será muito menor ao longo da execução do algoritmo quando comparamos com os números observados para configurações maiores de janela. Em função disso, o algoritmo `Window 4` exibe a produção de um número médio de *traces* levemente abaixo de 10000 em todo o intervalo de valores de n testados. Acreditamos que se o intervalo de valores de n fosse grande o suficiente, o algoritmo `Window 4` também apresentaria uma curva em forma de parábola. Assim como para os outros dois tamanhos de janela testados, a queda de desempenho se daria em função da realização de um número cada vez maior de combinações de i-*traces* com w-*traces*.

O gráfico de utilização de memória da Figura 3.17(b) mostra que os algoritmos `Random` e `Stack` possuem pequena variação no uso médio de memória. Dentro do intervalo testado, o algoritmo `Stack` se mantém o tempo todo oscilando em torno de 50 Megabytes de memória. O algoritmo `Random` apresenta inicialmente valores maiores de memória, chegando a aproximadamente 130 Megabytes para $n = 40$. Este valor decresce até aproximadamente $n = 60$, momento em que o comportamento de uso de memória se torna semelhante ao exibido pelo algoritmo `Stack`.

Os algoritmos de janela deslizante apresentam um consumo médio de memória maior que os algoritmos `Random` e `Stack`. Enquanto o algoritmo `Window 4` tem um comportamento mais estável, os algoritmos `Window 5` e `Window 6` apresentam um consumo de memória crescente sendo que o segundo apresenta uma taxa de aumento maior.

O gráfico da Figura 3.17(b) exibe uma vantagem do algoritmo `Stack` em relação aos demais algoritmos. Como ele é um algoritmo não aleatório, sabemos que ele não produzirá um *trace* mais de uma vez. Assim, não precisamos manter os *traces* produzidos em memória para verificação de duplicatas.

No caso do algoritmo `Random`, o maior consumo de memória coincide com a fase em que ele mais enumera *traces*. A partir do momento que o número de *traces* decresce, a quantidade de memória para armazená-los diminui e o consumo médio de memória se estabiliza num nível necessário a manutenção dos objetos que estão sendo utilizados para produzir a enumeração.

Além de manter o conjunto de *traces* calculadas em memória, os algoritmos `Window 4`, `Window 5` e `Window 6` também armazenam diversos conjuntos de i-*traces* que serão combinados com w-*traces*. Por exemplo, quando a janela está no nível i, o algoritmo enumerará w-*traces* que serão combinados com i-*traces* para comporem os $(i + w)$-*traces* que estarão associados ao nível $(i + w)$. Simultaneamente, o algoritmo de janela deslizante está armazenando em memória todos os j-*traces*, onde

$(i+1) < j < (i+w)$, que serão combinados com w-*traces* em passos futuros.

Para reduzir o consumo de memória dos algoritmos `Random`, `Window 4`, `Window 5` e `Window 6` podemos evitar de manter o conjunto de *traces* calculados em memória, escrevendo os resultados assim que obtidos. Contudo, um passo adicional para verificação/remoção de duplicatas deverá ser executado posteriormente.

Quando realizamos uma enumeração parcial de um conjunto mais amplo de dados, devemos observar se o resultado produzido não é enviesado. Em outras palavras, devemos verificar que a saída do algoritmo contém *traces* que cobrem todo o espaço de soluções da maneira mais uniforme possível.

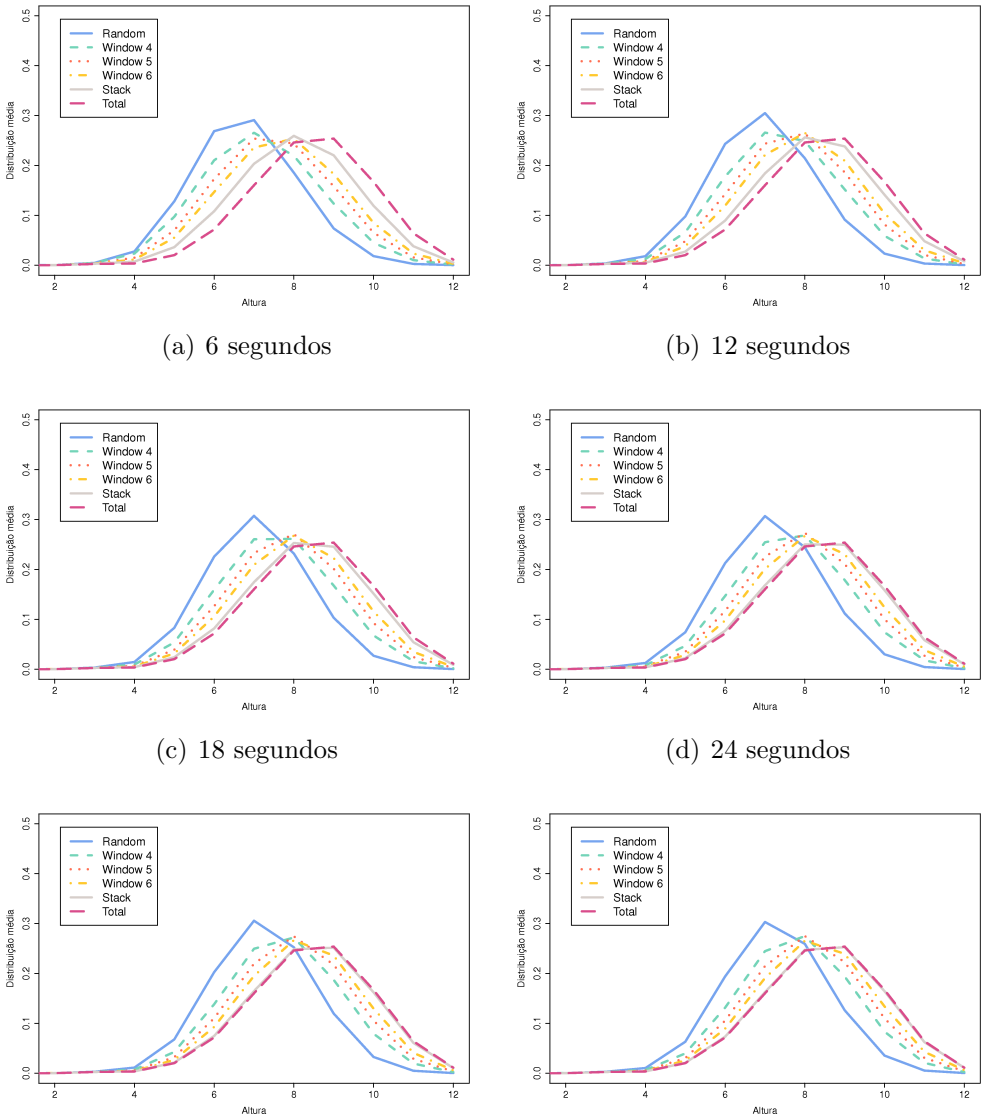
Para verificar esta característica, optamos por realizar um estudo da distribuição dos *traces* conforme as suas alturas e tamanhos médios de reversões no conjunto completo de *traces* e comparar com as distribuições observadas nas saídas produzidas pelos algoritmos propostos.

Assim, voltando ao conjunto de 500 permutações aleatórias com $n = 15$ e $d(\pi) = 12$, analisamos o conjunto completo de *traces* de cada uma das permutações. Novamente, definimos intervalos $(x - 1/2, x + 1/2]$ com $2 \leq x \leq 12$, no caso dos dados de altura, ou $2 \leq x \leq 11$, no caso dos dados de tamanho médio de reversão. De posse de um conjunto de *traces* de uma dada permutação, contamos o número de *traces* que eram englobados por cada intervalo. Feito isso, a porcentagem de *traces* contidos dentro do intervalo foi calculada através da divisão do valor obtido na contagem pelo tamanho do conjunto de *traces*. Finalmente, para cada intervalo, calculamos o valor médio da porcentagem observada em todas as 500 permutações.

O procedimento descrito acima foi executado para o conjunto de *traces* completos (`Total`) e para as soluções dos algoritmos `Random`, `Stack`, `Window 4`, `Window 5` e `Window 6` obtidas com os diferentes tempos de execução que foram utilizados. Os gráficos das Figura 3.18 e 3.19 mostram a evolução da distribuição dos *traces* segundo a altura ou o tamanho médio de reversão conforme o tempo limite de execução muda.

A análise dos gráficos das Figuras 3.18 e 3.19 mostram que, em geral, quanto maior o tempo fornecido para execução, maior será a proximidade da distribuição de *traces* produzidos em relação à distribuição observada no conjunto completo de *traces*.

Com maior ou menor grau, todos os algoritmos propostos tendem a se aproximar da curva `Total`. Como o algoritmo `Random` é o que menos enumera, ele é o que mais lentamente se aproxima da curva de referência. Dentro dos algoritmos de janela,



(a) 6 segundos

(b) 12 segundos

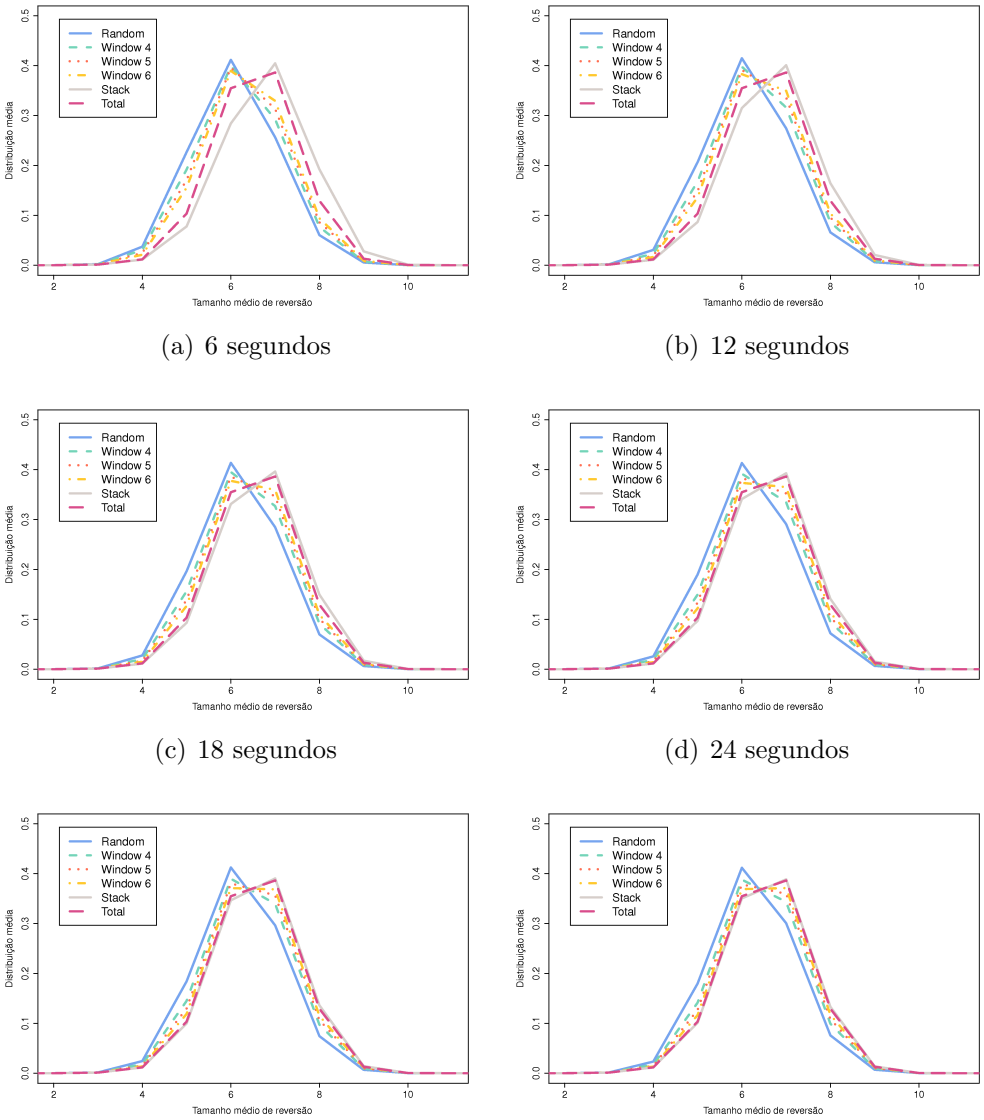
(c) 18 segundos

(d) 24 segundos

(e) 30 segundos

(f) 36 segundos

Figura 3.18: Distribuição média dos *traces* de acordo com a altura. Um total de 500 permutações aleatórias com $n = 15$ e $d(\pi) = 12$ foram consideradas. As curvas mostram a distribuição obtida no conjunto completo de *traces* das permutações (`Total`) e nos conjuntos produzidos pelos algoritmos `Random`, `Window 4`, `Window 5`, `Window 6` e `Stack` conforme o tempo limite de execução.



(a) 6 segundos (b) 12 segundos

(c) 18 segundos (d) 24 segundos

(e) 30 segundos (f) 36 segundos

Figura 3.19: Distribuição média dos *traces* de acordo com o tamanho médio das reversões. Um total de 500 permutações aleatórias com $n = 15$ e $d(\pi) = 12$ foram consideradas. As curvas mostram a distribuição obtida no conjunto completo de *traces* das permutações (`Total`) e nos conjuntos produzidos pelos algoritmos `Random`, `Window 4`, `Window 5`, `Window 6` e `Stack` conforme o tempo limite de execução.

`Window 6` é o que apresenta melhor aproximação.

No geral, o algoritmo `Stack` foi o que apresentou os melhores resultados. Como estamos trabalhando com um conjunto de permutações de pequeno porte, ele é capaz de enumerar a grande maioria dos *traces* e, por isso, é o que mais se aproxima da curva de distribuição real.

Os gráficos mostram que o algoritmo `Random` é o que possui curvas que mais se afastam da curva esperada. Isso se deve ao fato deste algoritmo privilegiar a geração de *traces* que representam um número maior de soluções. Os demais algoritmos não possuem esse problema.

Acreditamos que, com exceção do algoritmo `Random`, os algoritmos propostos tendem a produzir conjuntos de *traces* não enviesados. Caso isso não fosse verdade, as curvas dos algoritmos não apresentariam tendência de se aproximar da curva de referência. De maneira geral, quanto maior o tempo de execução, maior o número de *traces* e melhor é a aproximação da curva do algoritmo em relação à curva observada no conjunto completo de *traces*.

3.3 Conclusão

A enumeração parcial de *traces* se mostra uma alternativa viável para a produção de um conjunto de cenários alternativos para permutações que necessitam de um tempo muito grande para enumeração de todos os *traces*.

O algoritmo `Stack` se mostra útil apenas para permutações pequenas. Com o crescimento do tamanho das permutações e da distância de reversão, o número de ramos mortos aumenta bastante. Devido a este fator, o algoritmo `Stack` tem a sua capacidade de produção de *traces* reduzida já que a maior parte do tempo de processamento é destinada a exploração de ramos que resultarão em *trace* nenhum.

O algoritmo `Random` apresenta performance similar a do algoritmo `Stack` em relação ao número de *traces* produzidos. Por ser aleatório, ele tem a vantagem de produzir um novo conjunto de *traces* a cada execução enquanto o algoritmo `Stack` sempre produzirá o mesmo conjunto.

De modo geral, o algoritmo de janela deslizante apresenta capacidade de produzir um número maior de *traces*. Contudo, o tamanho da janela deve ser escolhido de acordo com o tamanho da permutação que será processada. A disponibilidade de tempo e de memória também são itens que devem ser observados.

Os testes mostraram que janelas maiores tem uma tendência de produzir um número maior de *traces* mais rapidamente. Contudo, devido ao aumento no número de operações de combinação de *i-traces* com *w-traces*, a capacidade de enumeração tende a cair com o aumento do tamanho das permutações.

O algoritmo `Random` apresenta uma vantagem interessante em relação aos demais. Como ele não trabalha com o algoritmo de enumeração de *traces* que utiliza pilha, ele pode ser facilmente adaptado para a utilização de restrições biológicas. Ao invés de realizar a seleção de uma reversão aleatória entre todas as reversões da lista de *optimal* 1-*sequences*, a seleção seria feita apenas sobre reversões que respeitam as restrições estabelecidas.

Os resultados do estudo que realizamos até aqui foram apresentados em formato de pôster na 14^{th} *International Conference on Research in Computational Molecular Biology* (`RECOMB 2010`) sob o título "Partial enumeration of traces of solutions for the problem of sorting by signed reversals" [15].

3.3.1 Trabalhos futuros

Este estudo apresenta muitas possibilidades de trabalhos futuros. De fato, a proposição de algoritmos que realizam a enumeração parcial de *traces* é apenas o primeiro passo.

O principal objetivo desta linha de pesquisa é oferecer cenários evolutivos alternativos através da enumeração de *traces*. Contudo, apenas enumerar não resulta em grande ajuda aos pesquisadores.

Uma idéia para tornar a enumeração de *traces* mais útil, que já foi adotada anteriormente por Braga *et al.*, é a aplicação de restrições biológicas durante a enumeração dos *traces* [29, 31]. Novas restrições podem ser implementadas e adaptadas para funcionarem dentro do esquema de enumeração parcial de *traces*.

Outra alternativa ainda mais interessante é a adoção de um sistema de classificação de *traces*. Através de critérios pré-estabelecidos, os *traces* produzidos receberiam pontuações que indicariam com maior grau de confiança a chance de um dado *trace* ser relevante do ponto de vista biológico.

Parte II

Breakpoints

Capítulo 4

Pontos de quebra no genoma

Na Parte I trabalhamos com o problema de ordenação de permutações orientadas através da utilização de reversões. Assim, o tempo todo nós trabalhamos com representações de genoma em um único cromossomo. Contudo, como apresentado no Capítulo 1, existem diversos eventos de rearranjo que podem afetar mais de um cromossomo.

Seja afetando um ou dois cromossomos, nosso objetivo aqui é estudar as regiões onde ocorreram as quebras que deram origem aos eventos. Tais regiões são comumente conhecidas como "Pontos de Quebra" ou, em inglês, *Breakpoints*.

Assim, neste capítulo apresentaremos os conceitos básicos ligado ao estudo destas regiões.

4.1 Rearranjos evolutivos

Rearranjos evolutivos são aqueles que diferenciam os genomas de espécies distintas. Logo, este tipo de rearranjo é diferente do observado em células somáticas como, por exemplo, as mutações que geram as células cancerosas.

Os rearranjos evolutivos são produzidos nas células da linhagem germinal e, por causa disso, eles são transferidos aos descendentes dos organismos onde eles ocorreram. Eles são fixos em toda a população da espécie em função dos mecanismos de seleção natural. Assim, eles se diferenciam dos rearranjos polimórficos que não são, ainda, fixados na população.

Como os mecanismos de seleção natural atuam na fixação dos rearranjos evoluti-

vos, não podemos observar mutações que sejam fortemente desfavoráveis ao indivíduo e à espécie. Assim, esta característica faz com que estes rearranjos se diferenciem daqueles que provocam desordem nos genomas.

Os rearranjos evolutivos são numerosos e desde a descoberta, em 1921, de uma inversão no cromossomo de uma *Drosophila*, o número de rearranjos identificados não pára de crescer. À medida que as técnicas de detecção evoluem, a resolução aumenta permitindo que rearranjos cada vez menores sejam detectados. Por exemplo, nos anos 80 acreditávamos que a diferença entre os genomas do homem e do chimpazé era coberta por 9 inversões e 1 fusão [171]. Contudo, até hoje foram detectadas 93 inversões suplementares com tamanhos entre 12 Kbp e 1 Mbp [117].

Entre as diferentes espécies podemos observar uma grande diversidade de cariótipos. Por exemplo, se observamos os mamíferos, os números de cromossomos do Muntiaco Indiano (*Muntiacus muntjak*) e do Rato Vermelho de Viscacha (*Tympanoctomys barrerae*) são, respectivamente, $2n = 6$ e $2n = 102$. Mesmo quando lidamos com espécies mais próximas, podemos nos surpreender ao ver que espécies fenotipicamente próximas como o Muntiaco Indiano e o Muntiaco Chinês (*Muntiacus reevesi*) possuem $2n = 6$ e $2n = 42$ cromossomos, respectivamente [66].

Os rearranjos cromossômicos possuem um importante papel na evolução dos genomas. No entanto, ainda são desconhecidas informações como a frequência de ocorrência e localização no genoma. Não sabemos explicar a razão das diferenças entre frequências e tipos de rearranjos observados em diferentes grupos de espécies. Por exemplo, enquanto o grupo dos primatas apresenta tendência de evoluir através de inversões, o grupo dos lêmures mostra preferência pelas translocações [59]. Em outro exemplo, temos os peixes e aves que possuem taxas de rearranjo muito menores do que a dos mamíferos e até mesmo menores do que a dos insetos [44].

Até os anos 2000, a distribuição dos rearranjos era considerada aleatória e o modelo de quebras aleatórias era utilizado para explicar esta crença. Contudo, com o aumento de informação promovida pelo crescimento do número de genomas completos disponíveis, este modelo passou a ser questionado.

4.1.1 Modelo de quebras aleatórias

Em 1984, Nadeau e Taylor propuseram que os rearranjos observados entre o homem (*Homo sapiens*) e o camundongo (*Mus musculus*) seguem o modelo de quebras aleatórias (*Random Breakage Model*) introduzido por Ohno em 1973 [114].

Através da análise da distribuição de 83 marcadores homólogos sobre os cromossomos do homem e do camundongo, Nadeau e Taylor obtiveram 46 segmentos conservados com ao menos dois marcadores que estavam na mesma ordem nos dois genomas.

Segundo o modelo de quebras aleatórias, a distribuição dos tamanhos dos segmentos conservados segue uma lei exponencial. Nadeau e Taylor mostraram que os segmentos identificados efetivamente obedeciam uma lei exponencial, confirmando o modelo aleatório. Neste trabalho, eles também estimaram o número de rearranjos existentes entre as duas espécies: 178 ± 38 rearranjos.

À medida que mais marcadores homólogos são identificados nos genomas, o número de segmentos conservados detectados entre o homem e o camundongo aumenta. No entanto, os comprimentos dos segmentos continuam a respeitar as predições do modelo de quebras aleatórias. Em 1996, 1416 marcadores definiam 181 segmentos conservados e atualmente, acreditamos que a maior parte deles já foi identificada [50]. O fato de o modelo continuar válido com o aumento da densidade de marcadores reforça a hipótese de um processo aleatório de distribuição dos rearranjos [113]. Porém, para completar o cenário, alguns segmentos pequenos ainda precisam ser identificados e métodos com melhor resolução devem ser desenvolvidos para validar a hipótese.

4.1.2 Modelo de regiões frágeis

O sequenciamento dos genomas do homem e do camundongo permitiu o desenvolvimento de métodos para alinhamento de genomas completos. Tais métodos permitiram a identificação de segmentos conservados em uma resolução muito maior [94,131].

Quando olhamos os dados obtidos com uma maior escala, observamos que os resultados são compatíveis com o modelo de quebras aleatórias, com 281 segmentos conservados com mais de 1 Mbp identificados por Pevzner e Tesler [131] e 344 com mais de 100 kbp identificados por Kent *et al.* [94].

Por outro lado, o número de pequenos segmentos conservados é muito maior do que o esperado e não pode ser explicado pelo modelo de quebras aleatórias [94]. Pevzner e Tesler detectaram mais de 3000 micro-rearranjos no interior dos segmentos conservados que representam os macro-rearranjos ($>$ 1 Mbp) [131]. Além disso, a distribuição destes pequenos segmentos não é uniforme ao longo do genoma.

A partir dos 281 segmentos conservados maiores do que 1 Mbp observados entre

o homem e o camundongo, Pevzner e Tesler conduziram um estudo para determinar o número mínimo de macro-rearranjos (inversões, translocações, fusões e fissões) necessários para transformar um genoma no outro. Eles obtiveram um limite inferior de 245 rearranjos. Tal número é grande se comparado ao número de pontos de quebra (ou *breakpoints*), que é igual a 258.

Um rearranjo como uma inversão produz geralmente dois pontos de quebra na sequência do cromossomo. Assim, para se obter somente 258 *breakpoints* com pelo menos 245 eventos de rearranjo, alguns pontos de quebra precisam ser utilizados mais de uma vez. Em média, cada *breakpoint* foi utilizado 1,9 vezes. Baseado nestas observações, Pevzner e Tesler propõem o modelo de regiões frágeis (*Fragile Breakage Model*) que determina que algumas regiões do genoma são mais frágeis que outras e, por isso, são mais susceptíveis aos eventos de rearranjo [132].

Este novo modelo gerou uma grande polêmica entre os seus defensores e os defensores do modelo de quebras aleatórias [2, 126, 132, 140, 143, 157]. A polêmica está principalmente ligada à taxa de reutilização.

Sankoff e outros pesquisadores, defensores do modelo de quebras aleatórias, argumentam que a taxa de reutilização não reflete uma propriedade biológica da evolução dos genomas e que ela é consequência de artefatos produzidos pelo método. A eliminação dos pequenos blocos (segmentos menores do que 1 Mbp) pode produzir uma forte taxa de reutilização [143, 157]. Além disso, a perda de sinais indicando rearranjos de segmentos conservados aleatórios pode ser outro responsável pela alta taxa de reutilização [140].

O modelo das regiões frágeis, por outro lado, é suportado por uma série de indícios. Muitas análises mostram a existência de reutilização de pontos de quebra ao longo da evolução. Regiões presentes no genoma humano, quando comparadas com genomas de diversas espécies de mamíferos são identificadas como "quebradas" diversas vezes em linhagens diferentes [68, 83, 112, 170]. Além disso, algumas regiões associadas a rearranjos que provocam desordem genômica ou a certos cânceres são co-localizadas com rearranjos evolutivos [48, 112].

Contudo, estes resultados dependem fortemente da resolução dos pontos de quebra comparados. Por exemplo, o estudo de Ma *et al.* sobre os rearranjos evolutivos dos mamíferos aponta para uma taxa de 8% de *breakpoints* reutilizados ao longo da evolução [109]. O estudo de Murphy *et al.*, por sua vez, indica uma taxa próxima de 20% porém considerando pontos de quebra com tamanhos na ordem de 1 Mbp [112]. Estas diferenças salientam o fato de que não sabemos, até agora, o tamanho das

regiões frágeis [92].

As observações recentes indicam que o genoma pode possuir regiões mais frágeis, susceptíveis a eventos de rearranjo, e regiões mais sólidas, que dificilmente se quebram. Contudo, devemos observar que estamos observando o resultado da seleção natural. Isso pode indicar que a fragilidade ou solidez aparentes das regiões não são reflexos dos diferentes níveis de susceptibilidade a eventos de rearranjo apresentados pela molécula de DNA, mas sim consequências de diferentes pressões do meio-ambiente.

Outra hipótese que pode explicar a reutilização dos *breakpoints* pode estar relacionada à "segurança" que algumas regiões oferecem ao processo de evolução das espécies. Os pontos de quebra relacionados a eventos de rearranjos evolutivos evidenciam alterações no genoma que não resultaram na morte do indivíduo. Assim, do ponto de vista evolucionário, os *breakpoints* envolvidos nestes rearranjos são "seguros" e, por isso, podem ser reutilizados. Segundo esta hipótese, a existência de regiões frágeis não é necessária. Os eventos de rearranjo ocorrem de maneira aleatória ao longo do tempo. Contudo, apenas os eventos "seguros" são mantidos pelos mecanismos de seleção natural.

4.2 Detecção de rearranjos através da comparação de genomas

A identificação de pontos de quebra no genoma das espécies depende da detecção dos eventos de rearranjo nos genomas. Esta etapa é de extrema importância pois ela afeta a sequência das análises.

Os únicos dados disponíveis para a identificação dos rearranjos evolutivos e seus *breakpoints* são as informações da organização dos genomas das espécies atuais. Estes genomas provêem de um ancestral comum e divergiram através do acúmulo de mutações pontuais e de rearranjos evolutivos.

Assim, a abordagem mais apropriada é a análise comparativa de genomas. Através da comparação dos genomas, procuramos identificar quais são as regiões que não sofreram rearranjos. Denominadas de *regiões conservadas*, elas são regiões contíguas, uma em cada um dos genomas comparados, que são situadas em uma mesma região do genoma do ancestral comum às espécies estudadas e que não sofreram rearranjos depois de suas divergências. Dizemos que estas regiões são *homólogas*.

Através da análise das regiões conservadas, podemos definir os eventos de rearranjo que ocorreram ao longo da evolução e, assim, definir a localização dos pontos de quebra nos genomas.

Os métodos de identificação de regiões conservadas entre diversos genomas são variados e nesta seção apresentaremos uma visão geral destas técnicas.

4.2.1 Métodos experimentais

Antes do advento das técnicas de sequenciamento de genomas completos, que permitiram a adoção de estratégias computacionais, os métodos de identificação de rearranjo baseavam-se exclusivamente em experimentos laboratoriais. Entre estes métodos podemos citar a comparação de cariótipos, a hibridação *in situ* (`FISH`), a hibridação comparada (`CGH`) e o sequenciamento de extremidades pareadas (`PEM`).

Comparação de cariótipos

Um cariótipo é uma imagem do núcleo da célula em metáfase, momento em que os cromossomos estão em uma configuração extremamente compacta. Nesta imagem podemos visualizar os diferentes cromossomos e identificá-los por tamanho.

A comparação do número de cromossomos e das diferenças de tamanho em genomas distintos fornecia dados sobre a evolução das espécies.

Na década de 1970, uma técnica denominada *Chromossome banding* introduziu uma maior resolução ao estudo dos diferentes cariótipos. Diferenças no nível de compactação e na composição de nucleotídeos das sequências de DNA provocam diferenças na coloração dos segmentos que compõem os cromossomos. Estas regiões de diferente coloração são denominadas bandas e a análise dos padrões que elas formam no cromossomo permitem caracterizá-lo.

Como o comprimento médio de uma banda é aproximadamente 4 Mbp, apenas os grandes rearranjos podem ser identificados. Além disso, se as espécies comparadas são muito distantes entre si, o padrão de bandas pode ser de difícil análise. Finalmente, as bandas indicam apenas características estruturais similares e não garantem a homologia da sequência.

Hibridação *in situ* – FISH

A técnica *Fluorescent In Situ Hybridization* (FISH) apareceu na década de 1990 e permitiu a análise de homologia das sequências de DNA.

Baseada no princípio de hibridação, processo em que uma fita simples de DNA se une a uma fita complementar para formar uma fita dupla, esta técnica trabalha com a localização de sondas ao longo dos cromossomos.

As sondas são pequenas moléculas de fita única de DNA marcadas com uma substância fluorescente. Elas se ligam aos cromossomos em posições em que elas representam a sequência complementar. A análise das posições que apresentam fluorescência permite determinar regiões homólogas entre o cromossomo analisado e o cromossomo que deu origem às sondas.

Conforme a compactação do DNA a resolução pode variar de 50 Kbp para os cromossomos em interfase a 3 Mbp para os cromossomos em metáfase.

Este método, também denominado de pintura cromossômica, foi estendido para que pudesse ser aplicado em espécies mais distantes (zoo-FISH). Contudo, o seu principal inconveniente é a incapacidade de detectar rearranjos intra-cromossômicos.

Hibridação comparada – CGH

A técnica *Comparative Genomic Hybridization* (CGH) é inspirada na técnica FISH e permite a identificação de rearranjos que alteram a quantidade de DNA presente no cromossomo como, por exemplo, as duplicações e deleções, mas não é capaz de identificar rearranjos como inversões e translocações.

Esta técnica consiste na hibridação de duas soluções de DNA, uma de referência e outra de interesse, marcadas com fluorocromos diferentes, sobre uma preparação de cromossomos em metáfase. Através da comparação dos níveis relativos de fluorescência, podemos identificar ganhos e perdas de DNA ao longo dos cromossomos.

Embora limitada em resolução como a técnica FISH, esta técnica se desenvolveu com a introdução dos chips de DNA. No lugar de cromossomos em metáfase, a hibridação é realizada sobre um chip contendo milhares de clones ou sondas de DNA que cobrem todo o genoma de referência. Contudo, a análise dos chips não é trivial e necessita de métodos de segmentação de sinal.

Sequenciamento de extremidades pareadas – PEM

A técnica *Paired-End Mapping* (PEM) é uma técnica que permite a identificação de quase todos os tipos de rearranjos em grande escala e com alta resolução. Ela consiste na fragmentação do genoma de interesse de maneira aleatória em clones de tamanho constante e no posterior sequenciamento sistemático das duas extremidades de cada clone. As sequências obtidas são mapeadas, através do alinhamento de sequências, sobre o genoma de referência. Os rearranjos são determinados quando a orientação ou a distância das extremidades de um clone são diferentes da esperada.

Esta técnica necessita que o genoma de referência esteja totalmente sequenciado. Além disso, como as extremidades de sequências são normalmente curtas (aproximadamente 500 bases), os genomas comparados não podem ser muito distantes em termos de sequência para que o mapeamento possa ser feito sem ambiguidade.

A combinação desta técnica com as novas tecnologias de sequenciamento em larga escala aumentaram consideravelmente a resolução e, ao mesmo tempo, reduziram o custo [97]. Esta técnica também é utilizada para a detecção de aberrações cromossômicas em células cancerosas [12].

4.2.2 Comparação da ordem dos genes

Com o crescimento da quantidade de informação disponível referente ao mapeamento dos genomas, métodos bioinformáticos começaram a ser desenvolvidos com o objetivo de se detectar regiões conservadas. Estes métodos identificam as regiões conservadas através da comparação das posições relativas dos genes nos diferentes genomas.

Genes ortólogos são genes pertencentes a duas ou mais espécies diferentes e são semelhantes por terem origem em um mesmo ancestral comum. Como a ordem e a orientação dos genes ortólogos tendem a se manter conservadas nas espécies, a hipótese mais parcimoniosa indica que eles tem origem em uma mesma região no ancestral comum e que eles não sofreram qualquer rearranjo após a divergência das espécies.

De posse da informação de posição física ou relativa sobre os genomas comparados e da relação de ortologia existente entre os marcadores, qualquer tipo de marcador ortólogo pode ser utilizado. Normalmente, estes marcadores são genes pois eles são os tipos de dados mais mapeados e sequenciados.

Identificação do posicionamento dos genes

A determinação da posição dos genes nos genomas pode ser feita de maneira experimental ou através de métodos de anotação de sequências.

Nos métodos experimentais, técnicas como a produção de mapas genéticos ou de mapas de irradiação podem ser utilizadas. Contudo, a resolução destes mapas é fraca (50 a 100 kbp para os mapas de irradiação e 1000 kbp para os mapas genéticos) e a informação da orientação dos genes não é contemplada. Assim, tais mapas permitem a possibilidade de se estudar a ordem dos genes em organismos que ainda não foram sequenciados [112, 159].

O processo de anotação consiste na associação de informações biológicas às sequências. Primeiro devemos identificar os elementos funcionais presentes no genoma e, então, associar uma função biológica a eles. Para a primeira etapa, o objetivo é identificar os genes protéicos com a utilização de métodos de predição de genes. Existem três categorias de métodos de predição de genes:

- métodos *ab initio* que utilizam apenas a informação da sequência para realização da predição dos genes. Estes métodos se baseiam no conhecimento *a priori* da estrutura dos genes (código genético, tamanho dos genes, introns e exons, promotores, etc).

- métodos que utilizam sequências de mRNA ou de proteínas. Tais sequências são obtidas através de técnicas de sequenciamento e são posteriormente posicionadas no genoma através da realização de alinhamentos de sequências (mapeamento em genomas de referência).

- métodos comparativos que utilizam anotações e informações obtidas para genomas de espécies próximas. Nestes métodos, sequências de genes provenientes de outras espécies são utilizadas para busca de similaridade ao longo do genoma que está sendo estudado (anotação automática).

Estes métodos são complementares e são frequentemente combinados. No entanto, os resultados apresentam tendência a super-estimação do número de genes já que pseudo-genes são muitas vezes identificados como genes reais. Devido a isso, as predições necessitam de verificação experimental.

Atribuição de ortologia

A *Homologia* é a característica que indica que duas sequências diferentes derivam de uma mesma sequência ancestral. As sequências homólogas podem ser divididas em duas categorias: *sequências parálogas* e *sequências ortólogas*.

As sequências parálogas são aquelas que divergiram após um evento de duplicação. As sequências ortólogas são aquelas que possuem um mesmo ancestral comum e que divergiram após um evento de especiação. Dizemos que dois genes são ortólogos se eles derivaram de um único gene ancestral no último ancestral comum às duas espécies que os possuem.

Para ilustrar estes conceitos, podemos utilizar o exemplo do gene da insulina, cuja árvore filogenética é representada pela Figura 4.1. No ancestral do homem (*Homo sapiens*), do rato (*Rattus norvegicus*) e do camundongo (*Mus musculus*) podíamos encontrar um gene `INS` ancestral responsável pela produção da insulina. Após a especiação do ancestral comum que deu origem ao ramo onde se encontra o homem e ao ramo onde se encontram os roedores, o gene `INS` sofreu uma duplicação gerando os genes `INS1` e `INS2`. Esse evento ocorreu antes da especiação do ancestral comum ao rato e ao camundongo. Assim, o homem possui uma cópia do gene da insulina e o rato e o camundongo possuem duas cópias.

O gene `INS` presente no homem é ortólogo aos genes `INS1` e `INS2` presentes nos roedores. O gene `INS1` do rato é ortólogo ao gene `INS1` e parálogo ao gene `INS2` presentes no camundongo.

As relações de ortologia podem ser múltiplas. Isso significa que n genes de uma espécie podem ser ortólogos a m genes de outra espécie. Para isso, basta que as duplicações que originaram as n cópias de uma espécie e m cópias da outra tenham ocorrido após o evento de especiação. As n cópias da primeira espécie ou as m cópias da segunda espécie são denominadas *co-ortólogas* ou *in-parálogas*.

Os *genes ortólogos de posição* são genes que, além de serem ortólogos, mantiveram o contexto genômico (vizinhança) do gene ancestral. A Figura 4.2 exibe o processo de duplicação do gene da insulina no ramo dos roedores. Após a duplicação, podemos ver que a fonte (`INS1`) manteve o contexto genômico do gene ancestral `INS` enquanto o alvo (`INS2`) foi inserido em uma região diferente do genoma. Assim, os genes `INS` encontrado no homem e `INS1` encontrado no rato e no camundongo são ortólogos de posição. O conceito de ortologia de posição é definido nos trabalhos de Bourque *et al.*, 2005 [26], Burgetz *et al.*, 2006 [26] e Swidan *et al.*, 2006 [154] para identificar

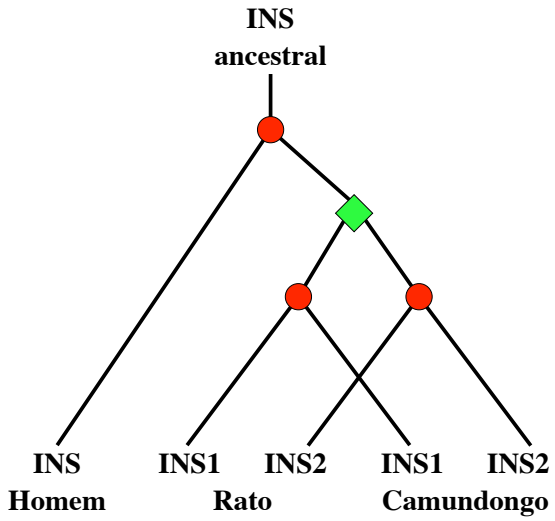


Figura 4.1: Árvore filogenética do gene da insulina encontrados no homem, no rato e no camundongo. Os círculos vermelhos indicam um processo de especiação e o losango verde indica um processo de duplicação.

genes ortólogos que mantém o mesmo contexto genômico.

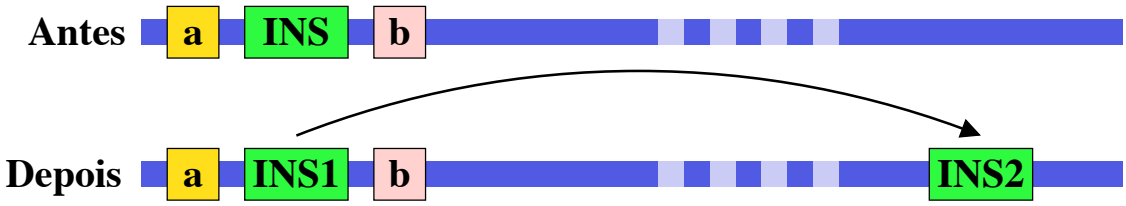


Figura 4.2: Esquema da duplicação do gene `INS` da insulina. O gene `INS1` é a fonte da duplicação e seu contexto (regiões vizinhas representadas por `a` e `b`) permanecem inalteradas. O gene `INS2` é o alvo e se localiza em um contexto diferente.

Dewey e Patcher propõem dois tipos de ortólogos de posição segundo o tipo de duplicação [52]. Uma duplicação dirigida é aquela que permite a identificação da fonte e do alvo. Por exemplo, em uma duplicação em *tandem*, não podemos identificar qual das cópias é a fonte e qual é o alvo pois ambas estão situadas no mesmo contexto genômico. Assim, Dewey e Patcher definem os *topo-ortógolos* como aqueles que após a divergência, nem a fonte nem o alvo sofreram uma duplicação dirigida. Os autores também definem os *monotopo-ortólogos*, que são topo-ortólogos

que depois da divergência não sofreram duplicações não dirigidas. Os monotopo-ortólogos são os únicos que definem relações 1-1.

Normalmente, as duplicações não são consideradas no estudo de rearranjos. A principal justificativa está no fato de que estes eventos normalmente atingem regiões limitadas do genoma. Apesar de existirem exceções como as duplicações segmentares observadas nos primatas ou mesmo as duplicações de genomas completos, a tendência é que estes eventos sejam estudados separadamente de outros tipos de rearranjo. Portanto, como procuramos analisar as modificações de ordens nos genes provocadas por rearranjos equilibrados (que não alteram a quantidade de DNA do genoma) como as inversões, translocações, fusões e fissões, a utilização de marcadores ortólogos de posição se faz necessária.

A detecção de homologia entre as sequências é feita normalmente segundo um critério de similaridade de sequência. Como duas sequências homólogas tem origem em uma mesma sequência, se o tempo evolutivo não é muito longo, esperamos que elas sejam similares.

Frequentemente utilizamos o alinhamento de sequências para avaliar a similaridade entre duas sequências. Através do alinhamento podemos pontuar os nucleotídeos similares e penalizar as mutações tais como inserções, deleções ou substituições e dessa forma obter uma pontuação de similaridade. Para determinar a homologia das sequências, podemos definir uma pontuação de similaridade mínima. Qualquer alinhamento que apresentar pontuação maior ou igual ao valor limite estipulado definirá as sequências como sendo homólogas.

A escolha da pontuação mínima não é uma tarefa trivial. No entanto, definir ortólogos ou parálogos é uma atividade ainda mais complexa e diversos métodos podem ser utilizados.

Melhor* hit *recíproco A maneira mais simples de atribuir informação de ortologia é através da utilização do método do melhor *hit* recíproco (*Reciprocal Best Hit* – `RBH` ou *Bidirectional Best Hit* – `BBH`). Se o gene a_i do genoma A possui o gene b_j do genoma B como aquele que apresentou melhor pontuação de alinhamento entre todas os genes do genoma B e, reciprocamente, o gene b_j possui o gene a_i como aquele que mostrou melhor pontuação entre todos os genes do genoma A, então a_i e b_j são melhores *hits* recíprocos. Este método serve para identificação de relações 1-1.

Na prática, o processo consiste na execução de todos os alinhamentos dois a dois

de genes dos genomas A e B. A cada gene de um genoma, associamos um gene do outro genoma que apresentou a melhor pontuação de similaridade. Por fim, as relações de reciprocidade são detectadas.

Este método tem a vantagem de ser o mais simples e rápido. Porém, ele possui algumas desvantagens. O método só será confiável se a lista completa de genes dos dois genomas é disponível. Além disso, o método só pode ser aplicado simultaneamente a dois genomas. Finalmente, esta técnica pode apenas detectar relações 1-1 e isso diminui a confiabilidade dos resultados devido à existência de relações n-m.

Atribuição de ortologia n***-***m A partir dos ortólogos 1-1 identificados pelo método do melhor *hit* recíproco, uma análise adicional pode ser feita para que as relações n-m sejam obtidas. O método `INPARANOID` executa esta análise buscando dentro de uma espécie os genes mais similares entre eles, independentemente dos genes da outra espécie [119].

Enquanto o método `INPARANOID` se aplica apenas a dois genomas simultaneamente, o método `COG` (*Clusters of Orthologous Groups*) pode ser usado com mais de dois genomas. Em um primeiro passo, ele agrupa os potenciais in-parálogos de uma espécie. Feito isso, um critério similar ao `RBH` é aplicado entre os grupos de in-parálogos de diferentes espécies [156].

O método `OrthoMCL` também pode ser aplicado a mais de dois genomas [105]. Desenvolvido para detectar ortologia e paralogia em genomas eucarióticos, este método possui duas etapas. Na primeira etapa, um grafo representando as relações das sequências é gerado com base em regras biológicas, previamente conhecidas, que ditam como os nós serão conectados e quais devem ser os pesos das arestas. O grafo gerado na primeira etapa é subdivido em subgrafos na segunda etapa através da utilização de um algoritmo `MCL` (*Markov Cluster*).

Abordagem filogenética A abordagem filogenética tende a ser mais confiável. Como seu objetivo é de inferir a história evolutiva de famílias de genes homólogos, ela permite a identificação de relações n-m e de paralogia. A técnica, que pode ser aplicada para um grande número de espécies ao mesmo tempo, se baseia na análise das árvores filogenéticas das espécies e das famílias de genes para detecção de eventos de especiação, duplicação e perdas de genes.

Comparando todos os genes de diferentes genomas dois a dois, podemos agrupá-los em famílias de genes em função da pontuação de similaridade. Em seguida, um

alinhamento múltiplo dos genes de uma mesma família é realizado e uma árvore filogenética é calculada. Entre todos os nós da árvore, queremos identificar os nós que indicam eventos de especiação e de duplicação. Como a topologia da árvore filogenética das espécies é conhecida, ela auxilia a detecção dos nós equivalentes às especiações. Para inferir os eventos de duplicação, o princípio da parcimônia é utilizado de modo a minimizar o número de eventos de duplicação e de perdas de genes [58, 121].

A plataforma Ensembl [85] utiliza uma metodologia baseada na análise de máxima verossimilhança entre árvores filogenéticas. Esta metodologia possui as seguintes etapas [162]:

1. Obtenção da tradução mais longa de cada gene para todas as espécies do banco de dados do Ensembl. Caso o gene possua mais de um *splice* a tradução que resulta na proteína mais longa é utilizada;

2. Utilização do programa `WUBlastp` para alinhar cada gene contra todos os outros genes (de outras espécies e da própria espécie);

3. Construção de um grafo esparso de relações entre genes. Genes são conectados se apresentarem melhor *hit* recíproco ou se a razão da pontuações de `BLAST` é maior do que 0,33 (*BLAST score ratio* [135]);

4. Utilização do programa `Hcluster_sg` [104] para identificação de *clusters* (componentes conexas) no grafo esparso;

5. Construção de um alinhamento múltiplo para cada *cluster* com o programa `MUSCLE` [60];

6. Para cada *cluster*, o alinhamento múltiplo e a árvore filogenética das espécies são fornecidas como entrada do programa `TreeBeST` [104]. Este programa constrói uma nova árvore fazendo reconciliação dos nós internos das árvores e identificação de duplicações.

7. Finalmente, para cada gene, a árvore construída é usada para inferência de relações de ortologia e paralogia.

Com esta metodologia, Ensembl é capaz de identificar diversos tipos de relações: ortólogos 1-1, ortólogos 1-n, ortólogos n-m, parálogos na espécie, parálogos entre espécies, etc.

O maior problema da abordagem filogenética é o custo computacional para executá-la. Os alinhamentos múltiplos podem exigir grande quantidade de tempo para serem realizados em função do número de espécies comparadas. Além disso, o método é sensível a artefatos de reconstrução filogenéticas como, por exemplo, o fenômeno de atração dos ramos longos. Por último, a abordagem filogenética considera que os genes são transmitidos verticalmente (do ancestral para o descendente) e, por isso, ela não é confiável para casos em que houve ocorrência de transferência horizontal (de uma espécie para outra).

Os métodos apresentados não permitem distinguir os ortólogos de posição entre os ortólogos n-m. Por exemplo, no caso do método `RBH`, a escolha cai sobre o par mais similar e não, necessariamente, sobre os ortólogos de posição. Um estudo conduzido sobre genomas de bactérias mostra que, dentre os ortólogos n-m, apenas 60% dos pares de genes mais similares são também ortólogos de posição [118]. Como consequência destes fatores, os ortólogos n-m são frequentemente descartados ou uma análise adicional é feita baseada na informação contextual de outros genes na vizinhança.

Identificação de segmentos conservados

Uma vez que temos, para cada par de genes ortólogos, as informações sobre cromossomos, posições e orientações dos genes nos dois genomas, o próximo passo é a identificação de segmentos conservados: regiões que possuem o mesmo conjunto de genes com mesma ordem e com mesma orientação.

Se nenhuma flexibilidade é permitida, esta tarefa é trivial. Toda vez que temos dois genes consecutivos sobre um genoma e seus ortólogos são consecutivos e possuem orientações concordantes no outro genoma, podemos dizer que eles pertencem a um mesmo bloco.

O problema fica mais complicado conforme aumentamos o grau de flexibilidade. Por exemplo, podemos autorizar que alguns genes não respeitem ordem ou orientação observadas no genoma de referência. Em alguns casos, esta flexibilidade é necessária por causa dos ruídos presentes nos dados de entrada. Erros de atribuição de ortologia ou erros de posição causados por erros de montagem são exemplos de ruídos que podem prejudicar a identificação de segmentos conservados. Em outros casos, a flexibilidade pode se justificar quando queremos evitar a identificação de segmentos muito pequenos. Por exemplo, quando estudamos espécies mais distantes, elas po-

dem apresentar muitos rearranjos. Aumentando a flexibilidade neste caso, estaremos privilegiando os grandes rearranjos.

Alguns métodos, denominados *Gene Teams* ou *Max-Gap Clusters* possuem o objetivo de identificar regiões que possuem conteúdo gênico similar mas não exigem que a ordem entre elas seja conservada. Este tipo de método não é adequados para o estudo de rearranjos pois grandes diferenças podem existir entre os blocos dos dois genomas. Estes tipos de métodos são mais apropriados ao estudo de genes que são funcionalmente associados [20, 84].

Para o estudo de rearranjo de genomas, procuramos blocos onde o conteúdo, a ordem e as orientações dos genes são conservados. Estes blocos conservados podem ser denominados de *blocos de sintenia*. Introduzido em 1971 por John H. Renwick, o termo sintenia designava a existência de dois *loci* sobre um mesmo cromossomo. O métodos de mapeamento da época permitiam a confirmação da localização de dois *loci* sobre um mesmo cromossomo, mas sem a informação de posição ou de distância (entre os *loci*) [124]. Assim, originalmente o termo sintenia se aplicava somente a um único genoma. O termo *sintenia conservada* indica a ocorrência de dois ou mais marcadores sobre um mesmo cromossomo em mais de um genoma [4, 61]. A expressão *segmento conservado* ou mais precisamente *segmento de ordem conservada* define uma região onde as ordem dos marcadores é conservada. Contudo, o termo bloco de sintenia é, atualmente, o mais utilizado na literatura [39, 94, 131, 146].

Numerosos métodos de construção de blocos de sintenia baseados na ordem dos genes ortólogos são disponíveis. Assim, não existe um método mais comumente utilizado pela comunidade. Muitas técnicas são descritas como métodos *ad hoc*, sem uma descrição formal dos objetos que estão sendo estudados e, muitas vezes, não são implementados em um programa que possa ser utilizado por outros projetos. Outra explicação para o grande número de métodos está na natureza dos dados de entrada que conforme a situação podem exigir táticas distintas (por exemplo, genomas de bactérias e de eucariotos possuem estruturas diferentes).

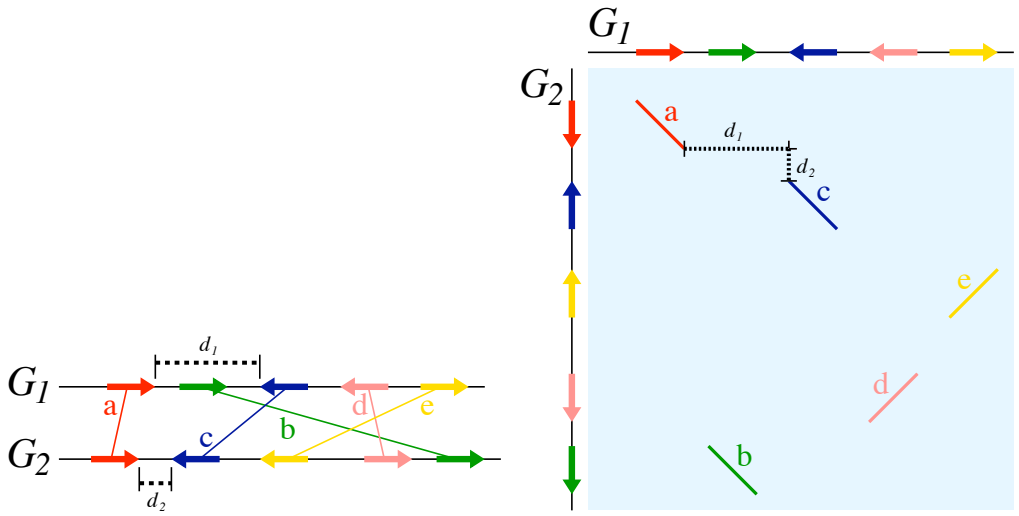
Antes de apresentar os métodos, precisamos definir dois conceitos: âncora e colinearidade.

- **Âncora** Um par de genes ortólogos é denominado de âncora. Uma âncora x é definida pela sua localização inicial em cada um dos dois genomas e pela sua orientação. Mais precisamente $x = (c_1, x_1, c_2, x_2, \sigma_x)$ onde x_1 (x_2) é a posição do gene x no cromossomo c_1 (c_2) do genoma G_1 (G_2). A orientação σ_x tem

o valor +1 se os dois genes estão na mesma orientação nos dois cromossomos. Caso contrário, σ_x recebe o valor -1.

A Figura 4.3 exibe um exemplo de *dotplot*. Com este tipo de gráfico podemos representar em um espaço bidimensional a distribuição das âncoras encontradas entre dois genomas G_1 e G_2. Em um *dotplot* as diagonais decrescentes indicam âncoras que possuem mesma orientação enquanto as diagonais crescentes apontam âncoras que possuem diferentes orientações.

- **Colinearidade** Duas âncoras x e y, com $x_1 < y_1$, são colineares se elas pertencem a um mesmo par de cromossomos e se $x_2 < y_2$ com $\sigma_x = \sigma_y = +1$ ou $x_2 > y_2$ com $\sigma_x = \sigma_y = -1$. No exemplo da Figura 4.3, os pares de âncoras (a, b), (a, c) e (d, e) são colineares.



(a) Genomas G_1 e G_2 (b) *Dotplot* dos genomas G_1 e G_2

Figura 4.3: Exemplo de gráfico *dotplot* de dois genomas G_1 e G_2 que possuem um único cromossomo e 5 âncoras a, b, c, d e e. As distâncias entre as âncoras a e c nos genomas G_1 e G_2 são representadas por d_1 e d_2, respectivamente.

A maior parte dos métodos de construção de blocos de sintenia se baseiam no seguinte esquema: duas âncoras pertencem a um mesmo grupo se elas são colineares e possuem distância entre elas menor ou igual a um valor limite G (*gap*). Além disso, um valor S determina o tamanho mínimo (em número de bases ou número de

genes) dos blocos que serão mantidos para estudo. Na literatura podemos encontrar diversos métodos que seguem esta base: `SynBrowse` [123], `Cinteny` [146], `SyntQL` [26, 172], `GeneSyn` [125], `FISH` [34], `DAGchainer` [71], `DiagHunter` [35], `ADHoRe` [160], `LineUp` [72] e `AutoGraph` [51].

As diferenças entre estes métodos podem ser encontradas nas condições das âncoras na entrada (com duplicação ou não, com sobreposição ou não), na definição das distâncias entre as âncoras, no modo de medir as distâncias (pares de bases ou número de genes), nos critérios de retenção de um bloco, na flexibilidade do critério de colinearidade, etc. Em função das escolhas feitas, as propriedades dos blocos obtidos podem ser muito diferentes.

A maioria dos algoritmos exigem que as relações de ortologia sejam 1-1 e que os genes não apresentem sobreposição em nenhum dos dois genomas [51, 72, 123]. Quando a ortologia n-m é aceita, os algoritmos podem limitar os números n e m ou realizar um pré-processamento para eliminação de duplicatas [34, 146].

`LineUp` e `AutoGraph` são algoritmos adaptados a dados de mapas genéticos que possuem genes não orientados e, por isso, não utilizam informação de orientação das âncoras [51, 72].

`GeneSyn` é um algoritmo que pode ser aplicado a mais de dois genomas simultaneamente [125].

Como exemplificado na Figura 4.3, a distância entre as âncoras é medida em cada um dos genomas. Assim, temos o valor d_1 referente à distância entre os elementos do par de âncoras no genoma G_1 e o valor d_2 ligado ao mesmo par de âncoras no genoma G_2. A distância $d_1(x, y)$ ($d_2(x, y)$) entre duas âncoras x e y no genoma G_1 (G_2) é definida por $d_1(x, y) = |x_1 - y_1|$ ($d_2(x, y) = |x_2 - y_2|$). A distância pode ser medida em números de pares de bases (posição física do gene no cromossomo) ou em número de genes (ordem numérica do gene no cromossomo).

A maneira como as distâncias d_1 e d_2 são combinadas para obtenção da distância global é variável. A distância de Manhattan ($d = d_1 + d_2$) é a mais utilizada. O algoritmo `FISH` e muitos algoritmos de alinhamento de genomas completos a utilizam [34, 86, 131]. Outra maneira de combinar estes valores é utilizando a distância máxima $d = max(d_1, d_2)$ [123, 172]. Alguns métodos consideram a diferença das distâncias d_1 e d_2. Por exemplo, o algoritmo `ADHoRe` utiliza a seguinte distância: $d = 2 \times max(d_1, d_2) - min(d_1, d_2)$ [160].

Para avaliar os blocos construídos, diversos métodos também estão disponíveis. O mais comum é considerar o tamanho do grupo em número de âncoras que ele

possui [34, 72, 160, 172]. Este critério pode ter o objetivo de eliminar os pequenos rearranjos ou eventuais ruídos e erros contidos nos dados. O princípio utilizado é o de que erros, quando existem, são isolados. Assim, quanto maior o número de âncoras, maior o grau de confiabilidade do bloco. O valor S que determina o tamanho mínimo de um bloco pode ser um parâmetro fornecido pelo usuário ou ser um valor estimado estatisticamente. Por exemplo, alguns métodos fazem simulações onde os genomas são aleatorizados (os genes são redistribuídos aleatoriamente sobre o genoma) para avaliação do tamanho dos grupos de genes esperados ao acaso [72,160]. Considerando um modelo nulo, no qual as âncoras são distribuídas aleatoriamente sobre uma matriz $n \times n$ (n é o número de âncoras), o algoritmo `FISH` calcula a probabilidade da existência de um grupo de k-âncoras de maneira analítica [34].

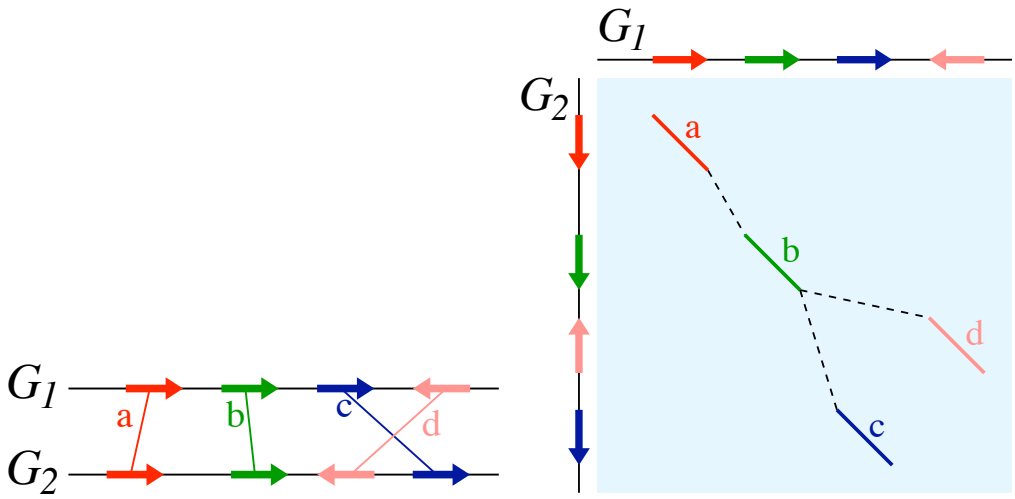
O critério de tamanho dos blocos também pode ser medido em número de pares de bases. Dessa maneira, consideramos a distância coberta por um grupo de âncoras sobre os genomas [146].

`DAGchainer` utiliza um critério mais complexo que considera o grau de similaridade das âncoras, a distância entre as âncoras e a diferença de distância entre os dois genomas [71].

O grau de desordem dentro do bloco também pode ser um critério utilizado para avaliação dos blocos. O algoritmo `ADHoRe` verifica se as âncoras de um grupo formam uma reta no *dotplot* [160]. Ele efetua uma regressão linear em cada grupo com as coordenadas das âncoras e avalia os grupos de acordo com o grau de ajuste dos pontos à reta da regressão.

Os métodos disponíveis na literatura possuem diversas abordagens algorítmicas que nem sempre são descritas precisamente. Muitos métodos utilizam grafos para representar as relações de distância entre as âncoras sobre os diferentes genomas [34, 71, 125]. Programação dinâmica e funções recursivas são utilizadas para busca das melhores cadeias de âncoras dentro de uma matriz ou de um *dotplot* [34, 35, 71]. Alguns métodos realizam processamento de modo iterativo ou recursivo, modificando os valores dos parâmetros a cada etapa, pouco a pouco aumentando o tamanho dos blocos [160].

A flexibilidade na construção de blocos de sintenia possui consequências que nem sempre são consideradas pelos métodos. O encadeamento de âncoras que não são consecutivas pode gerar situações de conflito: uma âncora pode possuir mais de um sucessor que satisfaz o critério de distância. Assim, um bloco de sintenia não pode ser representado por uma única cadeia de âncoras, todas colineares umas com as



(a) Genomas G_1 e G_2 (b) *Dotplot* dos genomas G_1 e G_2

Figura 4.4: Exemplo de conflito na construção de blocos de sintenia. O bloco de sintenia é construído da seguinte maneira: duas âncoras colineares pertencem ao mesmo bloco se elas estão distantes no máximo de G âncoras. Neste exemplo, as âncoras a, b, c e d pertencem a um mesmo bloco de sintenia: a e b são colineares e $d(a,b) < G$, b e c são colineares e $d(b,c) < G$ e b e d são colineares e $d(b,d) < G$. Podemos observar que b possui dois sucessores possíveis c e d. Assim, dentro deste bloco, podemos ter dois encadeamentos possíveis. Além disso, neste caso as âncoras não são todas colineares entre si: c e d não são colineares.

outras. A Figura 4.4 exibe uma exemplo deste tipo de conflito.

Outro conflito possível é a sobreposição de dois blocos de sintenia mesmo quando as âncoras não se sobrepõem e são do tipo 1-1. Por exemplo, uma âncora que foi ignorada na construção de um bloco pode pertencer a outro bloco.

Os conflitos de sobreposição ou de encadeamento nem sempre são fáceis de serem resolvidos. Alguns métodos sequer mencionam a possibilidade de ocorrência de conflitos. Outros métodos fazem escolhas arbitrárias para evitar os conflitos, mas os critérios utilizados não são justificados ou comentados.

Sankoff *et al.* propõem uma definição formal de blocos de sintenia sem conflitos [39,174]. A abordagem proposta difere dos métodos citados até então. O princípio utilizado é de eliminar âncoras do conjunto de dados inicial de modo a obter blocos de sintenia sem qualquer flexibilidade. Estes blocos, denominados de "cadeias puras" (*pure strip*), são compostos apenas por âncoras que são colineares e consecutivas nos

dois genomas. O critério para eliminar as âncoras é global: minimizar o número de âncoras eliminadas para a obtenção de apenas cadeias puras. Este problema é NP-difícil e a sua resolução não é viável em conjuntos de dados reais. Assim, os autores propõem heurísticas para limitar o espaço de busca. Elas são baseadas na adição de restrições, como por exemplo uma limitação da distância entre as âncoras, similar ao parâmetro G de outros métodos descritos anteriormente.

4.2.3 Alinhamento de genomas completos

Quando utilizamos a lista de marcadores ortólogos, a resolução dos segmentos conservados depende fortemente da densidade de marcadores disponíveis. Por exemplo, os genes ortólogos 1-1 entre homem e camundongo correspondem a aproximadamente 30% dos genes contidos no genoma humano.

O sequenciamento de genomas completos trouxe um novo nível de detalhe ao estudo de rearranjos de genomas. De posse dos dados das sequências dos cromossomos, podemos realizar alinhamentos para detecção dos blocos conservados. Uma série de métodos de alinhamento de genomas completos está disponível e podem ser vistos na revisão escrita por Lemaitre e Sagot [101].

O alinhamento de genomas completos, em particular no caso dos genomas dos vertebrados, apresenta uma série de problemas que nos impede de utilizar os algoritmos clássicos de alinhamento de sequências (alinhamentos local ou global).

O primeiro problema é o tamanho das sequências envolvidas que inviabiliza a utilização de algoritmos exatos. Por exemplo, o genoma humano é composto por quase 3 bilhões de pares de bases. Essa característica faz com que heurísticas sejam utilizadas para o processamento das sequências.

Além de serem grandes, as sequências são muito heterogêneas em relação a conservação da sequência. No caso do homem e do camundongo, por exemplo, a divergência ocorreu há 75 milhões de anos e apenas 40% de seus genomas são alinháveis. De acordo com estudos de alinhamentos, apenas 5% do genoma humano está sob pressão de seleção e as regiões codificantes representam apenas 1,5% do genoma [168].

Muitos algoritmos foram desenvolvidos com o objetivo de alinhar sequências codificantes. Contudo, alinhar sequências intergênicas é uma tarefa mais difícil pois elas são regiões menos conservadas, podem possuir grandes *indels* e segmentos que não possuem nenhuma similaridade com seus ortólogos. Assim, normalmente espera-se que os algoritmos sejam capazes de "saltar" estas regiões sem muito custo.

Por último, os genomas sofrem rearranjos e duplicações. Os algoritmos clássicos de alinhamento global requerem que a ordem e a orientação de nucleotídeos homólogos sejam as mesmas nas duas sequências e, além disso, um nucleotídeo pode ser alinhado uma única vez. Existem algoritmos exatos que realizam a detecção de rearranjos (inversões e translocações) e/ou duplicações [33, 99, 161]. Contudo, eles lidam apenas com pequenos rearranjos locais e duplicações em *tandem* e, devido a complexidade que apresentam, não são viáveis para a realização de alinhamentos com sequências de genomas completos.

Os algoritmos de alinhamento global não são adaptados a genomas que sofreram rearranjos. Contudo, ele pode ser utilizado para alinhar regiões que estão dentro das regiões conservadas. A estratégia, portanto, é detectar as regiões conservadas e alinhar globalmente cada uma delas independentemente.

A detecção das regiões conservadas também é feita através de alinhamentos, porém algoritmos locais são utilizados para identificação de pares de pequenas sequências, muito similares, denominadas âncoras. A dificuldade agora é distinguir, dentre todos os pares de sequências obtidos, os segmentos ortólogos dos segmentos não ortólogos, que são similares por coincidência ou por duplicação, denominados falsos positivos.

Para realizar esta distinção, um processo de filtragem é necessário. Para filtrar os falsos positivos, informação contextual é utilizada em conjunto com a suposição de que falsos positivos são representados por âncoras isoladas. Esta etapa possui muitas similaridades com os algoritmos de construção de blocos de sintenia listados no final da Seção 4.2.2. O objetivo é o mesmo e a diferença se encontra nos dados de entrada pois, ao invés de genes ortólogos, resultados de alinhamentos locais são utilizados. Porém, como as âncoras produzidas pelos alinhamentos são muito mais numerosas e apresentam muito mais ruído, métodos adaptados a esta realidade podem ser necessários [142].

Basicamente, os métodos de alinhamento de genomas completos possuem 3 etapas:

1. Detecção de âncoras

2. Filtragem de âncoras

3. Alinhamento ou extensão de regiões homólogas

Detecção de âncoras

O objetivo da etapa de detecção de âncoras é, considerando o tamanho dos genomas comparados, identificar similaridades locais entre eles. Esta etapa costuma estar presente em todos os métodos disponíveis na literatura. No entanto, eles apresentam diversos algoritmos diferentes. As âncoras podem ser palavras exatas ou quase exatas, podem ser alinhamentos locais com ou sem *indels*. Elas possuem tamanhos variados e, às vezes, devem respeitar algumas condições de unicidade e/ou de não sobreposição.

Alinhamento local O método `CHAINNET` [94] utiliza o algoritmo de alinhamento local `BLASTZ` [144]. Ele utiliza como âncoras apenas as subsequências sem *indels* dos alinhamentos obtidos.

O método `GRIMM` [131], por sua vez, utiliza os alinhamentos locais com *indels* produzidos pelo algoritmo `Pattern Hunter` [108]. Contudo, ele preserva apenas os alinhamentos sem sobreposição (em ambos os genomas) e que não são duplicados.

Os algoritmos de alinhamento local `BLASTZ` e `Pattern Hunter` são muito similares e foram desenvolvidos na mesma época com o objetivo de alinhar sequências pouco conservadas como, por exemplo, as que tem origem em regiões não codificantes. Eles são baseados na mesma estratégia *seed-and-extend* do algoritmo `BLAST` [3]. Utilizada para acelerar a busca por similaridade local, esta estratégia baseia-se na hipótese que duas sequências similares tem grandes chances de possuírem palavras exatas em comum denominadas sementes. As sementes servem como pontos de ancoragem dos alinhamentos. Elas são estendidas de cada lado sem a autorização de *indels* até que a pontuação atinja um determinado valor mínimo. Atingido esse valor, o alinhamento é ainda estendido, porém com a possibilidade de inclusão de *indels*, com um algoritmo de programação dinâmica. Ao final do processo, apenas os alinhamentos que possuem pontuação suficiente são mantidos.

`BLASTZ` e `Pattern Hunter` diferem do `BLAST` principalmente em relação ao tipo de semente utilizada. Ao invés de procurarem palavras exatas, eles buscam palavras de tamanho l nas quais algumas posições fixas devem ser exatas (sementes espaçadas). Além disso, o `BLASTZ` autoriza uma transição (mutação de uma base purina para outra base purina ou de uma base pirimídica para uma base pirimídica) no local de uma correspondência (*match*). Este tipo de semente é muito sensível e o `BLASTZ` é um dos algoritmos mais precisos para alinhar sequências não codificantes.

Couronne *et al*. [45] desenvolveram um método que utiliza o algoritmo de alinha-

mento local BLAT [93] para a fase de detecção de âncoras. Este algoritmo faz parte da família de algoritmos *seed-and-extend* e suas sementes são palavras exatas. Antes da extensão, existe uma etapa de seleção das sementes: apenas os grupos de sementes próximas e sobre uma mesma diagonal serão estendidos. BLAT não foi desenvolvido para o alinhamento de sequências de espécies diferentes mas ele tem a vantagem de ser muito rápido.

Estes três métodos de definição de âncoras possuem muitos parâmetros: tamanho das sementes, matrizes de substituição, penalização para os *indels*, valores mínimos de pontuação para extensão e para manutenção dos alinhamentos, etc. A escolha dos valores possui uma grande influência sobre a sensibilidade e especificidade dos algoritmos. Ela depende igualmente das espécies que estão sendo alinhadas. Apesar destes dois aspectos, a parametrização é pouco discutida e muitas vezes é arbitrária.

Busca por palavras exatas O método MAUVE [49] utiliza para esta etapa um algoritmo diferente dos outros três métodos citados. Enquanto o objetivo dos métodos anteriores é ter um alto grau de sensibilidade, MAUVE é muito rigoroso e específico na definição de âncoras. Ele busca por palavras exatas e únicas (aparecem uma única vez em cada genoma) denominadas MUMs (*Maximum Unique Match*).

O tamanho das âncoras encontradas pelos diferentes métodos varia de alguns pares de base (MAUVE) até segmentos mais longos com 500 bp (GRIMM), passando por âncoras de tamanho médio intermediário de 30 bp (CHAINNET).

Filtragem de âncoras

O objetivo da etapa de filtragem é a eliminação de falsos positivos do conjunto de âncoras. Também pode-se desejar a eliminação de pequenos rearranjos locais. A idéia principal é a utilização de informação contextual em conjunto com a hipótese de que erros, quando existem, são isolados. Segundo esta hipótese, duas âncoras que se encontram próximas uma da outra e a uma mesma distância em ambos os genomas possuem menos chance de serem um erro do que uma âncora isolada.

As estratégias utilizadas são similares aos métodos de construção de blocos de sintenia baseados na comparação da ordem de genes. Elas podem ser classificadas segundo dois tipos de abordagem: encadeamento de âncoras e reagrupamento (*clustering*) de âncoras.

Clustering Os métodos de *clustering* utilizam apenas a distância entre as âncoras como critério de agrupamento. O método `GRIMM` e o método proposto por Couronne *et al*. baseiam-se neste tipo de método.

A distância utilizada por `GRIMM` é a distância de Manhattan expressa em pares de bases. Duas âncoras pertencem a um mesmo *cluster* se a distância entre elas é inferior a um determinado valor G (*gap*), mesmo se elas não são colineares. Somente os *clusters* que cobrem uma distância mínima S sobre o genoma são mantidos. Devido a natureza deste método, as âncoras dentro dos *clusters* não formam, obrigatoriamente, uma cadeia e podem possuir orientações diferentes.

A estratégia do método de Couronne *et al*. parece ser similar (o método não é claramente descrito no artigo) pois as âncoras são agrupadas de acordo com a distância. Apesar de a orientação e ordem das âncoras não entrar na construção dos *clusters*, estes critérios são indiretamente verificados na etapa seguinte quando as sequências de um *cluster* são alinhadas globalmente. *Clusters* que apresentam muita desordem recebem baixa pontuação nos alinhamentos e, dessa maneira, são descartados.

Encadeamento de âncoras Os métodos de encadeamento, além da distância, levam em conta a ordem e a orientação das âncoras. Os métodos `MAUVE` e `CHAINNET` fazem parte desta família. Eles buscam realizar um encadeamento no qual todas as âncoras são colineares umas com as outras.

`MAUVE` não utiliza informação de distância física mas as âncoras devem ser colineares e consecutivas nos dois genomas sem apresentarem nenhuma desordem. Apenas as cadeias cujas âncoras cobrem um comprimento mínimo são mantidas. Esta estratégia permite a construção de cadeias sem conflitos: a ordem é estrita e não existem sobreposições. Por ser muito rigoroso (nenhuma flexibilidade é permitida), `MAUVE` é adequado a dados que possuem poucos erros ou ruídos.

`CHAINNET` encadeia as âncoras analisando ordem, orientação e distância. O algoritmo adota uma estrutura de árvore que permite particionar os dados em um espaço de k dimensões. Ela é utilizada para encontrar todos os pontos que estão sobre um dado hiper-retângulo com o objetivo de buscar âncoras colineares que podem ser encadeadas [173]. Ao contrário do que acontece com `MAUVE`, as cadeias produzidas podem se sobrepor ou podem "saltar" âncoras que não estão na ordem ou orientação esperada. `CHAINNET` produz todas as cadeias possíveis e associa a cada uma delas uma pontuação em função do número de âncoras, da distância coberta, das distâncias

entre as âncoras, etc. No artigo em que o método é descrito, a função de pontuação assim como os pesos relativos dos diferentes critérios não são precisamente definidos.

Alinhamento ou extensão de regiões homólogas

A fase de alinhamento de regiões homólogas é a que mais apresenta diferenças entre os métodos que podem ser encontrados na literatura. Isso se deve, principalmente, aos diferentes objetivos de cada método.

O método `GRIMM` foi desenvolvido para reconstrução da história dos rearranjos a partir da análise dos eventos que separam dois genomas. Assim, a saída do algoritmo é composta apenas pelo arranjo das diferentes regiões conservadas entre os dois genomas. De acordo com a estratégia do algoritmo, o modo como as sequências se alinham no interior das regiões conservadas não é útil para o estudo dos cenários de rearranjo. Além disso, as regiões conservadas produzidas pelo método são longas e podem conter pequenos rearranjos locais, denominados micro-rearranjos. Isso explica o motivo pelo qual `GRIMM` não impõem a ordenação das âncoras dentro dos *clusters*.

Nesta última etapa, `GRIMM` encadeia os *clusters* que se encontram colineares e consecutivos nos dois genomas, não importando a distância física que os separa. Isso permite a produção de longos blocos de sintenia. Uma etapa que não é clara na descrição do método é a maneira como a orientação dos *clusters* é atribuída já que as âncoras contidas dentro deles podem ter orientações diferentes.

O método `CHAINNET` e o método proposto por Couronne *et al*. possuem a meta de produzir alinhamentos entre dois genomas de modo que todos os nucleotídeos potencialmente homólogos estejam alinhados. Estes alinhamentos são úteis para estudo do processo evolutivo tanto em grande como em pequena escala. Apesar de terem meta similar, a terceira etapa de ambos os métodos são bem diferentes.

Couronne *et al*. não fazem a extensão dos *clusters* obtidos na etapa anterior. Eles simplesmente alinham cada *cluster* com um algoritmo de alinhamento global denominado `AVID` [32]. Apenas os alinhamentos que possuem pontuação maior que um dado valor mínimo são mantidos. Esta estratégia gera uma grande quantidade de pequenos alinhamentos.

`CHAINNET` trata as cadeias identificadas na segunda etapa, uma após a outra, em ordem decrescente de pontuação. As cadeias são colocadas sobre o genoma de referência e o algoritmo marca as posições que são cobertas por âncoras. Uma posição

do genoma pode ser coberta somente por uma âncora de uma única cadeia. Assim, quando uma cadeia cobre uma região onde existem posições já marcadas, o algoritmo registra apenas as partes da cadeia que cobrem posições que ainda não foram marcadas. Se uma cadeia pode se inserir em um buraco de uma cadeia já posicionada, ela é adicionada em um nível hierarquicamente inferior. Dessa maneira, o algoritmo produz uma rede de cadeias, com pais e filhos. A motivação por trás desta idéia é a identificação de rearranjos que estão inseridos em blocos de sintenia maiores.

A técnica adota pelo método `MAUVE` consiste em aplicar as duas primeiras etapas de maneira recursiva, com parâmetros menos rigorosos (tamanho mínimo dos `MUMs`) em regiões mais restritas. Entre cada par de `MUMs` consecutivos, as duas primeiras etapas são aplicadas: busca por `MUMs` e encadeamento. Feito isso, novas cadeias são procuradas entre as cadeias existentes. Quando nenhuma cadeia nova é encontrada, as sequências cobertas pelas cadeias são alinhadas globalmente.

Configurações dos métodos e tratamento de casos especiais

Os métodos de alinhamento de genomas completos encontram os mesmos problemas que os métodos baseados na análise da ordem dos genes: sobreposições de blocos, tratamento de duplicações e escolha de parâmetros para determinar que rearranjos devem ser mantidos e quais devem ser descartados.

Ajuste de parâmetros e descarte de rearranjos `GRIMM` denomina de micro-rearranjos os pequenos rearranjos que são eliminados ou mascarados no interior dos blocos de sintenia. Os parâmetros G (*gap*) e S (tamanho dos blocos) determinam quais rearranjos são guardados ou não. Normalmente estes parâmetros são fixos de maneira arbitrária. Quando o método foi publicado, ambos foram fixados em 1 Mbp, o que implica a perda de rearranjos menores do que este valor. Isso diminui consideravelmente a resolução do método. Contudo, ao reduzir este valor deve-se ter ciência de que o número de erros e falsos positivos pode aumentar.

Por ser similar ao método `GRIMM`, o método proposto por Couronne *et al*. poderia mascarar os pequenos rearranjos. Porém, o alinhamento global dos *clusters* evidencia as suas existências. Por outro lado, o método é parametrizado para produzir pequenos blocos e como eles são consecutivos e colineares nos dois genomas, eles não são reagrupados. Atualmente, com o objetivo de aumentar a confiabilidade dos blocos obtidos, os autores privilegiam a utilização de *exons* como âncoras potenciais

em detrimento dos alinhamentos genômicos (método `Mercator` [52,53]). Esta é uma estratégia frequentemente utilizada para comparação de genomas distantes [26].

O método `CHAINNET` não possui o parâmetro de tamanho e não elimina os rearranjos segundo este critério. Contudo, os dados produzidos não podem ser utilizados diretamente para análise. Por exemplo, na publicação do método os autores definiram um limite de 100 kbp para filtrar os rearranjos para realização da análise. Esta análise incluiu apenas as inversões (visíveis no nível 2) e os rearranjos sem sobreposição (nível 1).

A estrutura hierárquica de cadeias produzidas pelo `CHAINNET` é de difícil utilização. Em um mesmo nível podemos visualizar cadeias muito grandes em conjunto com cadeias muito pequenas e até mesmo âncoras isoladas. Além de produzir dúvidas quanto a confiabilidade do método, estes dados são difíceis de serem interpretados do ponto de vista biológico.

A plataforma `Ensembl` [85] utiliza uma combinação do método `GRIMM` com o método `CHAINNET`. Os alinhamentos produzidos pelo `CHAINNET` são usados como âncoras em um algoritmo de reagrupamento de blocos similar ao do `GRIMM` [94].

O algoritmo `MAUVE` foi concebido para comparação de genomas bacterianos próximos e não é adaptado para sequências de vertebrados. Sua etapa de encadeamento é pouco flexível e sua etapa de busca de âncoras é muito rigorosa para genomas com poucas regiões codificantes e com muitas repetições.

Blocos de sintenia que se sobrepõem Do ponto de vista biológico, dois blocos de sintenia que se sobrepõem sobre um genoma denotam a possibilidade da existência de rearranjos suplementares ou de duplicações sobre o outro genoma.

`CHAINNET` e `MAUVE` produzem alinhamentos sem sobreposição. Quando uma sobreposição é possível, estes métodos escolhem uma das possibilidades. O método proposto por Couronne *et al.* e o método `GRIMM` não realizam controle de sobreposição entre os blocos.

Duplicações Nenhum dos métodos citados tentam identificar duplicações. `GRIMM` e `MAUVE` eliminam as duplicações logo na primeira etapa quando exigem um conjunto de âncoras que aparecem apenas uma vez em cada genoma. `GRIMM` também elimina as âncoras que se sobrepõem. Estes critérios eliminam uma grande quantidade de âncoras. Caso o tratamento destas âncoras fosse delegado à etapa de filtragem, métodos para selecionar as boas âncoras e eliminar as não relevantes poderiam ser

aplicados. Além disso, como `GRIMM` permite a ocorrência de sobreposições, a detecção de duplicações poderia ser feita durante a fase de identificação de blocos de sintenia.

Couronne *et al*. não indicam claramente como as duplicações são tratadas. Pelas características do algoritmo, as duplicações são mantidas se elas estão dentro de um *cluster* significativo e, posteriormente, estão dentro de um alinhamento significativo.

No método `CHAINNET` as duplicações são processadas de maneira assimétrica. Elas são autorizadas apenas no genoma de referência. Se uma posição do genoma de referência pertence a diversas cadeias (posição potencialmente duplicada no outro genoma), a cópia que estiver na cadeia de maior pontuação é escolhida. Esta assimetria produz alinhamentos diferentes em função do genoma que é escolhido como referência.

Capítulo 5

Identificação e refinamento de *Breakpoints*

Durante o seu estudo de doutorado realizado no *Laboratoire de Biométrie et Biologie Évolutive* da *Université Claude Bernard – Lyon I*, Claire Lemaitre desenvolveu um método para detecção e refinamento de *breakpoints* a partir da análise comparativa da lista de genes ortólogos existente entre duas espécies [100, 102].

O método é composto de duas etapas principais: detecção de blocos de sintenia e refinamento dos *breakpoints* identificados. Durante o seu desenvolvimento, diversos *scripts* foram criados para a execução de tarefas intermediárias. Contudo, eles não eram interligados e alguns não podiam ser parametrizados.

Como parte do nosso objetivo final, que é estudar os pontos de quebra e descobrir características que nos permitam identificá-los mais precisamente, realizamos a implementação do método proposto por Lemaitre *et al.*. A implementação foi utilizada para identificação e refinamento de *breakpoints* que podem ser encontrados nos genomas do homem (*Homo sapiens*) e do camundongo (*Mus musculus*).

O método foi implementado em `Perl` [46] e `R` [138] e gerou um pacote nomeado `Cassis`. Este pacote foi publicado como um *Application Notes* na revista *Bioinformatics* sob o título "Cassis: precise detection of genomic rearrangement breakpoints" [16]. O código produzido assim como as instruções de uso estão disponíveis no endereço `http://pbil.univ-lyon1.fr/software/Cassis/`. Além disso, um pôster sobre este trabalho foi apresentado na 9^{th} *European Conference on Computational Biology* (`ECCB 2010`) sob mesmo título [17].

Neste capítulo apresentaremos detalhes do método e mostraremos resultados ob-

tidos no estudo comparativo entre o homem e o camundongo.

5.1 Obtenção de pontos de quebra em duas etapas

Para analisar as características das sequências dos pontos de quebra, é altamente desejável a obtenção de um conjunto de blocos de sintenia com boa resolução.

Para atingir este objetivo, métodos de alinhamento de genomas completos parecem ser mais adequados já que eles produzem uma densidade maior de marcadores. Contudo, como demonstrado no capítulo anterior (Seção 4.2.3), tais métodos possuem alguns inconvenientes e conforme os critérios utilizados, os resultados obtidos nem sempre possuem a resolução desejada.

A resolução é um compromisso entre sensibilidade e especificidade do método. Os alinhamentos de sequências, especialmente de sequências não codificantes, podem conter muitos falsos positivos causados por duplicações, repetições ou erros de sequenciamento. A etapa de filtragem ajuda a minimizar estes problemas. Porém, quanto maior o rigor dos parâmetros maior a perda de resolução.

Os métodos que utilizam conjuntos de marcadores ortólogos dependem da densidade de marcadores disponíveis. Eles tendem a ser mais confiáveis porém oferecem menor resolução à definição dos pontos de quebra, já que as bordas dos blocos de sintenia são limitados pelas bordas dos marcadores que estão em suas extremidades.

Em ambas as abordagens, o objetivo é identificar os blocos de sintenia e os *breakpoints* são definidos como uma mera consequência deste processo. Nenhuma análise é realizada sobre as sequências que estão entre os blocos de sintenia, regiões onde os pontos de quebra se encontram.

Diante deste cenário, Lemaitre *et al.* propuseram uma nova abordagem para a identificação de pontos de quebra no genoma em um processo que é dividido em duas etapas:

1. Identificar blocos de sintenia confiáveis sem exigir alta resolução
2. Refinar cada ponto de quebra de maneira independente através da extensão dos blocos de sintenia que o definem

Dividindo o processo em duas etapas, o método pode ser mais rigoroso no que se refere a identificação de blocos de sintenia. Uma vez que o espaço de busca é reduzido,

a segunda etapa pode ser menos rígida. Assim, Lemaitre *et al.* optaram por utilizar dados de genes ortólogos, mais confiáveis, para a primeira etapa e trabalhar com a sequência genômica na segunda. A primeira etapa será descrita na Seção 5.1.1 e a segunda etapa será apresentada na Seção 5.1.2.

5.1.1 Identificação de blocos de sintenia

Para fase de identificação de blocos de sintenia, Lemaitre *et al.* optaram pela utilização de conjuntos de genes ortólogos. Com base neste tipo de informação, espera-se a identificação de blocos com uma probabilidade menor de ocorrência de falsos positivos ou erros.

Antes de chegar ao método final, Lemaitre realizou um estudo envolvendo um método simples para definição de blocos de sintenia. Este método definia um *breakpoint* entre dois genes ortólogos a_r e b_r no genoma de referência G_r quando seus correspondentes a_o e b_o no genoma G_o não são consecutivos e colineares por estarem em cromossomos diferentes (*breakpoints inter-cromossômicos*) ou por estarem em um mesmo cromossomo, mas em diferentes contextos (*breakpoints intra-cromossômicos*). A Figura 5.1 exibe um esquema de um *breakpoint* inter-cromossômico e a Figura 5.2 ilustra um esquema de um *breakpoint* intra-cromossômico.

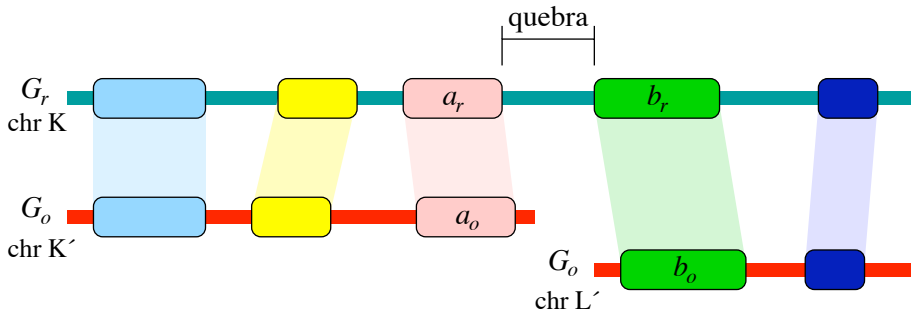


Figura 5.1: Esquema de um *breakpoint* inter-cromossômico. Um ponto de quebra é definido quando temos dois genes sucessivos a_r e b_r em um genoma de referência G_r e seus ortólogos a_o e b_o, respectivamente, estão em dois cromossomos diferentes no genoma G_o.

Lemaitre obteve dois conjuntos de genes ortólogos entre o homem e o camundongo para realizar o estudo. Ambos os conjuntos podem ser obtidos da base de dados `GeMCore` [115, 116] que foi construída com base nas sequências e anotações

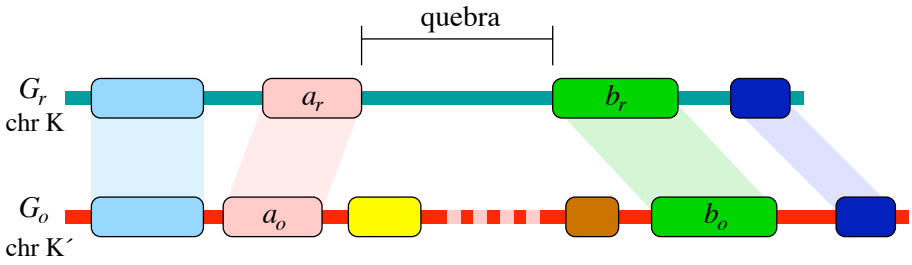


Figura 5.2: Esquema de um *breakpoint* intra-cromossômico. Um ponto de quebra é definido quando temos dois genes sucessivos a_r e b_r em um genoma de referência G_r e seus ortólogos a_o e b_o, respectivamente, estão em um mesmo cromossomo no genoma G_o mas não são colineares e consecutivos (estão em diferentes contextos).

extraídas da base de dados `Ensembl` (versão 24) [85]. O primeiro conjunto, que chamaremos `MHR`, foi construído utilizando o método do melhor *hit* recíproco. O segundo conjunto, que denominaremos `TREE`, foi obtido com uma abordagem filogenética de reconciliação de árvores. Nesta abordagem, os genes dos genomas são agrupados em famílias. Alinhamentos múltiplos são realizados com as sequências dos genes de uma família para a criação de uma árvore filogenética. Comparando a árvore obtida com a árvore filogenética das espécies, podemos distinguir eventos de duplicação e especiação e, assim, identificar as relações de ortologia [58].

Os conjuntos `MHR` e `TREE` são formados apenas por genes ortólogos 1-1. Com base nestes dois conjuntos de dados, um terceiro foi criado contendo a interseção dos genes dos dois outros conjuntos.

A Tabela 5.1 mostra os resultados obtidos por Lemaitre ao aplicar o método simples de indentificação de *breakpoints*.

Tabela 5.1: Resultados obtidos por Lemaitre na identificação de *breakpoints* sobre três conjuntos de genes ortólogos.

Conjunto de dados	Número de Genes ortólogos	Número de *Breakpoints*	Número (e %) de *Breakpoints* de tipo I ou II
`MHR`	15268	1480	510 (34%)
`TREE`	12474	986	352 (36%)
`MHR` ∩ `TREE`	12027	657	134 (20%)

Como podemos ver, os diferentes conjuntos apresentam uma grande diferença no número de *breakpoints* identificados. Comparando os conjuntos `MHR` e `TREE`, Lemaitre aponta que apenas 308 pontos de quebra eram idênticos (definidos pelo mesmo par de genes a_r e b_r). Isso indica que os dados de *breakpoints* são muito relacionados às metodologias de atribuição de ortologia empregadas.

A maior parte das diferenças entre os conjuntos `MHR` e `TREE` foi causada por genes que possuíam um ortólogo em um conjunto mas não possuíam em outro. Além disso, 74 genes do homem possuíam ortólogos no camundongo distintos em cada um dos conjuntos (em pelo menos um dos conjuntos, a atribuição de ortologia ao gene foi errônea).

Dentro dos conjuntos de pontos de quebra obtidos, Lemaitre pode avaliar a presença de duas classes que agruparam de 20 a 36% dos breakpoints conforme o conjunto de dados processados.

A primeira classe é composta por *breakpoints* do tipo `I`. Este tipo de ponto de quebra ocorre quando dois genes a_r e b_r no genoma de referência possuem dois genes ortólogos a_o e b_o separados por um único gene x_o no genoma G_o. A segunda classe é composta por *breakpoints* do tipo `II` que ocorrem quando dois genes c_r e e_r no genoma de referência possuem entre ele um gene d_r cujo ortólogo d_o não se encontra entre os genes c_o e e_o no genoma G_o. A Figura 5.3 exibe um esquema exemplificando estes dois tipos de breakpoints.

Estes dois tipos de *breakpoints* possuem as características esperadas nos casos de erros de atribuição de ortologia e, por isso, podemos desconfiar da informação de ortologia atribuídas aos genes x_o, no caso do tipo `I` e ao par (d_r, d_o), no caso do tipo `II` (ver Figura 5.3).

Diante destes resultados, Lemaitre concluiu que o método rigoroso adotado pode gerar uma série de *breakpoints* falsos positivos. Um processo de filtragem poderia ser aplicado, eliminando os genes ortólogos isolados que, sozinhos, quebram blocos de sintenia e geram os conflitos do tipo `I` ou `II`. Contudo, casos onde um único gene quebra um bloco de sintenia podem ser verdadeiros *breakpoints* e, além disso, a eliminação de genes que geram conflitos podem criar ou eliminar outros conflitos, dependendo da ordem em que eles são removidos.

Com o objetivo de melhorar a definição dos blocos de sintenia e minimizar problemas como os conflitos do tipo `I` e `II`, Lemaitre optou por utilizar uma nova abordagem para a definição de blocos de sintenia com a preocupação extra de definir formalmente os objetos envolvidos no processo. Esta abordagem deveria ser flexível

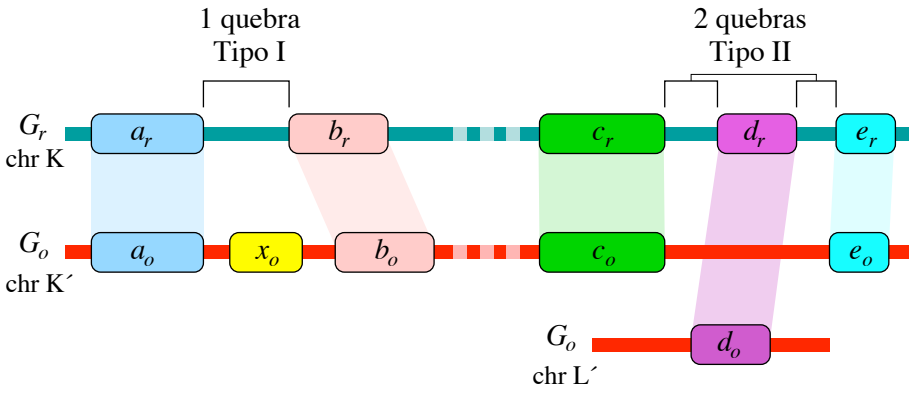


Figura 5.3: Representação de dois casos particulares de *breakpoints*. O primeiro caso é composto por um *breakpoint* de tipo `I` que é um ponto de quebra intra-cromossômico entre os genes a_r e b_r no genoma G_r e que foi detectado por causa da presença de um único gene x_o entre os genes a_o e b_o no genoma G_o. O segundo caso é formado por dois *breakpoints* do tipo `II` que são pontos de quebra inter-cromossômicos entre c_r e d_r e entre d_r e e_r e que foram identificados devido a presença do gene d_r entre os genes c_r e e_r.

e, ao mesmo tempo, garantir a produção de blocos que não se sobrepõem e cujos marcadores ortólogos contidos nas extremidades do bloco fossem os mesmos nos dois genomas. Este tipo de bloco de sintenia é denominado *bloco sem conflito* e, como vimos na revisão feita no Capítulo 4, ele possui características que raramente são observadas pelos métodos existentes.

Método de identificação de blocos de sintenia

O método recebe como entrada um número inteiro k e um conjunto de âncoras entre dois genomas G_r e G_o. Uma âncora é um par de marcadores, um em cada genoma, que são ortólogos. Apenas marcadores que não se sobrepõem e que pertencem a uma única âncora (relação 1-1) são aceitos pelo método. Marcadores sem ortólogos são ignorados.

Através de uma orientação arbitrária da sequência do cromossomo, definindo em qual extremidade se encontra o seu início, os marcadores podem ser identificados pelas posições que eles ocupam neste cromossomo (em relação ao início) e por suas orientações. A posição do marcador é definida pelo índice que ele ocupa na lista de marcadores que cobrem o cromossomo. Assim, o primeiro marcador do cromossomo

recebe o número 1, o segundo recebe o número 2, e assim por diante.

Uma âncora é definida por um par de cromossomos, um par de posições (índices dos marcadores em cada genoma) e uma orientação relativa. Assim, uma âncora a é identificada pelo objeto $(c_r, c_o, a_r, a_o, \sigma_a)$ onde, c_r e a_r são o cromossomo e a posição do marcador no genoma G_r, c_o e a_o são o cromossomo e a posição do marcador no genoma G_o e σ_a é igual a $+1$ se os dois marcadores estão na mesma orientação e igual a -1, caso contrário.

Se duas âncoras distintas a e b estão sobre os mesmos cromossomos nos dois genomas, a distância entre a e b, denotada por $d(a, b)$, é o máximo das diferenças entre os índices de posição de a e b em cada genoma: se a_r e a_o (b_r e b_o) são os índices da âncora a (b) sobre os genomas G_r e G_o, então $d(a, b) = max(|b_r - a_r|, |b_o - a_o|)$. Segundo esta definição de distância, se a e b são consecutivos em ambos os genomas, $d(a, b) = 1$. Se duas âncoras contém marcadores que não se localizam sobre o mesmo cromossomo em pelo menos um dos dois genomas, a distância entre elas é infinita.

Um grafo de âncoras $\mathcal{G}_k$ é um grafo dirigido cujos vértices são as âncoras e duas âncoras são ligadas por um arco se elas são colineares e possuem uma distância entre elas no máximo igual a k. De modo mais formal, um arco ab conecta as âncoras a e b se:

- $a_r < b_r$
- $d(a, b) \leq k$
- ($a_o < b_o$ e $\sigma_a = \sigma_b = 1$) ou ($a_o > b_o$ e $\sigma_a = \sigma_b = -1$)

Um caminho neste grafo representa um conjunto de âncoras colineares próximas nos dois genomas. Este caminho é denominado cadeia. Uma âncora pode pertencer a diversas cadeias.

O objetivo do método é encontrar conjuntos de vértices conectados no grafo G_k. Tais conjuntos são componentes fracamente conexas neste grafo. Uma componente conexa de um grafo não orientado é um sub-grafo conectado maximal: existe um caminho entre quaisquer dois vértices pertencentes à componente conexa. Uma componente fracamente conexa de um grafo orientado G, é uma componente conexa do grafo não orientado G' produzido através da substituição dos arcos por arestas. Dois vértices a e b estão em uma componente conexa de G' se existe um caminho orientado em G entre os vértices a e b ou entre b e a.

Se o valor de k é maior do que 1, uma âncora pode ser encadeada a diversas âncoras distintas. Assim, podemos obter componentes fracamente conexas que possuem âncoras que não são colineares duas a duas. Podemos também obter componentes fracamente conexas cujas coordenadas genômicas se sobrepõem indicando conflito entre cadeias.

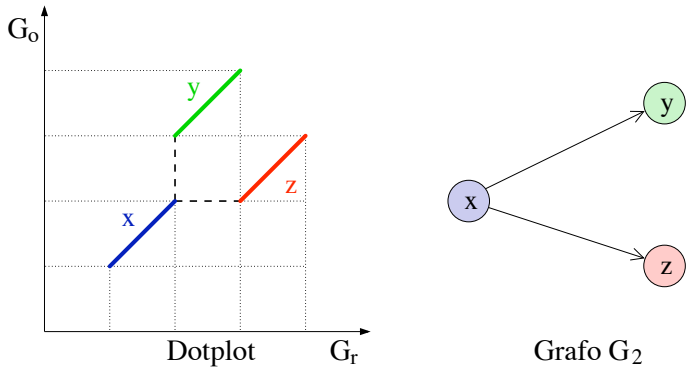


Figura 5.4: Exemplo de arcos que formam um conflito do tipo `I`. Na representação de *dotplot*, podemos observar as posições das âncoras x, y e z nos genomas G_r e G_o. As âncoras estão sobre o mesmo cromossomo nos dois genomas e $d(x, y) = d(x, z) = 2$. O grafo correspondente $\mathcal{G}_2$ ($k = 2$) possui uma única componente fracamente conexa. Os arcos xy e xz são conflitantes pois a ordem dos marcadores é x, y e z no genoma G_r e x, z e y no genoma G_o: as âncoras y e z não são colineares.

Em um conflito de tipo `I`, dois arcos ab e cd, que pertencem a uma mesma componente fracamente conexa de $\mathcal{G}_k$, apresentam conflito se as âncoras a, b, c e d não são todas colineares duas a duas. Este tipo de conflito denota a existência de diversas cadeias dentro de uma componente fracamente conexa. A Figura 5.4 mostra um exemplo em que $a = c$.

Um arco ab em uma componente fracamente conexa C apresenta conflito do tipo `II` se existe uma âncora c que possui ao menos um de seus marcadores localizado entre os marcadores de a e b, em um dos dois genomas, e c pertence a outra componente fracamente conexa D que possui ao menos k âncoras. Esta configuração de arcos indica a ocorrência de cadeias que se sobrepõem. A Figura 5.5 mostra um conflito deste tipo.

O sub-grafo de $\mathcal{G}_k$ que contém os arcos não conflitantes de $\mathcal{G}_k$ é denominado $\mathcal{H}_k$. Um k-bloco é uma componente fracamente conexa de $\mathcal{H}_k$ que possui ao menos k

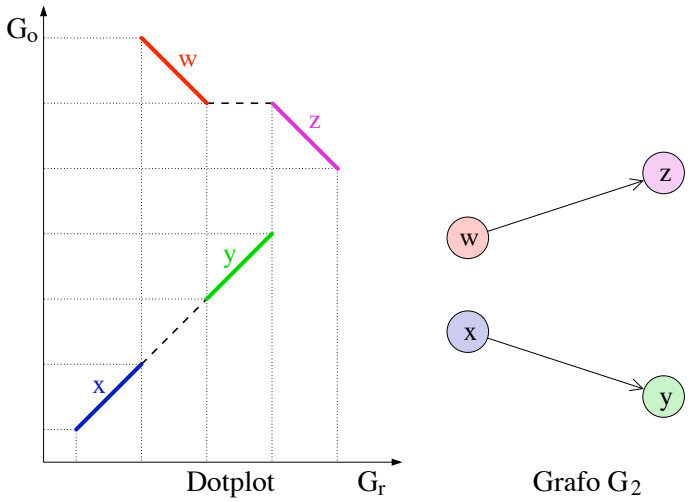


Figura 5.5: Exemplo de arcos que formam um conflito do tipo `II`. Na representação de *dotplot*, podemos observar as posições das âncoras x, y, w e z nos genomas G_r e G_o. As âncoras x e y estão sobre um mesmo cromossomo nos dois genomas e $d(a, b) = 2$. O mesmo pode ser dito sobre w e z. O grafo correspondente $\mathcal{G}_2$ ($k = 2$) possui duas componentes fracamente conexas: $\{x, y\}$ e $\{w, z\}$. O arco xy é conflitante pois $x_r < w_r < y_r$ e w_r pertence à outra componente fracamente conexa com pelo menos duas âncoras.

vértices. As coordenadas das âncoras nas extremidades dos k-blocos delimitam os blocos de sintenia do genoma.

A ausência de arcos conflitantes do tipo `II` garante que os blocos de sintenia não se sobrepõem. A eliminação dos arcos conflitantes do tipo `I` garante que as âncoras dentro de um bloco estão totalmente ordenadas. Dessa maneira, garante-se que cada extremidade de um bloco de sintenia é bem definida, sendo representada por uma mesma âncora nos dois genomas.

Aplicando as definições apresentadas, o tempo de cálculo dos k-blocos é polinomial. Se n é o número de âncoras, o cálculo dos índices de posições necessita de um procedimento de classificação de todas as âncoras sobre os dois genomas (tempo $O(n \log n)$). A construção do grafo gasta um tempo proporcional a $n \times k$ e produz no máximo $n \times k$ arcos. O tempo de cálculo das componentes fracamente conexas é proporcional ao número de arcos (tempo $O(n \times k)$). Para cada arco, a identificação de conflitos necessita da comparação com no máximo $2k$ outros arcos ou vértices. Assim, a complexidade de tempo total é $O(n \times k^2 + n \log n)$, onde k é um parâmetro fixo, geralmente pequeno (inferior a 10).

O Algoritmo 5.1.1 exibe os passos que devem ser efetuados para a produção de k-blocos. Este algoritmo tem o seu grau de flexibilidade controlada pelo parâmetro k. Se $k = 1$, nenhuma flexibilidade é permitida e apenas as âncoras colineares e consecutivas, nos dois genomas, são encadeadas. Se $k > 1$, os blocos são construídos através do encadeamento de marcadores colineares e com a aceitação de um certo número de marcadores "desordenados".

Este método difere de outros existentes no fato de que ele possui um único parâmetro. Em geral, os métodos possuem pelo menos dois parâmetros [86, 123, 131, 146]. O primeiro parâmetro G controla a distância máxima autorizada entre duas âncoras encadeadas e o segundo parâmetro S controla o tamanho mínimo do bloco construído para que ele seja mantido.

Lemaitre optou por utilizar um único parâmetro baseado na observação de que, em geral, os parâmetros G e S são relacionados. Para evitar que um bloco fique contido dentro de outro, é necessário fazer $G \leq S$. Por outro lado, a eliminação de blocos com tamanhos menores do que S é desejável pois supõem-se que tais blocos são erros. Como tais erros não podem influenciar o encadeamento de âncoras, normalmente é necessário fixar $G \geq S$. Assim, Lemaitre optou por fixar $G = S$. De fato, os parâmetros G e S são frequentemente iguais nas aplicações dos métodos existentes na literatura [26, 86, 131, 146].

Algoritmo 5.1.1: Construção de k-blocos

Entrada: Conjunto de âncoras sobre os genomas G_r e G_o e parâmetro k
Saída: Conjunto de k-blocos

Classificar as âncoras em G_r, atribuir o índice a_r a cada âncora a;
Classificar as âncoras em G_o, atribuir o índice a_o a cada âncora a;
`/* Construção do grafo */`
para cada *âncora* $a(a_r, a_o)$ **faça**
    para cada *âncora* $b(b_r, b_o)$ *tal que* $b_r = a_r + i$, $0 < i \leq k$ **faça**
        se *a e b são colineares e* $d(a, b) \leq k$ **então** Criar um arco $a \rightarrow b$;
Calcular as componentes fracamente conexas do grafo;
`/* Identificação de conflitos do tipo I */`
para cada *âncora* $a(a_r, a_o) \in$ *componente fracamente conexa de tamanho* $\geq k$ **faça**
    para cada *âncora* $b(b_r, b_o)$ *tal que* $(b_r = a_r + i,\ 0 < i \leq k)$ *ou* $(b_o = a_o + i,\ -k \leq i \leq k)$ **faça**
        se *(*$b \in$ *componente fracamente conexa que contém* a*) e (a e b não são colineares)* **então**
            Marque todos os arcos que entram e saem de a e b como conflitantes;
`/* Identificação de conflitos do tipo II */`
para cada *arco* $a(a_r, a_o) \rightarrow b(b_r, b_o) \in$ *componente fracamente conexa de tamanho* $\geq k$ **faça**
    para cada *âncora* $c(c_r, c_o)$ *tal que* $(a_r < c_r < b_r)$ *ou* $(a_o < c_o < b_o)$ **faça**
        se $c \in$ *componente fracamente conexa de tamanho* $\geq k$ *diferente da componente fracamente conexa que contém a e b* **então**
            Marque o arco $a(a_r, a_o) \rightarrow b(b_r, b_o)$ como conflitante;
`/* Cálculo dos` k`-blocos */`
Calcular as componentes fracamente conexas do grafo ignorando os arcos marcados como conflitantes e anotando para cada componente fracamente conexa as âncoras com posições (índices) mínima e máxima nos dois genomas;
para cada *Componente fracamente conexa de tamanho* $\geq k$ **faça**
    Crie um k-bloco cujas extremidades são determinadas pelas âncoras da componente fracamente conexa que possuem posições mínima e máxima nos dois genomas;
retorna *Conjunto de k-blocos;*

A partir do momento que se introduz a flexibilidade ao método, a possibilidade de ocorrência de conflitos passa a existir. Blocos podem se sobrepor ou diversas possibilidades de encadeamento dentro de um bloco podem existir. Ao invés de adicionar restrições para filtrar tais tipos de conflitos, Lemaitre adotou a estratégia de não resolvê-los e eliminar as regiões que apresentam conflitos. Apesar de parecer radical, tal tática se justifica porque a fase de identificação de blocos de sintenia é apenas a primeira etapa. Durante a etapa de refinamento dos pontos de quebra, informações que eventualmente tenham sido eliminadas por engano podem ser recuperadas. Se a eliminação de âncoras produziu um encurtamento dos blocos, o alinhamento de sequências realizado na fase de refinamento poderá ajudar a recuperar a informação perdida. Contudo, se o bloco inteiro foi eliminado, em função do descarte dos arcos conflitantes, ele não poderá ser recuperado.

Para definir o valor do parâmetro k, Lemaitre realizou testes com um conjunto de genes ortólogos entre o homem e o camundongo obtidos da base de dados `Ensembl` [85]. Utilizando a montagem `NCBI35` (Maio de 2004) do homem e a montagem `NCBI m35` do camundongo (Dezembro de 2005), os genes ortólogos foram determinados através do método da reconciliação de árvores filogenéticas. A lista de genes ortólogos 1-1 foi obtida e as sobreposições entres os genes foram resolvidos da seguinte maneira: se todos os genes que se sobrepõem fossem colineares nos dois genomas, um único gene representando todos eles era criado, caso contrário todos os genes eram eliminados. De um total de 13.557 genes ortólogos 1-1, foram obtidos 12.223 pares de genes (ou grupos de genes) sem sobreposições.

O método de construção de blocos de sintenia (k-blocos) foi aplicado sobre esta lista com valores de k dentro do conjunto $\{1,2,3\}$. A Tabela 5.2 exibe os resultados obtidos. Para $k = 1$, todos os genes foram mantidos pois ele não permite nenhuma flexibilidade. Entre $k = 1$ e $k = 2$ as diferenças entre número de genes e número de *breakpoints* são grandes: o número de pontos de quebra diminui pela metade após eliminação de apenas 205 genes (1,7%). Entre $k = 2$ e $k = 3$ as diferenças são menores: apenas 67 genes e 45 pontos de quebra são eliminados quando usamos $k = 3$. Isso indica que a maior limpeza é feita quando passamos de $k = 1$ para $k = 2$.

Olhando os resultados acima, Lemaitre escolheu o valor $k = 2$ como o mais apropriado para o método. Contudo, antes de formalizar esta decisão, um estudo sobre as características dos blocos e pontos de quebra gerados foi realizado.

Como as âncoras consideradas são genes, acredita-se que seja pouco provável a ocorrência de dois erros de atribuição de ortologia dentro de um mesmo bloco. Para

Tabela 5.2: Resultados obtidos pelo método de construção de k-blocos sobre a lista de pares de genes ortólogos entre homem (montagem `NCBI35`) e camundongo (montagem `NCBI m35`) obtidos do `Ensembl`. O método foi aplicado com valores de k dentro do conjunto {1,2,3}. Para $k = 2$ e 3, os k-blocos colineares e consecutivos nos dois genomas foram agrupados para formar um único bloco.

k	1	2	3
Número de genes ortólogos	12.223	12.018	11.953
Número de blocos de sintenia	786	389	344
Número de *breakpoints*	763	366	321

isso ocorrer, os dois genes errôneos no genoma do homem devem estar próximos e seus supostos ortólogos devem também estar próximo no genoma do camundongo e de forma colinear.

Com $k = 2$, foram obtidos 33 blocos com apenas dois genes e um tamanho médio de 205 kbp. Normalmente erros são esperados em blocos maiores. Destes 33 blocos, 14 quebram um bloco de sintenia maior obtido com $k = 3$. Destes 14, 9 blocos correspondem a uma inversão de dois genes. Como inversões são rearranjos comuns, acredita-se que estes 9 blocos são confiáveis. Além disso, 17 blocos de tamanho 2 se encontram em um ponto de quebra definido pelas fronteiras dos blocos obtidos com $k = 3$. Se estes blocos são realmente ortólogos, a probabilidade de se refinar eficientemente estes *breakpoints*, obtidos com $k = 3$, é baixa. Por estas razões, Lemaitre concluiu que a utilização de $k = 2$ permite a produção de blocos confiáveis.

5.1.2 Refinamento de *breakpoints*

A segunda etapa do processo consiste no refinamento dos pontos de quebra identificados pela primeira etapa. Um ponto de quebra é a região definida entre dois blocos de sintenia que são consecutivos em um genoma mas não são consecutivos em outro ou que não são colineares. Assim, os *breakpoints* são definidos pelas extremidades dos blocos de sintenia que estão em tornos deles. A Figura 5.6 ilustra dois exemplos de pontos de quebra.

Para a realização desta etapa, o método necessita, além da lista de blocos de sintenia identificados na fase anterior, das sequências de DNA dos cromossomos dos

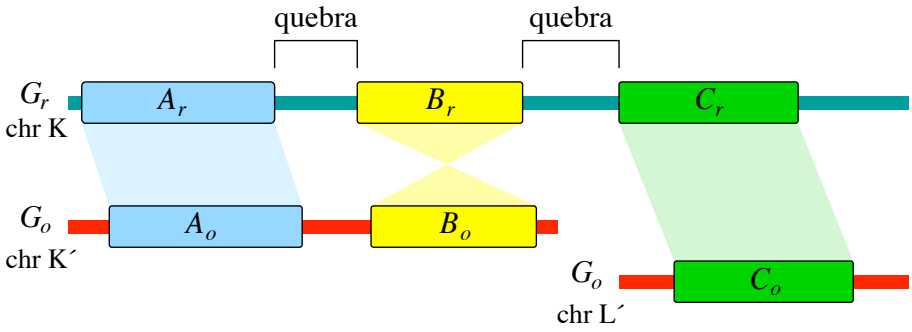


Figura 5.6: Os blocos de sintenia (A_r, A_o), (B_r, B_o) e (C_r, C_o) definem dois *breakpoints*. Os blocos (B_r, B_o) e (C_r, C_o) são consecutivos sobre o genoma G_r mas não são sobre o genoma G_o e assim, podemos definir um ponto de quebra entre eles. Os blocos (A_r, A_o) e (B_r, B_o) são consecutivos sobre os dois genomas, mas não são colineares e, por isso, podemos definir outro *breakpoint* entre eles.

genomas que estão sendo comparados: o genoma de referência G_r e o genoma que está sendo comparado G_o.

Um bloco de sintenia é definido por um par de coordenadas (A_r, A_o) e uma orientação σ_A. A coordenada A_r (A_o) é formada pelo cromossomo onde está o bloco mais as posições de início e fim do bloco neste cromossomo do genoma G_r (G_o).

Relacionadas ao ponto de quebra, são definidas três sequências de interesse: S_r, S_{oA} e S_{oB}. A Figura 5.7 exibe um exemplo onde os blocos (A_r, A_o) e (B_r, B_o) possuem orientações positivas. A sequência S_r é definida entre os blocos A_r e B_r sobre o genoma G_r. A sequência S_{oA} é aquela cuja extremidade 5' faz fronteira com o bloco A_o e cuja extremidade 3' faz fronteira com o próximo bloco situado sobre o mesmo cromossomo no genoma G_o. A sequência S_{oB} é aquela cuja extremidade 3' faz fronteira com o bloco B_o e cuja extremidade 5' faz fronteira com o bloco anterior situado sobre o mesmo cromossomo no genoma G_o.

As definições das sequências S_{oA} e S_{oB} dependem das orientações dos blocos de sintenia (A_r, A_o) e (B_r, B_o). Se a orientação do bloco (A_r, A_o) é negativa, então a sequência S_{oA} possui a sua extremidade 3' fazendo fronteira com o bloco A_o e sua extremidade 5' fazendo fronteira com o bloco anterior a A_o no mesmo cromossomo do genoma G_o. Da mesma maneira, se o bloco (B_r, B_o) possui orientação negativa, a sequência S_{oB} possui sua extremidade 5' fazendo fronteira com o bloco B_o e sua extremidade 3' fazendo fronteira com o próximo bloco no mesmo cromossomo do

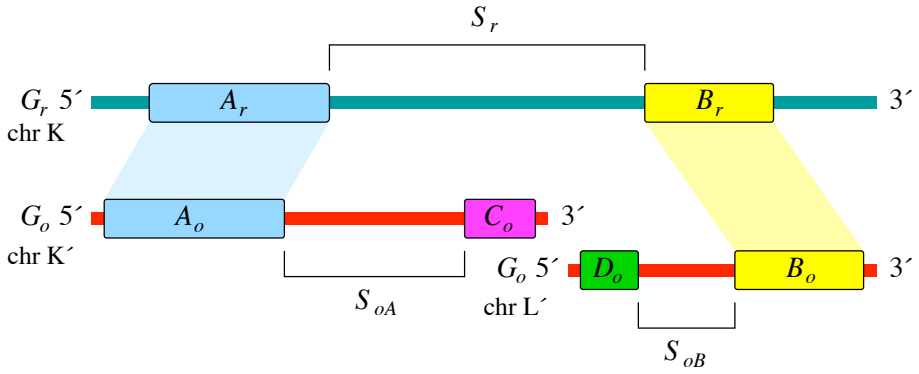


Figura 5.7: Um *breakpoint* possui três sequências de interesse: S_r, S_{oA} e S_{oB}.

genoma G_o.

Como o ponto de quebra ocorreu em alguma posição ao longo da sequência S_r, espera-se que a porção inicial desta sequência (fragmento que faz contato com o bloco A_r) seja ortóloga à porção da sequência S_{oA} que faz contato com o bloco A_o. Analogamente, espera-se que a porção final da sequência S_r (fragmento que faz contato com o bloco B_r) seja ortóloga à porção da sequência S_{oB} que toca o bloco B_o. A Figura 5.8 esquematiza a ocorrência destas regiões ortólogas em relação as sequências S_r, S_{oA} e S_{oB}.

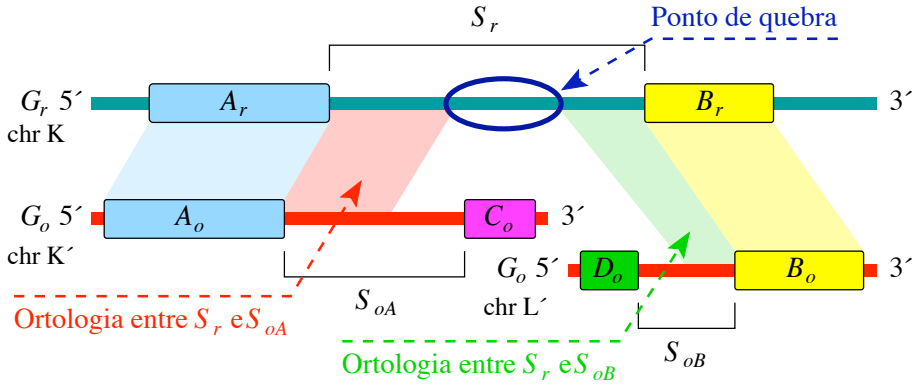


Figura 5.8: Representação esquemática das regiões ortólogas entre as sequências S_r, S_{oA} e S_{oB}

Para avaliar estas regiões, o método criado por Lemaitre realiza três etapas:

1. Detecção de similaridade entre S_r e S_{oA} e entre S_r e S_{oB}

2. Utilização da informação de similaridade para identificação do *breakpoint*

3. Validação estatística do ponto de quebra identificado

Detecção de similaridade

Para avaliar a similaridade entre sequências, diversos algoritmos de alinhamento estão disponíveis e a escolha do método a ser utilizado deve considerar a natureza das sequências e os dados que se espera obter.

As sequências a serem alinhadas (S_r, S_{oA} e S_{oB}) podem conter milhares ou até mesmo milhões de pares de bases. Este fator inviabiliza a utilização de algoritmos exatos e algoritmos que utilizam heurísticas como o `BLAST` [3] podem ser mais apropriados.

O objetivo da realização do alinhamento é identificar fragmentos de S_r que possuem alta similaridade com fragmentos de S_{oA} e S_{oB}. Dessa maneira, algoritmos de alinhamento locais são mais adequados do que os algoritmos de alinhamento global.

Outra razão para a utilização de algoritmos de alinhamento local está na composição das sequências a serem alinhadas. Tais sequências podem conter repetições, pequenos rearranjos, inserções e remoções de DNA e os algoritmos de alinhamento local podem lidar melhor com estes tipos de artefatos.

Lemaitre testou diversos algoritmos de alinhamento local que utilizam heurísticas: `BLAST` [3], `BLASTZ` [144], `CHAOS` [33] e `RepSeek` [1]. Todos este algoritmos se baseiam no princípio de *seed-and-extend* (Seção 4.2.3).

O `BLASTZ` foi o algoritmo que se mostrou mais adequado para a realização dos alinhamentos das sequências S_r, S_{oA} e S_{oB}. Este algoritmo foi desenvolvido para alinhar genomas inteiros, especialmente sequências não codificantes pouco conservadas. Os resultados obtidos por Lemaitre confirmam um estudo, realizado por Sun e Buhler, que mostrou que o `BLASTZ` é, atualmente, o algoritmo mais sensível quando se deseja alinhar sequências não codificantes [150]. Esta característica de sensibilidade, apresentada pelo `BLASTZ`, se explica pelo tipo de semente utilizada por esse algoritmo. Ao invés de utilizar palavras exatas como sementes, como no `BLAST`, o `BLASTZ` adota sementes que possuem tamanho l nas quais apenas algumas posições são fixas. Adicionalmente, o `BLASTZ` autoriza uma transição (mutação de uma base purina para outra base purina ou de uma base pirimídica para uma base pirimídica) no local de uma correspondência (*match*).

Uma observação que precisa ser feita em relação aos algoritmos de alinhamento, incluindo o `BLASTZ`, é que eles possuem inúmeros parâmetros e configurá-los não é uma tarefa trivial. Os valores adotados para os parâmetros dependem das espécies que estão sendo alinhadas e o ajuste deles necessita um estudo mais aprofundado. Neste sentido, `BLASTZ` apresenta uma vantagem. Por ser muito utilizado para o alinhamento de genomas completos, conjuntos de parâmetros para diferentes tipos de espécies estão disponíveis em sites como o do navegador de genomas `UCSC` [67,95,158].

Além de avaliar métodos de alinhamento de sequências, Lemaitre testou uma alternativa que utiliza uma abordagem diferente. O algoritmo `CLASSUS` realiza a contagem de palavras comuns entre duas sequências e é baseado em algoritmos de filtragem de sequências tais como `NIMBUS`, `ED'NIMBUS` e `TUIUIU` [127–129].

`CLASSUS` utiliza o número de k-fatores (palavras de tamanho k) comuns entre duas sequências como medida de similaridade. A contagem se faz em janelas deslizantes de tamanho L. O valor S define a pontuação mínima que uma posição deve ter para que ela faça parte de um *hit*. Este algoritmo tem a vantagem de possuir apenas três parâmetros e de ser muito mais rápido do que o algoritmo `BLASTZ`. Contudo, os testes não mostraram resultados satisfatórios quando comparados com os do algoritmo `BLASTZ` e, por isso, o estudo deste tipo de estratégia foi abandonado.

Sequências intergênicas costumam apresentar elementos repetidos: repetições simples, regiões de baixa complexidade e elementos transponíveis. As repetições simples são repetições em *tandem* de um *motif* curto. As regiões de baixa complexidade são segmentos de sequências que possuem uma composição de nucleotídeos "enviesada" (por exemplo, regiões que apresentam mais de 90% de adeninas). Os elementos transponíveis são regiões "móveis", que têm a capacidade de se deslocar no genoma, trocando de posição ou criando novas cópias ao longo das sequências dos cromossomos.

Estes três tipos de elementos repetitivos dificultam a aplicação de métodos de busca de homologia. Como eles aparecem frequentemente no genoma, diversas regiões diferentes podem apresentar um bom alinhamento com uma região que os contenha. Assim, a similaridade apresentada não serve como um índice de homologia entre as sequências.

Baseado nestas observações, Lemaitre realizou testes comparativos com sequências mascaradas pelo programa `RepeatMasker` [149] e com sequências sem mascaramento. Este programa pode mascarar as repetições simples e as regiões de baixa complexidade. Ele também realiza o mascaramento de elementos transponíveis uti-

lizando uma base de dados de elementos conhecidos.

Normalmente, é desejável que se faça o mascaramento de repetições simples e de regiões de baixa complexidade. Contudo, o mascaramento de elementos transponíveis é uma questão mais delicada. Por exemplo, eles são equivalentes a quase 45% do genoma humano e mascará-los pode resultar em perda de resolução. Além disso, conforme a divergência das espécies comparadas, alguns elementos transponíveis podem ser informativos.

Assim, a escolha de mascarar ou não os elementos transponíveis depende da divergência entre as espécies comparadas. Por exemplo, no caso do homem e do camundongo os testes de Lemaitre mostraram que ao mascarar estes elementos, a cobertura exibida pelos alinhamentos é levemente mais fraca, mas muito menos ruidosa, permitindo melhores análises das sequências comparadas.

Identificação do *breakpoint*

Uma vez que os alinhamentos de S_r com S_{oA} e S_r com S_{oB} foram realizados, podemos analisar a distribuição dos *hits* obtidos ao longo da sequência S_r. A Figura 5.9 exibe um exemplo de representação gráfica da distribuição dos *hits* dos alinhamentos sobre a sequência S_r. Nesta representação, os *hits* do alinhamento S_r contra S_{oA} são pintados de vermelho e os *hits* do alinhamento S_r contra S_{oB} são pintados de verde. Com esta representação, podemos visualizar e repartir os *hits* em zonas de similaridade ao longo da sequência S_r.

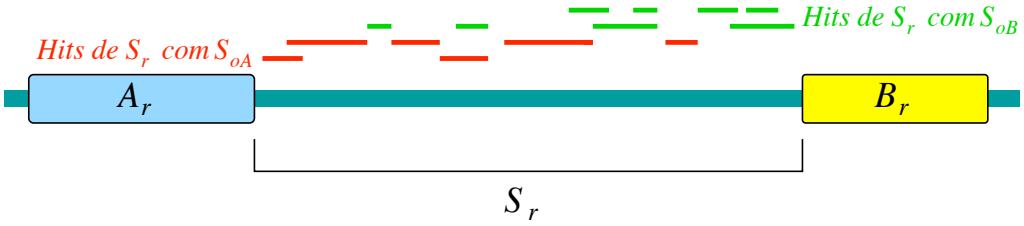


Figura 5.9: Representação gráfica dos *hits* dos alinhamentos de S_r com S_{oA} (vermelho) e S_r com S_{oB} (verde) sobre a sequência S_r.

Supondo que a porção inicial da sequência S_r apresente homologia com a sequência S_{oA} e que a porção final da sequência S_r apresente homologia com a sequência S_{oB}, o esperado é que uma concentração maior de *hits* vermelhos seja observada no início da sequência S_r e que uma concentração maior de *hits* verdes seja visualizada

no final da sequência S_r. Dessa maneira, duas regiões (uma vermelha e outra verde) poderiam ser distinguidas. Um bom *breakpoint* seria capaz de separar estas duas regiões.

Baseado neste tipo de representação, uma função que pontue cada posição k da sequência S_r pode ser definida:

$$s(k) = (e(S_{oA}, k) + d(S_{oB}, k)) - (d(S_{oA}, k) + e(S_{oB}, k)),$$

onde:

- $e(X, k)$ = número de pares de bases da sequência S_r similares à sequência X em posições inferiores ou iguais a k (e = à esquerda de k).
- $d(X, k)$ = número de pares de bases da sequência S_r similares à sequência X em posições superiores a k (d = à direita de k).

Caso a sequência S_r apresente uma distribuição de *hits* onde a porção inicial possui uma maior concentração de *hits* com S_{oA} e uma porção final apresenta uma maior concentração de *hits* com S_{oB}, a função de pontuação $s(k)$ deve apresentar uma curva que tem um caráter crescente no início da sequência (região homóloga a S_{oA}) e uma porção final decrescente (região homóloga a S_{oB}). A Figura 5.10 exibe um exemplo desta curva.

Para identificar o "verdadeiro" *breakpoint*, uma análise da curva da função $s(k)$ pode ser efetuada. Devido ao comportamento de crescimento no início e decrescimento no final, uma estratégia lógica seria a busca pelo ponto ou região onde se encontra o máximo da função $s(k)$. Contudo, antes de adotar esta estratégia, é preciso verificar a característica desta curva quando construída com *hits* de alinhamentos de sequências genômicas.

As sequências alinhadas podem medir milhares de pares de base e contém regiões intergênicas que não são tão conservadas quanto as regiões que possuem genes. Assim, os *hits* dos alinhamentos destas sequências podem apresentar um certo espaçamento entre eles. Devido a estas características, o mais provável é que a curva da função $s(k)$ apresente uma região inicial que cresce em passos (como se fossem degraus de uma escada), uma região de platô no centro onde os valores ficam flutuando próximo do valor máximo e, por fim, uma região que decresce em passos. Portanto, procurar apenas pelo valor máximo nesta curva não garante que o *breakpoints* está sendo efetivamente identificado.

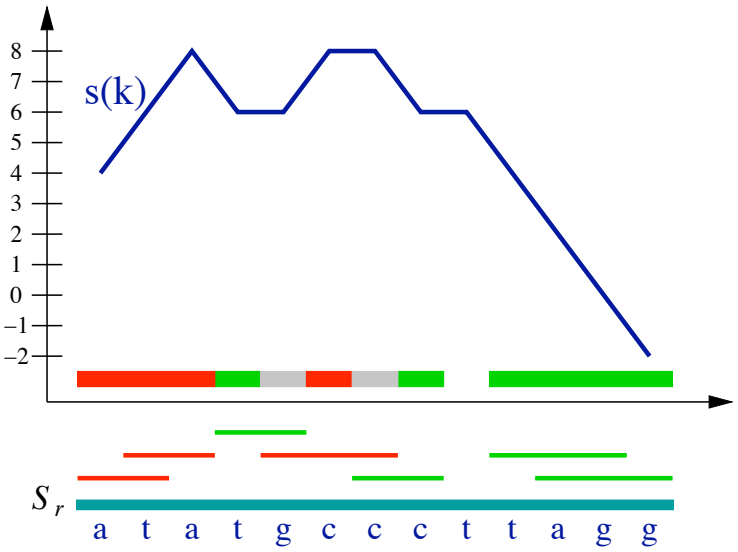


Figura 5.10: Exemplo gráfico de curva da função de pontuação $s(k)$. Dentro do gráfico, as barras vermelhas exibem as posições onde existem apenas *hits* de S_r com S_{oA} e as barras verdes apontam as posições onde existem apenas *hits* de S_r com S_{oB}. As barras cinzas indicam as posições onde S_r teve *hits* com as duas sequências. Finalmente, o espaço em branco exibe uma posição onde S_r não teve *hit*.

Para tratar este problema, a estratégia desenvolvida por Lemaitre baseou-se em métodos para resolução do problema de detecção de ruptura de sinal. Estes métodos buscam pela melhor partição do sinal em segmentos. Dentro de um segmento as características do sinal são homogêneas e entre os segmentos, as características observadas são diferentes.

Nesta abordagem, a sequência S_r é convertida em uma sequência numérica I. O tamanho de I é igual a n (tamanho de S_r) e cada posição k da sequência S_r está associado à posição I_k. A conversão é feita da seguinte forma:

- I_k recebe $+1$ se a posição k é coberta por pelo menos um *hit* com S_{oA} e nenhum *hit* de S_{oB}
- I_k recebe -1 se a posição k é coberta por pelo menos um *hit* com S_{oB} e nenhum *hit* de S_{oA}
- I_k recebe 0 se a posição k é coberta por nenhum hit ou é coberta por *hits* com S_{oA} e com S_{oB}

Esta conversão é similar a realizada pela função $s(k)$ pois ela dá um peso positivo ou negativo de acordo com a origem das sequências. De fato, $s(x) = 2\Sigma_{k=1}^{x} I_k + C$ (com $C = -\sum_{k=1}^{n} I_k$).

A sequência I pode ser tratada com métodos de detecção de rupturas para segmentá-la em regiões. Uma abordagem possível seria a realização de diversas segmentações para produção de conjuntos com diferentes números de segmentos. Feito isso, uma análise dos diferentes conjuntos escolheria qual a melhor segmentação.

Rapidamente é possível ver que esta abordagem não é viável devido a complexidade de se analisar os diferentes conjuntos: seria preciso definir um método de comparação, um modo de relacionar os diferentes segmentos com a sua devida origem (homóloga de S_{oA}, homóloga de S_{oB} ou ponto de quebra), entre outras dificuldades.

A abordagem mais simples é a de utilizar a informação disponível *a priori*. A sequência S_r possui grande probabilidade de estar dividida em três regiões distintas e, assim, o método de ruptura de sinal pode ser aplicado para dividir a sequência I em três segmentos (ou menos, caso um dos segmentos seja nulo).

O primeiro segmento deverá apresentar uma média de pontuação positiva pois ele representa a região homóloga a S_{oA} e, por isso, deve conter mais *hits* vermelhos do que verdes. Analogamente, o último segmento deverá mostrar uma média de pontuação negativa pois é homólogo a S_{oB} e deve conter mais *hits* verdes do que vermelhos.

Em relação ao segmento central, as propriedades que ele deve apresentar não são claras. Contudo, para que ele possa ser diferenciado dos outros dois segmentos, ele é modelado pelo valor nulo. Dessa maneira, o modelo é fixado no segmento do meio e as restrições de sinal são impostas aos outros dois segmentos.

De maneira mais formal, o modelo é escrito da seguinte maneira:

$$I_t = s(t) + \epsilon_t, t = 1..n$$

A função s é uma função constante por partes definida por dois pontos de ruptura u_1 e u_2, tal que $s(t) = s_j$ para $u_{j-1} < t \leq u_j$ ($0 = u_0 < u_1 < u_2 < u_3 = n$). Os valores s_1, s_2 e s_3 são definidos por:

- $s_1 = \begin{cases} \frac{\sum_{t=1}^{u_1} I_t}{u_1} & \text{se } \sum_{t=1}^{u_1} I_t > 0 \\ \infty & \text{caso contrário} \end{cases}$
- $s_2 = 0$

- $s_3 = \begin{cases} \frac{\sum_{t=u_2+1}^{n} I_t}{n-u_2} & \text{se } \sum_{t=u_2+1}^{n} I_t < 0 \\ \infty & \text{caso contrário} \end{cases}$

O objetivo é encontrar um par de valores (u_1, u_2) que minimizem a função de contraste (erro quadrático):

$$RSS(u_1, u_2) = \sum_{j=1}^{3} \sum_{t=u_{j-1}+1}^{u_j} (I_t - s_j)^2 = \sum_{t=1}^{u_1} (I_t - s_1)^2 + \sum_{t=u_1+1}^{u_2} I_t^2 + \sum_{t=u_2+1}^{n} (I_t - s_3)^2 \tag{5.1}$$

Para autorizar menos do que 3 segmentos, basta autorizar o valor 0 para u_1 (primeiro segmento nulo), o valor n para u_2 (terceiro segmento nulo) e autorizar $u_1 = u_2$ (segmento central nulo).

No caso geral, um algoritmo de programação dinâmica com complexidade $O(n^2)$ pode ser utilizado para segmentar a sequência [5]. No entanto, como o modelo restringe a segmentação a no máximo 3 segmentos, ele é mais simples que o caso geral. Assim, buscar pelos pontos de ruptura é uma tarefa mais fácil, bastando maximizar duas somas independentemente. Para mostrar esse resultado, Lemaitre apresentou e provou o Lema 5.1.1. Com estas observações, a complexidade do algoritmo caiu para $O(n)$.

Lema 5.1.1. *Seja uma sequência I_k de comprimento n, tal que para todo $k \in \{1, n\}$, $I_k \in \{-1, 0, 1\}$, as posições u_1 e u_2 que minimizam a função de contraste $RSS(u_1, u_2)$ (Equação 5.1) são tais que:*

- $\frac{1}{\sqrt{u_1}} \sum_{t=1}^{u_1} I_t$ *é maximal*
- $\frac{1}{\sqrt{n-u_2}} \sum_{t=u_2+1}^{n} I_t$ *é minimal*

Validação estatística

Independentemente da estrutura apresentada pela sequência I, o método de segmentação irá produzir a melhor segmentação em até três segmentos. Devido a isso, é de extrema importância verificar se os dados realmente estão estruturados em três segmentos, respeitando as restrições impostas pelo modelo, ou se os dados não estão estruturados. Neste último caso, os pontos de ruptura obtidos não fazem sentido

como solução para o problema proposto e, assim, deve-se concluir que não é possível refinar o *breakpoint* com os dados dos alinhamentos.

Quanto maior for a estruturação da sequência I em três segmentos que respeitam as condições impostas, menor será o valor minimizado da função de contraste, indicando um melhor ajuste. A validação estatística consiste em avaliar se o ajuste obtido é melhor do que os que seriam obtidos em uma sequência não estruturada.

Para isso, sequências não estruturadas são criadas a partir da sequência I. Os valores de I são permutados e o método de segmentação é executado sobre as novas sequências. Como a sequência I representa *hits* em um alinhamento, as posições não são independentes umas das outras. Assim, ao invés de permutar as posições individualmente, a permutação é feita considerando os blocos de valores idênticos (blocos de valores 0, +1 e −1).

A hipótese nula de que I não é estruturada em três segmentos que respeitam as condições impostas pelo modelo é aceita se mais de 5% das sequências, criadas a partir da permutação dos blocos, tiverem um valor de ajuste menor do que o apresentado por I.

5.2 Pacote `Cassis`

Durante o desenvolvimento de seu estudo de Doutorado, Lemaitre desenvolveu a metodologia apresentada na seção anterior e realizou diversos testes comparando diferentes espécies de mamíferos [100].

Para este estudo, uma série de *scripts* em `R` foram escritos para execução das tarefas intermediárias. No entanto, eles não eram interligados e alguns deles não podiam ser parametrizados exigindo intervenção no código fonte para realização das mudanças exigidas pelo par de espécies que estivesse sendo analisado.

Um de nossos objetivos nesta tese é a realização de uma análise das características apresentadas pelas regiões dos pontos de quebra e, para isso, é desejável a obtenção de um conjunto confiável de *breakpoints*. Como o método proposto por Lemaitre utiliza marcadores ortólogos 1-1 para definir os *breakpoints* sem sobreposição e com uma melhor resolução, optamos por realizar a implementação deste método.

Outra razão para explicar a necessidade de implementação do método, está no desejo de utilizar informações mais atualizadas. Em seu trabalho, ao comparar os genomas do homem (*Homo sapiens*) com o do camundongo (*Mus musculus*), Lemaitre

utilizou dados obtidos da versão número 24 do `Ensembl` [85] que correspondiam às montagens `NCBI35` do genoma humano e `NCBI m35` do genoma do camundongo. No momento da realização do nosso estudo, a base de dados do `Ensembl` estava em sua versão número 56, contendo dados das montagens `GRCh37` do homem, originária do *Genome Reference Consortium*, e `NCBI m37` do camundongo, proveniente do *Mouse Genome Sequencing Consortium*.

Implementamos os métodos propostos por Lemaitre e produzimos um pacote de programas denominado `Cassis`. Nesta seção detalharemos alguns aspectos da implementação deste pacote e mostraremos alguns resultados produzidos por ele na análise comparativa dos genomas do homem e do camundongo.

5.2.1 Implementação do pacote

O pacote `Cassis` foi desenvolvido em `Perl` [46] e em `R` [138] e testado em ambiente `Linux` e `Mac OS`.

Composto por uma série de *scripts*, ele foi projetado tanto para atender as necessidades do usuário comum, que apenas deseja aplicar o método, como para atender usuários avançados, que realizam testes de parâmetros ou, até mesmo, montam suas próprias metodologias de identificação e refinamento de *breakpoints*.

O pacote possui um *script* principal denominado `cassis.pl` que coordena todas as etapas exigidas pelo processo: identificação de *breakpoints*, criação e alinhamento das sequências S_r, S_{oA} e S_{oB}, segmentação e validação estatística. Ele tem o papel de validar os dados e parâmetros de entrada e chamar os *scripts* responsáveis por cada etapa.

Para simplificar a sua utilização, `cassis.pl` possui apenas 5 parâmetros obrigatórios referentes aos dados de entrada (tabela de genes ortólogos, tipo da tabela e localização das sequências dos cromossomos das duas espécies) e aos dados de saída (localização onde ele deve escrever os resultados). Os parâmetros para configuração do método são todos opcionais e os seus valores padrões são os mesmos que foram utilizado por Lemaitre para comparação dos genomas do homem e do camundongo.

Os demais *scripts* do pacote são responsáveis pela execução das diferentes etapas do processos e são divididos em dois grupos: *scripts* para processamento de grupos de sequências e *scripts* para realização de atividades atômicas como, por exemplo, alinhamento de um par de sequências, segmentação de uma sequência, etc. Esta modularização do código do pacote permite que um usuário avançado possa criar seu

próprio esquema de identificação e refinamento de *breakpoints* através da alteração de nossos *scripts* ou, até mesmo, através da substituição deles por suas próprias soluções.

O *script* `cassis.pl` coordena o seguinte fluxo de atividades:

1. Verificação dos dados de entrada
2. Identificação dos *breakpoints* com base na tabela de genes ortólogos
3. Criação dos arquivos `FASTA` das sequências S_r, S_{oA} e S_{oB}
4. Alinhamento das sequências S_r, S_{oA} e S_{oB}
5. Criação de *dotplots* mostrando a distribuição dos *hits* ao longo da sequência S_r
6. Segmentação dos *breakpoints* e validação estatística

Verificação dos dados de entrada

Nesta fase `Cassis` faz uma série de verificações básicas: formato do arquivo que contém a tabela com os dados de ortologia, existência de arquivos FASTA com as sequências dos cromossomos para os dois genomas, verificação dos valores de parâmetros, etc.

Identificação de *breakpoints*

Para a etapa de identificação de *breakpoints*, o *script* trabalha apenas com tabelas de genes ortólogos 1-1. Qualquer par de genes que não respeite esta condições é descartado. Antes de realizar a identificação de *breakpoints*, um processo de eliminação de genes que se sobrepõem é executada. Durante este processo, conjuntos de genes que se sobrepõem, mas são colineares e consecutivos em ambos os genomas, são unidos para formar um único gene. Caso contrário, eles são eliminados.

A tabela de genes ortólogos 1-1 sem sobreposições é então processada para identificação dos blocos de sintenia segundo o Algoritmo 5.1.1. Na realidade, baseado nos resultados de Lemaitre, que mostram que a adoção de $k = 2$ é o melhor compromisso entre flexibilidade e sensibilidade (veja Tabela 5.2), decidimos implementar uma versão simplificada deste algoritmo que fixa $k = 2$.

Criação dos arquivos FASTA

Como consequência da identificação dos blocos de sintenia, temos as coordenadas de começo e fim das sequências S_r, S_{oA} e S_{oB}. Assim, podemos criar os arquivos FASTA de cada uma delas para a realização da etapa de alinhamento.

Normalmente, as sequências S_r, S_{oA} e S_{oB} podem ser criadas conforme o esquema exibido na Figura 5.7. Contudo, testes realizados por Lemaitre mostraram que melhores alinhamentos podem ser obtidos quando utilizamos as versões estendidas destas sequências.

Além da região localizada entre os blocos A_r e B_r, a sequência S_r inclui o último gene do bloco A_r e o primeiro gene do bloco B_r. De maneira análoga, a sequência S_{oA} (S_{oB}) inclui o primeiro/último gene dos blocos A_o (B_{oB}) e C_o (D_o). Um esquema das versões estendidas destas sequências pode ser visto na Figura 5.11.

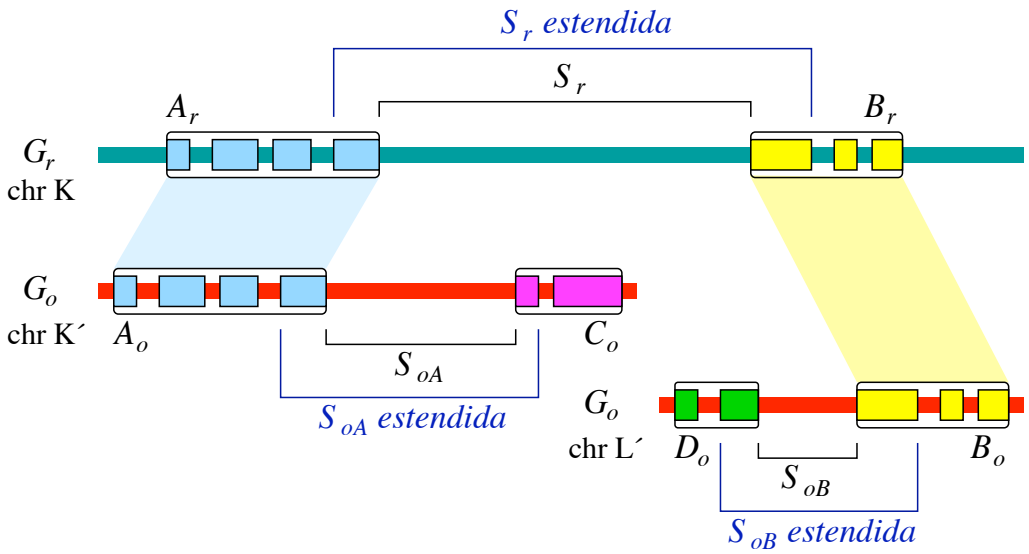


Figura 5.11: Versões estendidas das sequências S_r, S_{oA} e S_{oB}. As versões estendidas destas sequências incluem o primeiro/último gene dos blocos de sintenia que as definem.

A adição das sequências dos genes auxilia os algoritmos de alinhamento a encontrarem as regiões homólogas nas sequências, especialmente nas bordas das versão não estendida da sequência S_r. Com melhores alinhamentos, melhores resultados podem ser obtidos pelo processo de segmentação.

O pacote `Cassis` adota a extensão das sequências S_r, S_{oA} e S_{oB} como procedimento padrão. Contudo, parâmetros do *script* `cassis.pl` permitem que o usuário desative este procedimento.

Alinhamento das sequências

O alinhamento das sequências é feito utilizando o programa `LASTZ` [79]. Entre outros algoritmos, este programa implementa o algoritmo do `BLASTZ`. Além disso, ele possui correções de erros contidos na última versão do `BLASTZ`, que foi descontinuado.

`Cassis` oferece três opções de conjuntos de parâmetros para realização de alinhamentos. O primeiro conjunto é adequado ao alinhamento de espécies próximas como, por exemplo, o homem (*Homo sapiens*) e o macaco-rhesus (*Macaca mulatta*). O segundo conjunto é o padrão, e é adequado para alinhamento de espécies com distância intermediária como, por exemplo, o homem e o camundongo (*Mus musculus*). Finalmente, a terceira opção é formada por parâmetros utilizados em alinhamentos de espécies distantes como, por exemplo, o homem e o galo (*Gallus gallus*).

Para facilitar o uso, o `Cassis` recebe como parâmetro apenas um número indicando o nível desejado. Isso reduz a capacidade de parametrização do *script*. Contudo, caso um usuário avançado deseje utilizar outras configurações, o código pode ser facilmente alterado para adição de novas opções.

Criação de *dotplots*

Esta fase não é essencial ao processo de refinamento. Ela é realizada apenas para produção de gráficos que auxiliem a análise dos dados produzidos pelo pacote. A Figura 5.12 exibe um exemplo de gráfico de *dotplot* construído para um ponto de quebra identificado entre o homem e o camundongo.

Segmentação e validação estatística

Os *hits* dos alinhamentos de S_r com S_{oA} e S_r com S_{oB} são utilizados no processo de segmentação que é responsável pelo refinamento do *breakpoint*. O pacote produz as novas coordenadas do *breakpoint* e atribui a ele um valor de *status* que indica se ele passou ou não no teste estatístico. Além disso, o método também produz um gráfico com a representação gráfica da segmentação realizada. A Figura 5.13 exibe um exemplo deste tipo de gráfico.

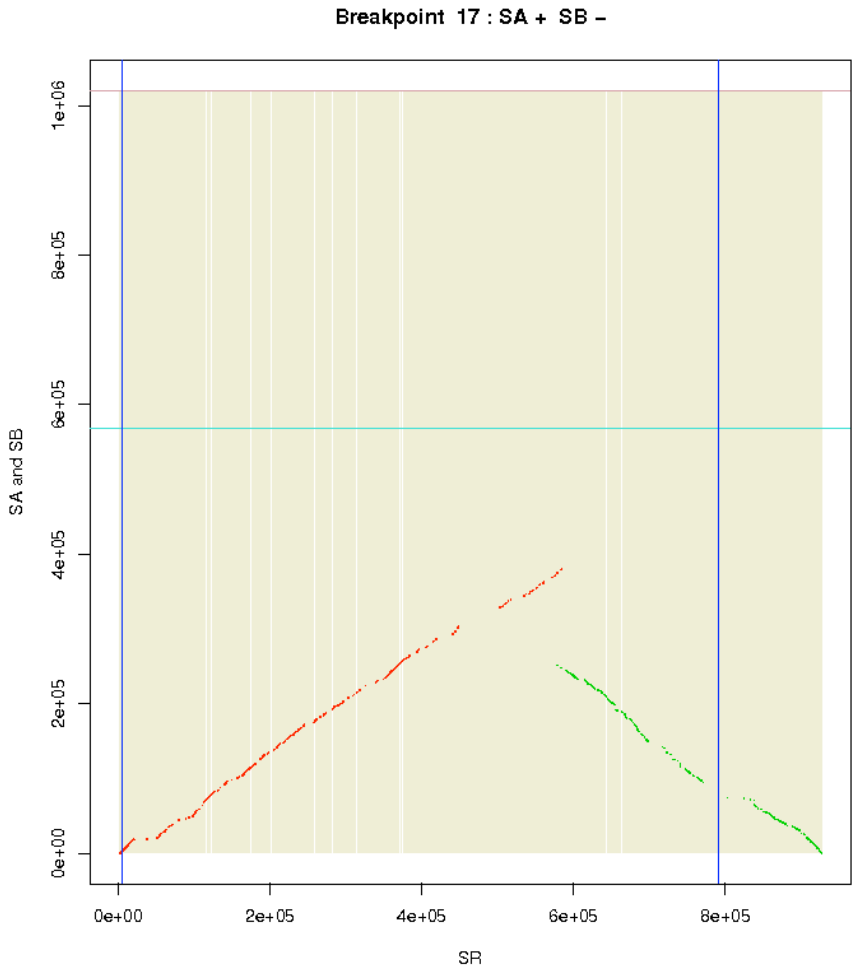


Figura 5.12: Exemplo de gráfico de *dotplot* produzido pelo pacote `Cassis`. Os segmentos vermelhos e verdes indicam as localizações dos *hits* obtidos no alinhamento de S_r com, respectivamente, S_{oA} e S_{oB}. As regiões pintadas de amarelo claro indicam trechos em que a sequência S_r teve suas repetições mascaradas. As linhas verticais azuis indicam as coordenadas das bordas do ponto de quebra antes da segmentação. As linhas horizontais rosa e verde claro indicam, respectivamente, os limites das sequências S_{oA} e S_{oB}.

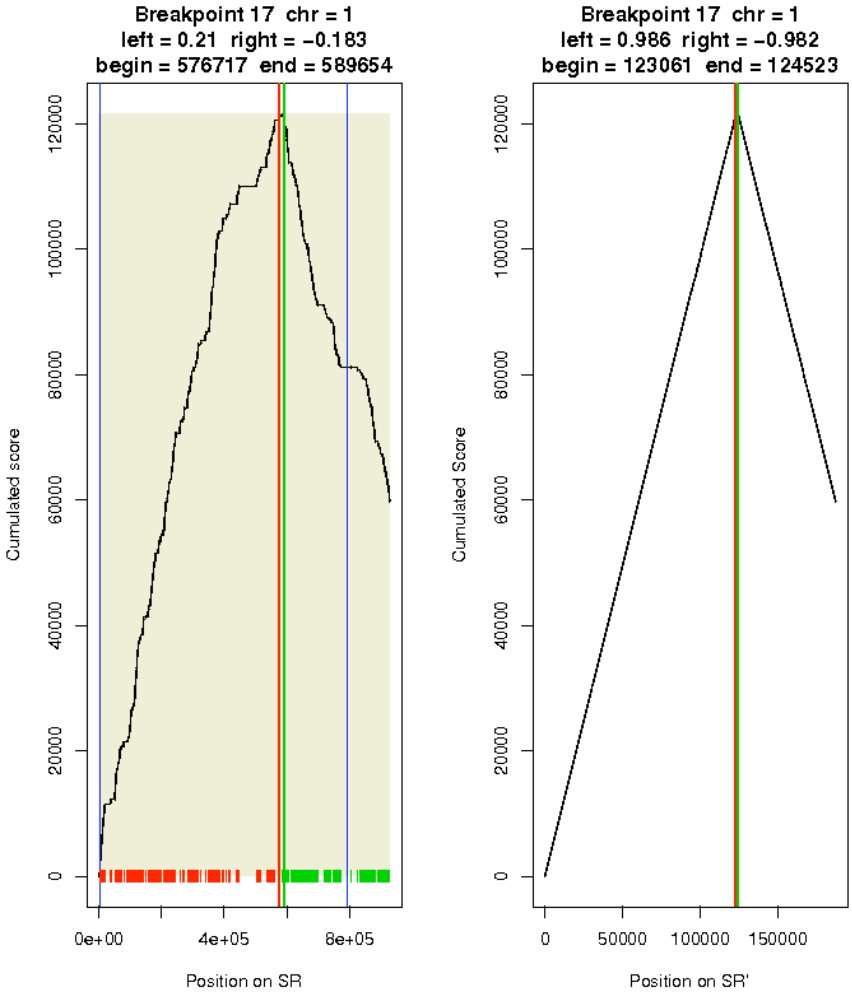


Figura 5.13: Exemplo de gráfico de segmentação produzido pelo pacote `Cassis`. O gráfico da esquerda mostra a curva da soma acumulada dos valores (0, +1 e −1) para cada posição da sequência S_r estendida. As pequenas barras vermelhas e verdes na porção inferior do gráfico denotam a composição dos *hits* na região observada (S_{oA} e S_{oB}, respectivamente). As linhas verticais azuis indicam as coordenadas das bordas do ponto de quebra antes da segmentação. As linhas verticais vermelha e verde indicam as novas coordenadas de início e fim do *breakpoint* após o refinamento. As regiões pintadas de amarelo claro indicam trechos em que a sequência S_r teve suas repetições mascaradas. O gráfico da direita representa a mesma curva quando construída sobre a sequência S'_r que é obtida através do descarte de todas posições de S_r que não possuem *hits*.

Utilizando blocos de sintenia de outros métodos

Para aumentar a flexibilidade do pacote, adicionamos a ele a opção de receber uma lista de blocos de sintenia pré-calculados de acordo com algum outro método utilizado pelo usuário. Assim, ao invés de realizar uma etapa de delimitação dos blocos de sintenia, o pacote realiza diretamente a definição das coordenadas das sequências S_r, S_{oA} e S_{oB}.

Como apenas as coordenadas dos blocos estão disponíveis, a extensão das sequências é feita através da adição de um fragmento de tamanho arbitrário da borda dos blocos que delimitam as sequências. O valor padrão foi definido em 50 kbp, que é próximo ao valor médio que observamos para o tamanho dos genes que aparecem nas extremidades dos blocos de sintenia, identificados pelo nosso método, entre o homem e o camundongo.

O nosso método de construção de blocos de sintenia garante a produção de blocos que não se sobrepõem. Contudo, blocos produzidos por outros métodos podem não apresentar esta característica. Simplesmente descartar os blocos que apresentam sobreposição pode ser uma decisão muito dura. Assim, esta decisão é adiada para o momento em que as coordenadas das sequências S_r, S_{oA} e S_{oB} são definidas. Caso uma destas sequências, após a extensão, tenha tamanho menor do que 50 kbp, consideramos que a sobreposição entre os dois blocos, que definem a sequência, é grande demais e o candidato a *breakpoint* é descartado da análise.

Mascaramento de repetições

`Cassis` não inclui, em seu conjunto de procedimentos, a etapa de mascaramento de repetições, que é uma atividade que exige um tempo muito grande de processamento.

Apesar disso, a utilização de sequências de cromossomos que tiveram suas repetições mascaradas é extremamente recomendável. Os alinhamentos serão mais rápidos e os resultados obtidos por eles serão mais significativos.

A recomendação dada ao usuário é que ele faça o mascaramento ou obtenha de base de dados como o `Ensembl`, as sequências dos cromossomos já processadas pelo programa `RepeatMasker` [149]. Porém, caso ele opte por realizar o mascaramento, o pacote contém um *script* auxiliar, que utiliza o `RepeatMasker` para a execução desta tarefa.

Disponibilidade

O código do pacote Cassis é distribuído sob licensa GNU GPL License e está disponível no endereço http://pbil.univ-lyon1.fr/software/Cassis/. Além do código, o usuário poderá encontrar a documentação que explica o modo de utilização, os parâmetros de entrada e saída e a organização do pacote como um todo.

Um *Application Notes*, apresentado o pacote, foi publicado na revista *Bioinformatics* sob o título "Cassis: precise detection of genomic rearrangement breakpoints" [16].

5.2.2 Comparação dos genomas do homem e do camundongo

Como mencionado anteriormente, um dos motivos da implementação do pacote Cassis era a realização de uma análise entre os genomas do homem e do camundongo com um conjunto de dados mais recente do que o utilizado por Lemaitre.

Esta análise foi feita com dados obtidos do Ensembl em sua versão número 56, disponibilizada em Setembro de 2009. Esta versão continha a montagem GRCh37 do genoma humano (*Genome Reference Consortium* – Fevereiro de 2009) e a montagem NCBI m37 (*Mouse Genome Sequencing Consortium* – Abril de 2007).

As sequências FASTA de todos os cromossomos, com exceção do cromossomo Y, foram obtidas para os dois genomas. As sequências, que foram recuperadas do *site* FTP do Ensembl, estavam com suas repetições mascaradas.

A ferramenta Biomart [148] disponível na página do Ensembl foi utilizada para recuperação da lista de genes ortólogos entre o homem e o camundongo. Como descrito na Seção 4.2.2, a lista de genes ortólogos de Ensembl é construída com métodos que avaliam a máxima verossimilhança entre árvores filogenéticas [162].

Cassis foi utilizado para processar a lista de 15.047 pares de genes ortólogos 1-1 obtidos. A primeira fase identificou um total de 369 pontos de quebra que foram submetidos ao processo de segmentação. Destes, apenas 4 *breakpoints* (1,08%) não puderam ser refinados (falharam na verificação estatística e receberam $status = 0$). A Tabela 5.3 exibe as características dos *breakpoints* obtidos antes e depois do processo de segmentação.

O processo de refinamento de *breakpoints* do pacote Cassis procura pela melhor segmentação da sequência de valores 0, $+1$ e -1 (correspondentes aos *hits* dos alinhamentos de S_r com S_{oA} e S_r com S_{oB} ao longo de S_r) em 3 segmentos que

Tabela 5.3: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos *breakpoints* identificados entre o homem e o camundongo pelo programa `Cassis` ao processar a lista de pares de genes ortólogos obtida do `Ensembl`. Dos 369 *breakpoints* identificados, 365 passaram na verificação estatística ($status = 1$) e 4 não passaram ($status = 0$).

Descrição	**N**	**Minímo**	**Máximo**	**Média**	**Mediana**
Antes da segmentação					
Todos	369 (100%)	1	32.752.838	771.401	244.568
$Status = 1$	365 (98,92%)	1	32.752.838	775.512	244.568
$Status = 0$	4 (1,08%)	79.048	1.031.931	396.243	236.998
Depois da segmentação					
Todos	369 (100%)	21	5.133.352	201.251	52.986
$Status = 1$	365 (98,92%)	21	5.133.352	203.057	53.262
$Status = 0$	4 (1,08%)	2.069	72.226	36.459	35.771

respeitem as condições pré-estabelecidas: primeiro segmento homólogo a S_{oA}, segundo segmento referente ao ponto de quebra e terceiro segmento homólogo a S_{oB}. Contudo, não existe garantia que a segmentação produza uma região de *breakpoint* menor do que a original. Dos 365 *breakpoints* que passaram na verificação estatística, um total de 25 pontos de quebra (6,85%) apresentaram aumento de comprimento após a segmentação. Assim, a maioria dos *breakpoints* (93,15%) apresentou algum refinamento. A Tabela 5.4 exibe uma comparação entre os valores mínimo, máximo, média e mediana observados nos conjuntos de pontos de quebra que passaram na verificação estatística e tiveram seus comprimentos aumentados ou diminuídos.

A Tabela 5.4 também exibe os valores mínimo, máximo, médio e a mediana das porcentagens de refinamento observadas nos 340 *breakpoints* que tiveram comprimento reduzido. Em média, a redução apresentada é de 62,26% mostrando que o método implementado por `Cassis` é realmente capaz de melhorar a resolução dos pontos de quebra identificados.

A Figura 5.14 mostra o histograma de distribuição das diferenças de tamanhos observadas nos *breakpoints* antes e depois da segmentação. O gráfico considera 358 dos 365 pontos de quebra que passaram na verificação estatística. Por apresentarem diferenças maiores do que 5 Mbp, os demais *breakpoints* foram removidos com o

Tabela 5.4: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos *breakpoints*, identificados entre o homem e o camundongo (`Ensembl`), que passaram na verificação estatística. Dos 365 *breakpoints*, 25 tiveram os seus comprimentos aumentados após a segmentação ($Depois \geq Antes$) e 340 tiveram seus comprimentos reduzidos ($Depois < Antes$). A tabela também exibe o refinamento (redução de comprimento) observado nestes 340 *breakpoints*.

Descrição	N	Minímo	Máximo	Média	Mediana
Antes da segmentação					
$Depois \geq Antes$	25 (6,85%)	1	5.115.573	282.827	34.067
$Depois < Antes$	340 (93,15%)	303	32.752.838	811.739	256.577
Depois da segmentação					
$Depois \geq Antes$	25 (6,85%)	1.539	5.133.352	311.471	51.645
$Depois < Antes$	340 (93,15%)	21	4.860.908	195.085	54.883
Redução de comprimento [%]					
$Depois < Antes$	340 (93,15%)	0,31	99,99	62,26	71,20

propósito de se obter uma melhor visualização.

Analisando o gráfico podemos ver que a maior parte dos *breakpoints* que tiveram aumento no tamanho sofreram um pequeno acréscimo no comprimento. Em relação aos pontos de quebra que tiveram tamanhos reduzidos, a maioria mostrou diferenças de tamanho menores do que 1 Mbp. No entanto, alguns *breakpoints* tiveram grande redução apresentando diferenças maiores do que 2 Mbp.

5.2.3 Utilizando `Cassis` com blocos de sintenia

`Cassis` permite a execução da metodologia de segmentação em *breakpoints* obtidos através da análise de blocos de sintenia produzidos por outros métodos. Nesta seção, apresentaremos os resultados obtidos com dois conjuntos diferentes de blocos de sintenia observados entre o homem e o camundongo: blocos obtidos da base de dados `Compara` da plataforma `Ensembl` e blocos obtidos com o algoritmo `MAUVE`.

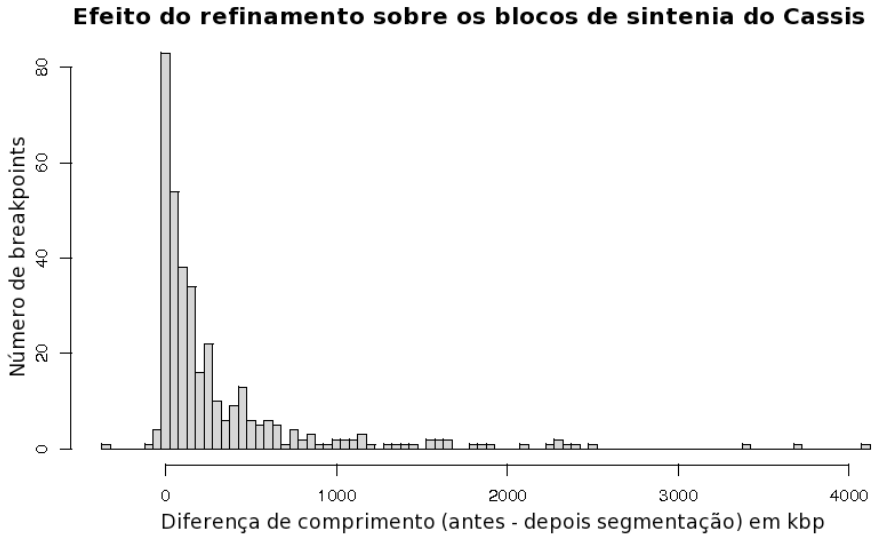


Figura 5.14: Histograma de distribuição das diferenças de tamanhos observadas nos *breakpoints*, produzidos por `Cassis` com os pares de genes ortólogos do homem e do camundongo obtidos do `Ensembl`, antes e depois da segmentação. O gráfico considera 358 *breakpoints* que passaram pela validação estatística (7 *breakpoints* com diferenças maiores do que 5 Mbp foram removidos para facilitar a visualização).

`Compara`

A base de dados `Compara` da plataforma `Ensembl` contém o registro dos blocos de sintenia observados entre o homem e algumas espécies, entre elas, o camundongo.

Estes blocos de sintenia são construídos com uma metodologia que possui duas etapas. Na primeira etapa o programa `BLASTZ-NET` (`CHAINNET`) é utilizado. Ele realiza alinhamentos entre as sequências das espécies para definição de âncoras que são encadeadas de acordo com a localização delas em ambos os genomas. Feito isso, a melhor subcadeia em cada região é escolhida para a espécie de referência definindo assim um bloco de sintenia. Na segunda etapa, um processo de reagrupamento dos blocos, que é dividido em duas fases, é feito levando em consideração a distância entre os blocos. Na primeira fase, blocos de sintenia que estão a menos de 200 kbp são agrupados. Na fase seguinte, dois grupos que estão em sintenia são ligados se não existem mais de 2 blocos entre eles, que não apresentam sintenia com eles, e se eles estão a menos de 3 Mbp um do outro [94].

Os blocos de sintenia entre o homem e o camundongo foram obtidos da base de dados do `Compara` da plataforma `Ensembl`. Um total de 345 blocos foram obtidos e os valores mínimo, máximo, média e mediana dos tamanhos dos blocos nos dois genomas são exibidos pela Tabela 5.5.

Tabela 5.5: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos blocos de sintenia entre o homem e o camundongo obtidos da base de dados `Compara` do `Ensembl`.

Descrição	**Minímo**	**Máximo**	**Média**	**Mediana**
Homo sapiens	108.946	58.118.452	7.844.447	3.697.691
Mus musculus	102.552	52.624.330	7.077.399	3.172.511

Os blocos foram processados pelo pacote `Cassis` e 292 *breakpoints* foram identificados. Deste total, 234 pontos de quebra foram processados pela fase de segmentação. O restante dos *breakpoints* foram descartados por possuírem sequências menores do que o tamanho limite (50 kbp). Isto ocorreu devido a ocorrência de sobreposição entre os blocos de sintenia. Em todos os casos, as sobreposições ocorreram no genoma do camundongo e isso é explicado pela metodologia utilizada pelo `Ensembl` para construção de blocos de sintenia. A Tabela 5.6 mostra o número de *breakpoints* identificados conforme o *status* que receberam do programa `Cassis`.

Tabela 5.6: Número de *breakpoints* identificados pelo `Cassis` ao processar a lista de blocos de sintenia entre homem e camundongo da base de dados `Compara` do `Ensembl`. No total, foram identificados 292 *breakpoints*. Destes, 58 foram descartados ($status < 0$) por possuírem sequências com tamanho inferior ao limite de 50 kbp.

Descrição	***status***	**N**
S_r, S_{oA} e S_{oB} com tamanhos apropriados	1	234
S_{oA} com tamanho menor do que o limite	-3	27
S_{oB} com tamanho menor do que o limite	-4	22
S_{oA} e S_{oB} com tamanho menor do que o limite	-5	9

Os 234 pontos de quebra que possuem sequências S_r, S_{oA} e S_{oB} com tamanhos maiores do que o limite mínimo foram submetidas ao processo de segmentação e 207

(88,46%) passaram na verificação estatística ($status = 1$). Entre os 27 restantes, 26 (11,11%) não tiveram sucesso na verificação estatística ($status = 0$) e um não pode ser segmentado por que nenhum *hit* foi produzido no alinhamento de suas sequências. A Tabela 5.7 mostra o valores mínimo, máximo, média e mediana dos comprimentos dos pontos de quebra antes e após o refinamento.

Tabela 5.7: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos *breakpoints* identificados entre o homem e o camundongo pelo programa `Cassis` ao processar a lista de blocos de sintenia da base de dados `Compara` do `Ensembl`. Dos 234 *breakpoints* que foram identificados e tinham sequências com tamanhos apropriados, 207 passaram na verificação estatística ($status = 1$), 26 não passaram ($status = 0$) e um não pode ser refinado porque não apresentou *hits* em seus alinhamentos ($status = -1$).

Descrição	**N**	**Minímo**	**Máximo**	**Média**	**Mediana**
Antes da segmentação					
Todos	234 (100%)	1	15.172.525	463.422	114.679
$Status = 1$	207 (88,46%)	1	15.172.525	503.025	124.520
$Status = 0$	26 (11,11%)	1.386	521.357	160.279	77.096
$Status = -1$	1 (0,43%)	147.243	147.243	147.243	147.243
Depois da segmentação					
Todos	234 (100%)	21	4.238.335	163.619	65.865
$Status = 1$	207 (88,46%)	21	4.238.335	164.782	53.257
$Status = 0$	26 (11,11%)	14.773	532.964	154.983	100.582

Entre os 207 *breakpoints* que passaram na verificação estatística, 83 (40,10%) apresentaram aumento em seus comprimentos. A média de comprimento que era de 70.602 bp antes da segmentação passou para 77.229 bp depois do procedimento de refinamento, mostrando que no geral o crescimento das sequências foi pequeno. Por outro lado, 124 pontos de quebra (59,90%) apresentaram redução de comprimento atingindo um refinamento médio de 57,30%. A Tabela 5.8 lista os valores mínimo, máximo, média e mediana dos comprimentos dos *breakpoints* destes dois conjuntos, além de exibir a porcentagem de refinamento do conjunto que apresentou redução de comprimento.

A Figura 5.15 mostra o histograma de distribuição das diferenças de tamanhos observadas nos *breakpoints* antes de depois da segmentação. O gráfico considera 202

Tabela 5.8: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos *breakpoints*, identificados entre o homem e o camundongo (`Compara`), que passaram na verificação estatística. Dos 207 *breakpoints*, 83 tiveram os seus comprimentos aumentados após a segmentação ($Depois \geq Antes$) e 124 tiveram seus comprimentos reduzidos ($Depois < Antes$). A tabela também exibe o refinamento (redução de comprimento) observado nestes 124 *breakpoints*.

Descrição	**N**	**Minímo**	**Máximo**	**Média**	**Mediana**
Antes da segmentação					
$Depois \geq Antes$	83 (40,10%)	1	543.101	70.602	24.923
$Depois < Antes$	124 (59,90%)	848	15.172.525	792.470	251.879
Depois da segmentação					
$Depois \geq Antes$	83 (40,10%)	21	543.249	77.229	33.298
$Depois < Antes$	124 (59,90%)	174	4.238.335	223.386	73.567
Redução de comprimento [%]					
$Depois < Antes$	124 (59,90%)	0,25	99,86	57,30	61,32

dos 205 pontos de quebra que passaram na verificação estatística. Por apresentarem diferenças maiores do que 3 Mbp, os demais *breakpoints* foram removidos com o propósito de se obter uma melhor visualização.

O histograma da Figura 5.15 mostra que os todos os *breakpoints* que apresentaram aumento em seu tamanho apresentam um pequeno acréscimo de comprimento. Entre os pontos de que foram refinados, a imensa maioria apresenta uma pequena redução de comprimento. Contudo, um total de 20 *breakpoints* tiveram redução maior do que 500 kbp (15 *breakpoints* que aparecem no gráfico somados aos 5 *breakpoints* com diferenças maiores do que 3 Mbp).

`MAUVE`

O programa `MAUVE` foi desenvolvido para a análise de genomas bacteriais [49]. Ele executa diversas etapas e uma delas realiza a construção de uma lista de LCBs (*Locally Collinear Blocks*) entre as espécies comparadas. Cada bloco colinear identificado é uma região homóloga entre todas as espécies que estão sendo comparadas (o programa pode lidar com mais de dois genoma simultaneamente).

Apesar de não ser criado para a análise de genomas de vertebrados, os autores



Figura 5.15: Histograma de distribuição das diferenças de tamanhos observadas nos *breakpoints*, produzidos por `Cassis` com os blocos de sintenia do homem e do camundongo obtidos da base de dados `Compara`, antes e depois da segmentação. O gráfico considera 202 *breakpoints* que passaram pela validação estatística (5 *breakpoints* com diferenças maiores do que 3 Mbp foram removidos para facilitar a visualização).

realizaram o alinhamento dos genomas do homem, do rato e do camundongo para exibir a escalabilidade do método. Para isso, eles criaram dois arquivos FASTA, um para cada espécie, contendo uma sequência formada pela concatenação das sequências de todos os cromossomos. Feito isso, eles executaram o procedimento de identificação de LCBs utilizando uma semente de tamanho 31 (comprimento mínimo de uma âncora) para a realização dos alinhamentos e um peso mínimo igual a 90 que os blocos deveriam apresentar para serem mantidos na lista de LCBs identificados.

Para avaliar outro conjunto de blocos de sintenia, obtidos com um método completamente distinto, decidimos reproduzir este experimento utilizando apenas os genomas do homem e do camundongo. Assim, criamos arquivos FASTA com as sequências concatenadas de todos os cromossomos, exceto o Y, para o homem e o camundongo e executamos o programa `MAUVE` com os mesmos parâmetros.

Através deste procedimento, `MAUVE` identificou 674 LCBs entre os dois genomas. Desta lista, descartamos 2 LCBs que continham um ponto de concatenação em seu interior: entre os cromossomos 10 e 11 (17.532.868 bp) e entre os cromossomos 14 e 15

(71.148.726 bp), ambos no genoma humano. A Tabela 5.9 mostra as características de tamanhos apresentadas pelos blocos que não foram descartados.

Tabela 5.9: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos 672 LCBs identificados pelo programa Mauve nos genomas do homem e do camundongo.

Descrição	Minímo	Máximo	Média	Mediana
Homo sapiens	98	58.978.356	3.964.687	717.051
Mus musculus	98	48.176.114	3.588.420	545.930

Comparando as Tabelas 5.5 e 5.9, podemos ver que Mauve obteve quase duas vezes mais blocos de sintenia do que os listados na base de dados Compara. Os blocos obtidos pelo programa Mauve apresentam uma média de tamanho menor. Um conjunto de 111 LCBs (16,52%) possuem tamanhos menores do que 5 kbp. Isso pode indicar que o programa Mauve está detectando pequenos rearranjos que acontecem no genoma mas que são mascarados por outros métodos. Por outro lado, como alguns blocos são muito pequenos, eles podem estar relacionados a falsos positivos.

Os 672 LCBs obtidos foram fornecidos como entrada para o Cassis que identificou 649 pontos de quebra. Um total de 582 *breakpoints* passaram na verificação estatística e 67 falharam. A Tabela 5.10 exibe as características de comprimento apresentado por estes pontos de quebra.

A maioria dos *breakpoints* que não passaram na verificação estatística são originalmente pequenos. Este motivo, associado a um possível grau menor de conservação destas sequências produziu alinhamentos que não foram capazes de gerar informação suficiente para a realização do refinamento.

Entre os 582 *breakpoints* que passaram na verificação estatística, 118 (20,27%) tiveram aumento do tamanho e 464 (79,73%) foram refinados. A Tabela 5.11 exibe as características destes dois conjuntos de pontos de quebra e ao refinamento obtido nos casos em que o comprimento foi reduzido.

O programa MAUVE identifica blocos de sintenia com uma maior resolução do que a apresentada pelos outros dois conjuntos de blocos de sintenia (blocos produzidos pelo Cassis e blocos produzidos pelo Compara). Neste conjunto tivemos uma porcentagem de *breakpoints* não refinados menor do que a observada no conjunto produzido com os blocos de sintenia do Compara: 20,27% contra 40,10%. Além disso, em média os pontos de quebra refinados apresentaram redução de 41,22% de seus tamanhos.

Tabela 5.10: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos *breakpoints* identificados entre o homem e o camundongo pelo programa `Cassis` ao processar a lista de LCBs do programa `MAUVE`. Dos 649 *breakpoints* que foram identificados e tinham sequências com tamanhos apropriados, 582 passaram na verificação estatística ($status = 1$) e 67 não passaram ($status = 0$).

Descrição	**N**	**Minímo**	**Máximo**	**Média**	**Mediana**
Antes da segmentação					
Todos	649 (100%)	2	29.940.399	311.898	59.543
$Status = 1$	582 (89,68%)	49	29.940.399	342.103	68.791
$Status = 0$	67 (10,32%)	2	699.084	49.516	12.765
Depois da segmentação					
Todos	649 (100%)	6	11.577.359	177.551	51.471
$Status = 1$	582 (89,68%)	6	11.577.359	193.671	59.525
$Status = 0$	67 (10,32%)	78	697.008	37.527	14.388

Tabela 5.11: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos *breakpoints*, identificados entre o homem e o camundongo (`MAUVE`), que passaram na verificação estatística. Dos 582 *breakpoints*, 118 tiveram os seus comprimentos aumentados após a segmentação ($Depois \geq Antes$) e 464 tiveram seus comprimentos reduzidos ($Depois < Antes$). A tabela também exibe o refinamento (redução de comprimento) observado nestes 464 *breakpoints*.

Descrição	**N**	**Minímo**	**Máximo**	**Média**	**Mediana**
Antes da segmentação					
$Depois \geq Antes$	118 (20,27%)	49	1.280.914	49.899	11.688
$Depois < Antes$	464 (79,73%)	2.143	29.940.399	416.414	98.389
Depois da segmentação					
$Depois \geq Antes$	118 (20,27%)	1.323	1.298.477	100.421	78.253
$Depois < Antes$	464 (79,73%)	6	11.577.359	217.385	48.285
Redução de comprimento [%]					
$Depois < Antes$	464 (79,73%)	0,01	99,98	41,22	35,02

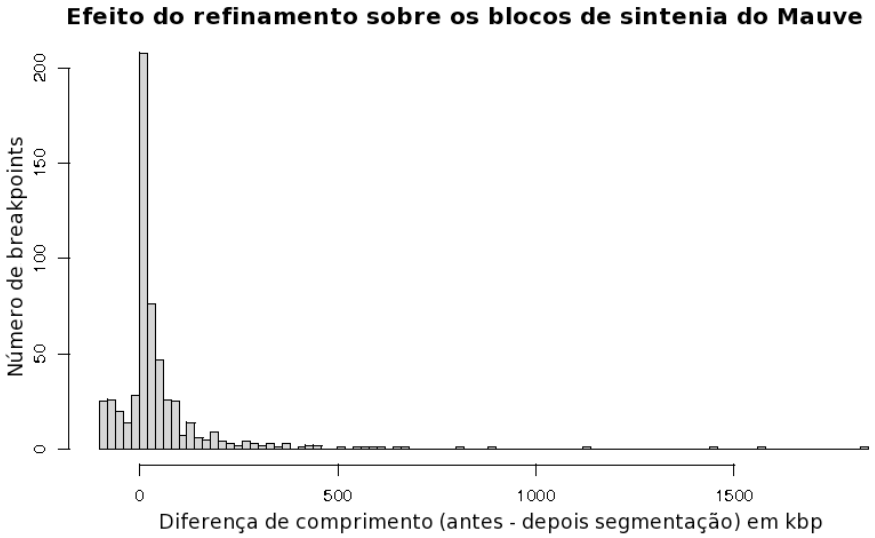


Figura 5.16: Histograma de distribuição das diferenças de tamanhos observadas nos *breakpoints*, produzidos por `Cassis` ao processar a lista de LCBs do programa `MAUVE`, antes e depois da segmentação. O gráfico considera 579 *breakpoints* que passaram pela validação estatística (3 *breakpoints* com diferenças maiores do que 2 Mbp foram removidos para facilitar a visualização).

A Figura 5.16 mostra o histograma de distribuição das diferenças de tamanhos observadas nos *breakpoints* antes de depois da segmentação. O gráfico considera 579 dos 582 pontos de quebra que passaram na verificação estatística. Por apresentarem diferenças maiores do que 2 Mbp, os demais *breakpoints* foram removidos com o propósito de se obter uma melhor visualização.

O gráfico da Figura 5.16 mostra que o número de *breakpoints* que tiveram acréscimo no tamanho é muito maior do que os observados nos histogramas dos outros dois métodos (Figuras 5.14 e 5.15). Como mencionado anteriormente, `MAUVE` produz um número muito maior de blocos de sintenia e muitos deles definem pequenas regiões de *breakpoints* que são mais difíceis de serem refinadas. Entre os pontos de que foram refinados, a imensa maioria apresenta uma pequena redução de comprimento. Contudo, um total de 16 *breakpoints* tiveram redução maior do que 500 kbp (13 *breakpoints* que aparecem no gráfico somados aos 3 *breakpoints* com diferenças maiores do que 2 Mbp).

Comparação entre `Cassis`, `Compara` e `MAUVE`

Para facilitar a comparação, daremos os seguintes nomes para os diferentes conjuntos de *breakpoints*:

- `GENES` - Conjunto de *breakpoints* obtidos com blocos de sintenia criados pelo programa `Cassis` a partir da lista de pares de genes ortólogos entre o homem e o camundongo obtida do `Ensembl`;
- `BLOCKS` - Conjunto de *breakpoints* obtidos pelo programa `Cassis` com base nos blocos de sintenia existentes entre o homem e o camundongo obtidos da base de dados `Compara` do `Ensembl`;
- `LCBS` - Conjunto de *breakpoints* obtidos pelo programa `Cassis` com base nos LCBs produzidos pelo programa `MAUVE` a partir das sequências de cromossomos do homem e do camundongo.

A Tabela 5.12 mostra os valores mínimo, máximo, média e mediana dos comprimentos das sequências dos *breakpoints* de cada conjunto, antes e depois da segmentação.

Tabela 5.12: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos *breakpoints* identificados entre o homem e o camundongo pelo programa `Cassis` com os conjuntos de blocos de sintenia `GENES`, `BLOCKS` e `LCBS`, antes e depois da segmentação.

Descrição	**N**	**Minímo**	**Máximo**	**Média**	**Mediana**
Antes da segmentação					
GENES	369	1	32.752.838	711.401	244.568
BLOCKS	234	1	15.172.525	463.422	114.679
LCBS	649	2	29.940.399	311.898	59.543
Depois da segmentação					
GENES	369	21	5.133.352	201.251	52.986
BLOCKS	234	21	4.238.335	163.619	65.865
LCBS	649	6	11.577.359	177.551	51.471

Se observarmos os tamanhos dos pontos de quebra antes da segmentação, podemos ver que o conjunto `GENES` possui uma média de tamanho substancialmente maior

do que a observada nos outros dois conjuntos. Como o programa `Cassis` utiliza a lista de genes ortólogos para definir os blocos de sintenia, suas extremidades ficam ancoradas nas bordas dos genes ortólogos. No caso dos outros dois conjuntos, alinhamentos são utilizados para a construção dos blocos baseados em similaridade de sequências e, por isso, eles podem avançar além dos genes ortólogos. Contudo, após a realização da segmentação a média e, especificamente, a mediana dos *breakpoints* do conjunto `GENES` ficam muito mais próximas dos valores exibidos pelos outros dois conjuntos.

Os conjuntos `BLOCKS` e `LCBS` apresentaram *breakpoints* menores do que os do conjunto `GENES`. Uma possível consequência disso é que o número de pontos de quebra que não foram refinados (tiveram acréscimo no tamanho) nos conjuntos `BLOCKS` e `LCBS` são maiores do que os do conjunto `GENES`. A Tabela 5.13 exibe as características das sequências que não foram refinadas em cada um dos conjuntos.

Tabela 5.13: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos *breakpoints*, identificados entre o homem e o camundongo pelo programa `Cassis` com os conjuntos de blocos de sintenia `GENES`, `BLOCKS` e `LCBS`, que não foram refinados.

Descrição	**N**	**Minímo**	**Máximo**	**Média**	**Mediana**
Antes da segmentação					
`GENES`	25 (6,85% de 365)	1	5.115.573	282.827	34.067
`BLOCKS`	83 (40,10% de 207)	1	543.101	70.602	24.923
`LCBS`	118 (20,27% de 582)	49	1.280.914	49.899	11.688
Depois da segmentação					
`GENES`	25 (6,85% de 365)	1.539	5.133.352	311.471	51.645
`BLOCKS`	83 (40,10% de 207)	21	543.249	77.229	33.298
`LCBS`	118 (20,27% de 582)	1.323	1.298.477	100.421	78.253

Dos 83 *breakpoints* que não foram refinados no conjunto `BLOCKS`, apenas 28 (33,73%) tiveram acréscimo maior do que 1 kbp após a fase de segmentação do programa `Cassis`. Assim, em aproximadamente 1/3 dos casos `Cassis` produziu resultados que pioram substancialmente a resolução dos *breakpoints* produzidos pelo metodologia empregada na base de dados `Compara`.

Em relação ao conjunto `MAUVE`, dos 118 *breakpoints* que apresentaram acréscimo no tamanho, 113 (95,76%) tiveram extensão maior do que 1 kbp. Contundo, obser-

vamos que quase metade deles (52) são delimitados, em pelo menos um dos lados, por um LCB com tamanho menor do que 10 kbp. Isso sugere que estes *breakpoints* podem não ser confiáveis.

Como os blocos de sintenia da base de dados `Compara` e os LCBs produzidos pelo `MAUVE` utilizam informação de similaridade de sequência, eles possuem a vantagem de não terem suas extremidades ancoradas em genes como no caso dos blocos produzidos pelo `Cassis`. No entanto, apesar desta característica apresentada pelos dois conjuntos, `Cassis` foi capaz de refinar a maioria dos seus *breakpoints*. A Tabela 5.14 mostra a características dos *breakpoints* que puderam ser refinados em cada um dos três conjuntos.

Tabela 5.14: Mínimo, máximo, média e mediana dos comprimentos (em bp) dos *breakpoints*, identificados entre o homem e o camundongo pelo programa `Cassis` com os conjuntos de blocos de sintenia `GENES`, `BLOCKS` e `LCBS`, que foram refinados. A Tabela também exibe a porcentagem de refinamento dos *breakpoints* dentro destes conjuntos.

Descrição	**N**	**Minímo**	**Máximo**	**Média**	**Mediana**
Antes da segmentação					
GENES	340 (93,15% de 365)	303	32.752.838	811.739	256.577
BLOCKS	124 (59,90% de 207)	848	15.172.525	792.470	251.879
LCBS	464 (79,73% de 582)	2.143	29.940.399	416.414	98.339
Depois da segmentação					
GENES	340 (93,15% de 365)	21	4.860.908	195.085	54.883
BLOCKS	124 (59,90% de 207)	174	4.238.335	223.386	73.567
LCBS	464 (79,73% de 582)	6	11.577.359	217.385	48.285
Redução de comprimento [%]					
GENES	340 (93,15% de 365)	0.31	99.99	62.26	71.20
BLOCKS	124 (59,90% de 207)	0.25	99.86	57.30	61.32
LCBS	464 (79,73% de 582)	0.01	99.98	41.22	35.02

Por definição os blocos de sintenia produzidos pelo programa `MAUVE` não apresentam sobreposições entre eles. Relacionada ao procedimento de identificação de *breakpoints*, realizado pelo `Cassis` a partir de blocos de sintenia, a inexistência de sobreposições entre os blocos é uma característica desejável pois ela elimina a chance

de ocorrência de descartes de *breakpoints* devido a criação de sequências (S_r, S_{oA} e S_{oB}) com tamanhos inferiores ao limite estipulado.

Esta característica não é observada nos dados provenientes da base de dados Compara. De fato, a metodologia utilizada é assimétrica e garante apenas a obtenção de blocos que não se sobrepõem no genoma de referência (genoma humano neste caso). Dos 345 blocos de sintenia do conjunto BLOCKS, 88 apresentavam sobreposição no genoma do camundongo. Cassis foi capaz de identificar 292 *breakpoints*, mas 58 deles não puderam ser processados pela etapa de segmentação porque os tamanhos das sequências estendidas S_{oA} ou S_{oB} eram menores do que o limite estabelecido (50 kbp), evidenciando ocorrências de grandes sobreposições entre blocos.

5.2.4 Conclusão

A implementação do método proposto por Lemaitre gerou um pacote denominado Cassis disponível para uso público.

Este pacote permitiu a realização de testes com um conjunto de pares de genes ortólogos obtidos com versões mais recentes dos genomas humano e do camundongo. Lemaitre utilizou dados de ortologia provenientes das montagens NCBI35 do homem e NCBI m35 do camundongo disponíveis na versão 24 do Ensembl. Esta lista possui um total de 13.577 pares de genes ortólogos 1-1 que, após eliminação de sobreposições e identificação de blocos de sintenia, geraram 366 pontos de quebra.

Nós utilizamos dados das montagens GRCh37 do genoma humano e NCBI m37 do camundongo disponíveis na versão 56 do Ensembl. A partir do processamento de uma lista de 15.047 pares de genes ortólogos, conseguimos um número um pouco maior de *breakpoints*: 369 pontos de quebra.

De posse deste novo conjunto de pontos de quebra, nós realizamos estudos de suas características para definição de uma nova metodologia de detecção de *breakpoints*. Este estudo será abordado no próximo capítulo.

A implementação do pacote também permitiu a realização de testes comparativos entre o método de identificação de blocos de sintenia do Cassis e outros métodos de definição de blocos de sintenia: blocos obtidos segundo a metodologia da base de dados Compara e blocos gerados pelo programa Mauve.

Estes testes mostraram, que em geral, devido ao fato de os limites de nossos *breakpoints* serem ancorados nos primeiros/últimos genes ortólogos dos blocos de sintenia que os definem, o comprimento dos pontos de quebra obtidos são maiores do que

aqueles observados nos outros dois conjuntos criados com metodologias baseadas em similaridade de sequências.

Por outro lado, o algoritmo de segmentação mostrou grande poder de refinamento. Tanto os *breakpoints* do conjunto `GENES` como os *breakpoints* dos conjuntos `BLOCKS` e `LCBs` apresentaram um grande percentual de refinamento.

O programa `MAUVE` não foi concebido para identificação de blocos de sintenia em genomas de vertebrados. Ele produziu uma quantidade muito maior de blocos que geraram mais *breakpoints* que os outros dois conjuntos. Muitos do blocos de sintenia obtidos são pequenos e podem estar ligados a pequenos rearranjos que são mascarados no interior de blocos maiores, identificados por outros métodos. Apesar de ser uma hipótese possível, a existência de um grande número de *breakpoints* flanqueados por blocos de sintenia menores do que 10 kbp indicam a possibilidade de eles serem falsos positivos.

O número de *breakpoints* identificados com a lista de genes ortólogos é maior do que o número de pontos de quebra identificados com os blocos de sintenia do `Compara`. Esta diferença se deve principalmente à metodologia de construção de blocos de sintenia. O método implementado por `Cassis` é mais rigoroso no aspecto que não permite a sobreposição de blocos de sintenia e, ao mesmo tempo, possui mais resolução pois utiliza informações de genes ortólogos para definir os seus blocos. A metodologia do `Compara` permite sobreposição de blocos no genoma que está sendo comparado ao genoma de referência e é construído baseado em alinhamento de sequências. Devido a estas características, os blocos produzidos pelo `Compara` tendem a ser maiores.

Trabalhos futuros

Uma desvantagem do `Cassis` em relação ao `MAUVE` e ao `Compara` é a incapacidade de lidar com mais de dois genomas. Assim, um trabalho futuro seria a adaptação da metodologia para realização de comparações com um número maior de genomas.

Esta adaptação contudo não é simples e exigirá uma série de estudos para definir a melhor maneira de se lidar com múltiplos genomas. Por exemplo, `Cassis` foi concebido para trabalhar com pares de genes ortólogos entre duas espécies. A análise de n espécies exigirá grupos de n genes ortólogos, um em cada espécie. Conforme n aumenta, obter esta lista de dados pode ficar mais complicada devido às dificuldades de se estabelecer as relações de ortologia.

Além disso, o nível de flexibilidade terá um papel importante. Podemos exigir que apenas genes ortólogos que aparecem nas n espécies serão utilizados para a construção dos blocos. Apesar dessa exigência gerar blocos mais confiáveis, informações interessantes serão perdidas como, por exemplo, um par de genes ortólogos entre uma espécie A e uma espécie B, mas que não possui ortólogo ou possui um ortólogo duplicado na espécie C.

A etapa de segmentação também possui questões de extrema relevância. Por exemplo, a metodologia de segmentação foi concebida para tratar de uma sequência de valores -1, 0 e $+1$ provenientes de *hits* de alinhamentos de S_r com S_{oA} e S_{oB}. Adaptar este método ou desenvolver novo método de segmentação será necessário para o tratamento de dados provenientes de múltiplos alinhamentos.

Outro exemplo de problema a ser analisado são os alinhamentos das sequências. Optar por alinhamentos de sequências duas a duas ou por um alinhamento múltiplo pode ter impacto significativo no processo.

Uma abordagem simples que pode ser proposta seria a utilização do `Cassis`, da maneira como foi concebido, para a realização da comparação de espécies duas a duas. Posteriormente, os dados produzidos seriam analisados e as relações de evolução poderiam ser inferidas.

Capítulo 6

Análise de regiões intergênicas

O pacote `Cassis` apresentado no Capítulo 5 (Seção 5.2) foi projetado para a identificação de pontos de quebra baseado nas informações de genes ortólogos. Se por um lado, temos uma produção de blocos de sintenia mais confiáveis, por outro lado, podemos ter perdas de informações no caso de existência de genes que não possuem informação ou que contém erros de atribuição de ortologia. Os erros, inclusive, podem provocar a detecção de *breakpoints* falsos ou errôneos. Além disso, no caso de estudos de genomas de espécies que não possuem informação de ortologia, o método utilizado por `Cassis` não é aplicável.

Desenvolver um método que fosse independente da geração prévia de listas de marcadores ortólogos é a motivação para o estudo detalhado neste capítulo. A idéia é utilizar informações extraídas das próprias sequências dos cromossomos dos genomas que estão sendo comparados para realizar a identificação dos *breakpoints*.

Em seu estudo de doutorado, Lemaitre analisou as características das sequências contidas no interior de regiões de *breakpoints* [100]. Os resultado obtidos mostraram que as regiões de *breakpoints* tendem a apresentar, em alinhamentos com suas regiões ortólogas, níveis de similaridade menor dos que os obtidos por outras regiões fora de *breakpoints*. Isso indica que estas sequências possuem um grau de conservação menor. Além disso, através da comparação entre as sequências dentro dos pontos de quebra com as sequências de regiões adjacentes, Lemaitre detectou uma série de características locais que podem estar ligadas aos eventos de rearranjo: regiões ricas em duplicação assim como presença de elementos transponíveis.

Baseado nestas características, conduzimos um estudo de uma metodologia que procura identificar os pontos de quebra com base nos diferentes níveis de similaridade

apresentados pelas sequências dos cromossomos de dois genomas distintos.

Como a evolução tende a evitar pontos de quebra no interior de genes, optamos por eliminá-los de nossa análise. Assim, o nosso estudo considera apenas o grau de similaridade apresentado por regiões intergênicas. Apesar destas regiões possuírem um grau menor de conservação e de apresentarem uma grande quantidade de artefatos como duplicações e elementos transponíveis, acreditamos que a análise delas em conjunto pode nos fornecer pistas importantes para a delimitação de regiões de pontos de quebra.

Assim, neste capítulo apresentaremos os detalhes de nossa pesquisa em busca de uma nova metodologia.

6.1 Definição do conjunto de dados

Para nossos estudos utilizamos os mesmos dados que foram empregados na avaliação do pacote `Cassis`.

Através do *site* `FTP` da plataforma `Ensembl` (versão 56 – Setembro de 2010), obtemos as sequências dos cromossomos do homem e do camundongo. As sequências humanas têm origem na montagem `GRCh37` do genoma humano (*Genome Reference Consortium* – Fevereiro de 2009). As sequências do genoma do camundongo são provenientes da montagem `NCBI m37` (*Mouse Genome Sequencing Consortium* – Abril de 2007).

As sequências FASTA de todos os cromossomos, com exceção do cromossomo Y, foram obtidas para os dois genomas. Estas sequências foram recuperadas em suas versões processadas pelo programa `RepeatMasker` [149], que realiza o mascaramento de repetições simples, de regiões de baixa complexidade e de elementos transponíveis. A Tabela 6.1 mostra os tamanhos dos cromossomos obtidos para as duas espécies.

Utilizamos a ferramenta `Biomart` [148], também da plataforma `Ensembl`, para obter a lista de todos os genes dos genomas humano e do camundongo. Esta ferramenta possui um processo de anotação de genes automatizado que se baseia no uso de mRNA e proteínas contidos em banco de dados públicos. Os transcritos identificados são revisados com utilização de um conjunto de genes manualmente curado do projeto `Vega` e com transcritos de proteínas codificantes provenientes do projeto `CCDS` [47, 133]. O projeto `Vega` (*Vertebrate Genome Annotation*) é um repositório para anotações manuais de alta qualidade realizadas sobre genomas de vertebrados

Tabela 6.1: Tamanho em pares de bases, número e porcentagem de bases mascaradas (**Mask** e **%Mask**) apresentados pelos cromossomos dos genomas do homem e do camundongo

	Homem – *Homo sapiens*			**Camundongo – *Mus musculus***		
Crom.	**Tamanho**	**Mask**	**%Mask**	**Tamanho**	**Mask**	**%Mask**
1	249.250.621	140.266.537	56,28	197.195.432	89.913.587	45,60
2	243.199.373	122.531.588	50,38	181.748.087	76.350.005	42,01
3	198.022.430	102.588.875	51,81	159.599.783	73.187.707	45,86
4	191.154.276	100.804.596	52,73	155.630.120	71.255.361	45,79
5	180.915.260	93.838.851	51,87	152.537.259	67.496.260	44,25
6	171.115.067	87.667.005	51,23	149.517.037	66.097.405	44,21
7	159.138.663	83.312.396	52,35	152.524.553	75.658.746	49,60
8	146.364.022	76.821.843	52,49	131.738.871	58.089.743	44,09
9	141.213.431	82.696.898	58,56	124.076.172	53.653.362	43,24
10	135.534.747	69.417.262	51,22	129.993.255	57.291.438	44,07
11	135.006.516	71.925.405	53,28	121.843.856	49.989.899	41,03
12	133.851.895	72.050.237	53,83	121.257.530	53.531.476	44,15
13	115.169.878	66.339.364	57,60	120.284.312	53.549.589	44,52
14	107.349.540	64.095.543	59,71	125.194.864	55.339.159	44,20
15	102.531.392	61.683.454	60,16	103.494.974	44.765.560	43,25
16	90.354.753	52.281.441	57,86	98.319.150	43.079.315	43,82
17	81.195.210	42.540.405	52,39	95.272.651	43.024.683	45,16
18	78.077.248	39.287.779	50,32	90.772.031	39.939.716	44,00
19	59.128.983	36.575.289	61,86	61.342.430	26.598.627	43,36
20	63.025.520	34.227.361	54,31	-	-	-
21	48.129.895	30.330.268	63,02	-	-	-
22	51.304.566	34.169.484	66,60	-	-	-
X	155.270.560	97.220.886	62,61	166.650.296	98.134.602	58,89

sequenciados completamente [169]. O projeto `CCDS` (*Consensus Coding Sequence*) tem o objetivo de produzir um conjunto padrão de regiões codificantes de proteínas que são frequentemente anotadas e que possuem alta qualidade [134].

Para o genoma humano, foram recuperados 46.946 genes e pseudo-genes que apresentaram um tamanho médio de 31.005,24 bp. Para o genoma do camundongo, obtemos 31.458 genes e pseudo-genes com tamanho médio de 30.901,09 bp. As Tabelas 6.2 e 6.3 mostram o número e o tamanho médio dos genes obtidos do Ensembl, respectivamente, para o homem e para o camundongo (coluna "Genes `Ensembl`").

Esta lista de genes foi processada da seguinte maneira: todos genes consecutivos que apresentaram sobreposições ou que estavam a menos de 100 bp de distância um do outro foram agrupados para formar um único gene. Este procedimento foi feito para evitar a criação de regiões intergênicas menores do que 100 bp. A lista de 46.946 genes do genoma humano foi reduzida para 29.542 genes com tamanho médio de 46.106,27 bp. Já a lista de genes do camundongo passou de 31.458 para 26.276 genes com tamanho médio de 46.106,27 bp. As Tabelas 6.2 e 6.3 mostram o número e o tamanho médio dos genes processados, respectivamente, para o homem e para o camundongo (coluna "Genes Processados").

Toda região situada entre dois genes processados foi marcada como uma região intergênica. Cada região foi identificada com o par (c, i) onde c é o cromossomo onde ela se localiza e i é o índice de posicionamento dela dentro do cromossomo. Os índices foram atribuídos por ordem de posicionamento na sequência do cromossomo: a primeira região recebeu número 1, a segunda recebeu número 2, e assim por diante. No genoma humano foram identificadas 29.542 regiões intergênicas que apresentaram tamanho médio de 56.708,85 bp. No genoma do camundongo foram listadas 26.296 regiões intergênicas com tamanho médio de 64.062,79 bp. As Tabelas 6.2 e 6.3 mostram o número e o tamanho médio das regiões intergênicas, respectivamente, para o homem e para o camundongo (coluna "Regiões Intergênicas").

Estes dois conjuntos de regiões intergênicas foram utilizados para o nosso estudo. Note que, para definir estas regiões, utilizamos informações de genes obtidas com base em uma refinada metodologia desenvolvida para a plataforma `Ensembl`. Contudo, no caso da inexistência deste tipo de informação para o genoma estudado, métodos para predição de genes com base na análise das sequências poderiam ser utilizados.

Além disso, o conjunto de genes utilizados é muito diferente do conjunto de genes ortólogos utilizados pelo `Cassis`. Além de ser maior, as únicas informações utilizadas são as coordenadas e orientações dos genes nos cromossomos. Nenhuma informação

Tabela 6.2: Número e tamanho médio dos genes e regiões intergênicas do homem. Todos os genes obtidos do `Ensembl` foram processados de modo que genes consecutivos que apresentassem sobreposição ou estivessem a menos de 100 bp de distância foram agrupados para formar um único gene. As regiões intergênicas foram definidas a partir da lista de genes processados.

	Genes				**Regiões Intergênicas**	
	`Ensembl`		**Processados**			
Crom.	**N**	**Tam. médio**	**N**	**Tam. médio**	**N**	**Tam. médio**
1	5.230	26.450,27	3.081	41.712,68	3.082	39.173,86
2	4.004	36.336,21	2.195	60.560.59	2.196	50.213,51
3	2.926	43.654,03	1.547	75.104,33	1.548	52.865,66
4	1.548	44.285,01	1.129	58.828,38	1.130	110.386,76
5	1.672	39.771,83	1.203	53.722,69	1.204	96.583,77
6	2.873	29.299,25	1.833	43.013,31	1.834	50.311,70
7	2.945	34.428,50	1.593	57.635,74	1.594	42.236,47
8	1.429	38.174,38	1.032	51.218,32	1.033	90.519,57
9	2.386	25.584,85	1.506	38.270,95	1.507	55.459,44
10	2.284	33.644,88	1.335	53.638,74	1.336	47.849,57
11	2.141	27.742,17	1.530	37.088,12	1.531	51.118,02
12	1.752	32.882,67	1.316	42.849,27	1.317	58.817,20
13	1.222	32.951,70	757	49.638,58	758	102.366,06
14	1.510	24.706,93	989	37.016,68	990	71.454,59
15	1.315	31.308,70	841	47.203,68	842	74.623,63
16	1.425	27.542,49	992	37.323,58	993	53.705,70
17	2.052	22.929,57	1.342	33.159,95	1.343	27.322,83
18	569	47.730,65	394	68.338,44	395	129.498,49
19	2.005	17.264,88	1.474	21.393,33	1.475	18.708,62
20	1.307	26.671,73	796	40.605,75	797	38.523,64
21	731	29.502,17	412	46.152,07	413	70.496,96
22	1.241	23.362,48	595	40.738,84	596	45.411,00
X	2.379	25.938,65	1.627	35.029,91	1.628	60.366,64
Todos	46.946	31.005,24	29.519	46.106,27	29.542	56.708,85

Tabela 6.3: Número e tamanho médio dos genes e regiões intergênicas do camundongo. Todos os genes obtidos do `Ensembl` foram processados de modo que genes consecutivos que apresentassem sobreposição ou estivessem a menos de 100 bp de distância foram agrupados para formar um único gene. As regiões intergênicas foram definidas a partir da lista de genes processados.

	Genes				**Regiões Intergênicas**	
	`Ensembl`		**Processados**			
Crom.	**N**	**Tam. médio**	**N**	**Tam. médio**	**N**	**Tam. médio**
1	1.647	43.498,49	1.374	51.338,83	1.375	92.113,37
2	3.013	25.812,31	2.373	32.279,56	2.374	44.291,78
3	1.377	33.409,29	1.156	39.312,48	1.157	98.664,27
4	2.383	26.038,73	1.903	31.767,34	1.904	49.987,86
5	1.654	37.620,92	1.322	46.335,64	1.323	68.995,88
6	1.645	35.714,70	1.394	41.553,62	1.395	65.656,84
7	2.585	21.027,49	2.260	23.638,51	2.261	43.830,83
8	1.424	32.218,22	1.228	36.751,28	1.229	70.470,54
9	1.511	33.972,47	1.302	38.696,31	1.303	56.556,85
10	1.232	38.303,49	1.068	43.481,15	1.069	78.162,19
11	2.483	22.513,93	1.984	27.667,38	1.985	33.728,85
12	1.112	38.323,43	891	46.644,19	892	89.347,03
13	1.163	33.895,61	1.016	38.098,51	1.017	80.205,08
14	1.224	37.632,21	1.043	42.967,83	1.044	76.991,78
15	980	37.259,16	845	42.793,76	846	79.591,31
16	922	36.491.98	761	43.264,24	762	85.820,30
17	1.271	27.915.47	1.077	32.493,40	1.078	55.915,82
18	652	48.281.60	561	54.379,27	562	107.233,56
19	871	31.543.23	757	35.687,71	758	45.286,06
X	2.309	20.047,83	1.961	23.059,02	1.962	61.891,72
Todos	31.458	30.901,09	26.276	36.321,73	26.296	64.062,79

extra sobre relação de ortologia entre as duas espécies está disponível.

6.2 Alinhamento de sequências intergênicas

Para avaliar a similaridade das regiões intergênicas, optamos pela realização de alinhamentos locais. Baseados no estudo de Sun e Buhler, que mostraram que o `BLASTZ` [144] é o alinhador mais adequado para a realização de alinhamentos de sequências não codificantes [150] e nos resultados obtidos por Lemaitre que confirmaram as observações deste estudo [100], optamos por utilizar este programa para a realização de nossos alinhamentos.

Contudo, alinhar 29.542 regiões intergênicas humanas contra 26.296 regiões intergênicas do camundongo (776.836.432 pares de regiões) demandaria muito tempo. Assim, decidimos realizar uma primeira filtragem utilizando o programa `BLAST` [3], que possui uma performance melhor do que o `BLASTZ`.

O programa `BLASTN` foi utilizado para realização de alinhamentos de todos as regiões intergênicas humanas contra uma base construída com as regiões intergênicas do camundongo que identificamos. Da mesma maneira, as regiões intergênicas do camundongo foram alinhadas contra uma base construída com as regiões intergênicas humanas. Os seguintes parâmetros foram utilizados:

- `-F F`: Filtro de regiões de baixa complexidade ou segmentos contendo repetições curtas foi desligado.
- `-G -1`: Custo para abrir um buraco (*gap*).
- `-E -1`: Custo para estender um buraco.
- `-q -3`: Penalização para nucleotídeos que não se correspondem (*mismatch*).
- `-r 2`: Pontuação dada para nucleotídeos que se correspondem (*match*).
- `-W 11`: Tamanho da semente utilizada no `BLASTN`

Estes parâmetros são menos restritos do que os valores padrões do programa e foram adotados para maximizar a identificação de pares de regiões que apresentem algum nível de similaridade. Qualquer par de regiões que apresentasse um *hit*, independente do valor de *e-value*, foi selecionado para realização de alinhamento com

o `BLASTZ`. Este procedimento resultou na identificação de 5.022.606 pares de regiões intergênicas com pelo menos um *hit* (0,65% dos 776.836.432 pares possíveis).

Os 5.022.606 pares de regiões intergênicas foram então alinhados com o programa `BLASTZ` com o seguinte conjunto de parâmetros:

- `K=3000`: Pontuação mínima que um alinhamento deve apresentar para ser mantido na primeira fase do algoritmo.
- `H=2000`: Pontuação mínima que os alinhamentos devem possuir para serem interpolados.
- `L=3000`: Pontuação mínima que um alinhamento deve apresentar para ser mantido na segunda fase do algoritmo.
- `O=400`: Custo para abertura de um buraco.
- `E=30`: Custo para estender um buraco.
- `B=2`: Ativa a análise da sequência de DNA considerando as duas direções.
- `Matriz de substituição padrão do programa`

	A	C	G	T
A	91	−114	−31	−123
C	−114	100	−125	−31
G	−31	−125	100	−114
T	−123	−31	−114	91

Estes parâmetros são similares aos usados pelo plataforma *UCSC Genome Browser* [67] para a realização de alinhamentos entre o homem e o camundongo.

Analisando os resultados dos alinhamentos `BLASTZ`, identificamos 135.967 pares de regiões intergênicas que apresentaram ao menos um *hit* (2,70%) e 4.886.639 pares que não tiveram *hits* (97,30%). O conjunto de *hits* foi usado em todas as nossas análises que utilizaram o genoma humano como genoma de referência.

Tanto os alinhamentos `BLASTN` como os alinhamentos `BLASTZ` foram divididos em 4 processos que executaram em uma máquina dotada de 8 processadores Intel Xeon X550 2,67 GHz e 24 GB de RAM. Para processar todos os pares de sequências, o

`BLASTN` necessitou alguns dias para concluir a tarefa. O `BLASTZ`, por sua vez, necessitou algumas semanas para concluir o alinhamento dos pares de regiões intergênicas selecionados pelo `BLASTN`.

Ao optar pela utilização apenas de sequências intergênicas, estamos utilizando a hipótese que estas regiões não apresentam bons *hits* com sequências gênicas e, assim, o descarte dos genes não apresenta problema para a nossa metodologia.

Para avaliar este aspecto, realizamos o alinhamento com `BLASTZ` de todas as regiões intergênicas do homem contra todos os genes do camundongo para avaliar o nível de similaridade que estas sequências apresentam. Para cada região intergênica do homem, as coberturas total (considerando a região completa) e não-mascarada (considerando apenas as bases não mascaradas) foram calculadas. A Tabela 6.4 exibe os resultados obtidos.

A Tabela 6.4 mostra que apenas 2,54% das regiões intergênicas do homem apresentaram algum *hit* com uma sequência gênica do camundongo. Além disso, apesar de algumas regiões apresentarem porcentagens de cobertura mais altas, em torno de 37,9% considerando as sequências completas ou 67,8% para as sequências sem as bases mascaradas, a média (entre as regiões que apresentaram algum *hit*) é muito baixa: 0,64% para as sequências completas e 1,23% para as sequências sem bases mascaradas. De fato, das 29.542 regiões intergênicas do genoma humano, apenas 13 (0,04%) apresentaram cobertura maior do que 5% quando consideramos a sequência completa. Se consideramos apenas as bases não mascaradas, 33 (0,11%) regiões apresentaram cobertura maior do que 5%. Esses resultados nos fornecem mais segurança quanto a decisão de descarte das sequências de genes.

6.3 Metodologia para detecção de *breakpoints*

De posse dos dados de alinhamentos entre as regiões intergênicas das duas espécies, iniciamos o desenvolvimento de uma metodologia de análise dos *hits*. Para facilitar a descrição, o genoma humano, que é nosso genoma de referência, será denominado G_r e o genoma do camundongo será denominado G_o.

Os *hits* foram agrupados de acordo com o cromossomo a que estão associados no genoma G_r. Assim, inicialmente cada *hit* possui o seguinte conjunto de informações:

- Par (C_r, I_r) referente à região intergênica identificada pelo índice I_r no cromossomo C_r do genoma G_r.

Tabela 6.4: Resultados dos alinhamentos de todas as sequências intergênicas humanos contra todos as sequências gênicas do camundongo. A tabela lista o número total de regiões intergênicas (**N**), o número de regiões intergênicas que apresentaram algum *hit* (***Hits***), a porcentagem das regiões que apresentaram algum *hit* (**%*Hits***) e as coberturas mínima, máxima e média observadas nas regiões que apresentaram algum *hit* considerando a região completa (**Todas bases**) ou apenas as bases que não foram mascaradas (**Bases não mascaradas**).

				Cobertura por *hits* [%]					
	Regiões Intergênicas			**Todas bases**			**Bases não mascaradas**		
Crom.	**N**	***Hits***	**%*Hits***	**Min.**	**Máx.**	**Média**	**Min.**	**Máx.**	**Média**
1	3.082	39	1,25	0,012	5,842	0,617	0,017	7,061	1,252
2	2.196	56	2,50	0,008	37,906	1,017	0,015	67,784	1,979
3	1.548	35	2,21	0,015	30,707	1,242	0,029	30,707	1,570
4	1.130	51	4,53	0,004	0,362	0,087	0,007	0,697	0,168
5	1.204	58	4,82	0,008	24,326	0,577	0,012	24,326	0,748
6	1.834	48	2,62	0,005	26,836	1,048	0,010	63,886	2,370
7	1.594	30	1,88	0,013	12,307	0,874	0,025	25,023	2,054
8	1.033	48	4,65	0,004	2,890	0,311	0,007	4,441	0,549
9	1.507	42	2,79	0,009	3,033	0,495	0,017	7,181	0,982
10	1.336	39	2,92	0,016	2,212	0,263	0,030	4,281	0,578
11	1.531	37	2,42	0,005	6,117	0,477	0,009	9,906	0,993
12	1.317	39	2,96	0,003	7,907	0,557	0,019	22,432	1,284
13	758	20	2,64	0,003	0,698	0,146	0,005	2,476	0,334
14	990	22	2,22	0,013	34,215	2,223	0,019	45,956	3,528
15	842	25	2,97	0,008	6,673	0,759	0,020	15,513	1,709
16	993	20	2,01	0,011	1,758	0,451	0,019	5,531	1,080
17	1.343	21	1,56	0,003	5,628	0,903	0,023	11,644	1,799
18	395	26	6,58	0,005	1,612	0,132	0,008	3,867	0,297
19	1.475	20	1,36	0,051	3,889	0,957	0,079	6,727	2,090
20	797	15	1,88	0,024	12,067	1,084	0,042	21,237	2,067
21	413	7	1,70	0,043	0,804	0,282	0,062	1,333	0,425
22	596	3	0,50	0,056	0,766	0,358	0,094	1,481	0,720
X	1.628	48	2,95	0,009	2,399	0,252	0,024	5,401	0,700
Total	29.542	749	2,54	0,003	37,906	0,637	0,005	67,784	1,234

- Par de coordenadas (B_r, E_r) referente às posições inicial e final do *hit* ao longo da sequência do genoma G_r.
- Par (C_o, I_o) referente à região intergênica identificada pelo índice I_o no cromossomo C_o do genoma G_o.
- Orientação $\sigma_o \in \{-1, +1\}$ do *hit* no cromossomo C_o.

Esta lista de *hits* possui sobreposições e com o objetivo de facilitar o tratamento destes dados decidimos processá-la de modo a distinguir dois tipos de *hits*: *hits* que identificam correspondências únicas e *hits* que identificam correspondências com múltiplas sequências. Para isso, as regiões que apresentaram sobreposições de *hits* foram identificadas e fragmentadas de modo a separar regiões que possuíam apenas cobertura de um único *hit* e regiões que eram coberta por mais de um *hit* distinto (diferentes cromossomos, índices ou mesmo orientações, em casos que um mesmo par (C_o, I_o)). Os *hits* que indicam múltiplas correspondências receberam $\sigma_o = 0$. A Figura 6.1 exibe um pequeno exemplo do modo como os *hits* foram reorganizados.

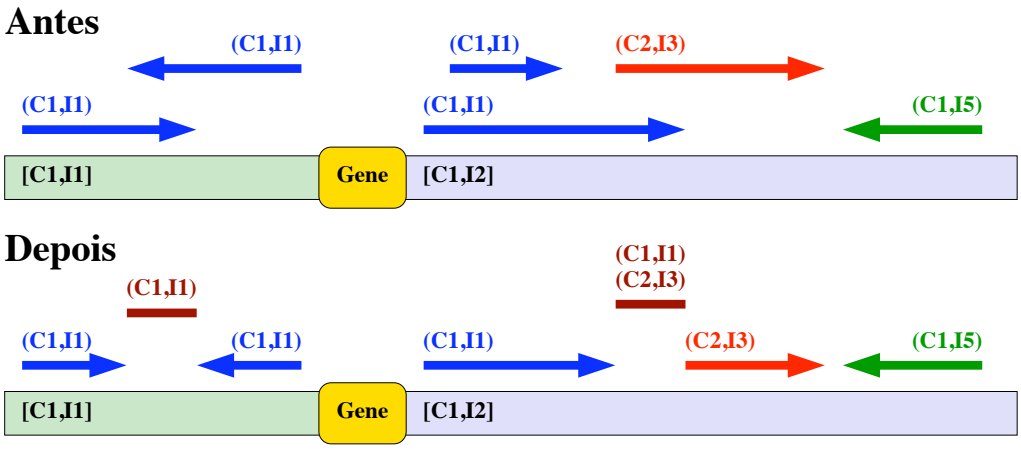


Figura 6.1: Qualquer *hit* que apresentasse sobreposição com outro foi fragmentado em regiões menores para distinção de locais que são cobertos por um único *hit* e locais que são cobertos por múltiplos *hits* (que diferem ou em relação ao par (C_o, I_o) ou em relação à orientação σ_o).

Após este processamento, um *hit* possui o seguinte conjunto de informações:

- Par (C_r, I_r) referente à região intergênica identificada pelo índice I_r no cromossomo C_r do genoma G_r.
- Par de coordenadas (B_r, E_r) referente às posições inicial e final do *hit* ao longo da sequência do genoma G_r.
- Lista de $n \geq 1$ pares (C_o, I_o) referentes às regiões intergênicas do genoma G_o que possuem correspondência com a região delimitada em (C_r, I_r).
- Orientação σ_o do *hit*. O valor de $\sigma_o \in \{-1, +1\}$ quando o *hit* está relacionado a uma única região em G_o ou $\sigma_o = 0$ quando existem múltiplas correspondências.

6.3.1 Fragmentação das regiões intergênicas

Uma análise inicial dos *hits* mostrou que se tratam de dados difíceis de serem interpretados. As taxas de coberturas das regiões intergênicas variam muito entre si e podem ter comportamentos completamente diferentes ao longo da sequência de um cromossomo.

Analisar as regiões intergênicas individualmente também se mostra complicado por que os seus tamanhos também apresentam grande variação, podendo ir de algumas centenas até alguns milhões de pares bases. Essa disparidade causa problemas na análise de cobertura das regiões pois uma pequena região intergênica é muito menos informativa do que uma grande região. Assim, compará-las entre si não parece ser uma abordagem apropriada.

Outro ponto a ser observado durante esta análise é que as sequências de genes não estão presentes. Este fato pode influenciar negativamente estratégias que realizam o cálculo de cobertura considerando segmentos de sequências de tamanhos fixos como, por exemplo, a técnica de janela deslizante. Como os genes possuem diferentes tamanhos e como suas posições não possuem qualquer *hit*, a presença deles dentro do segmento "mascara" a real cobertura de *hits* da região.

Desse modo, temos que conceber um método que seja capaz de processar estes dados considerando o peso relativo das regiões intergênicas (tamanho e porcentagem de bases não mascaradas), lembrando que as sequências gênicas não são consideradas.

Para equilibrar o peso das regiões intergênicas, todas elas foram divididas em fragmentos de 100 bp, tamanho equivalente ao comprimento mínimo observado em todo o conjunto. A divisão das regiões ocorreu sem nenhuma sobreposição entre

os fragmentos e, portanto, cada fragmento equivale a uma porção única de 100 bp. As identificações dos fragmentos consideraram os seus índices de posicionamento ao longo da sequência do cromossomo no genoma G_r (o primeiro fragmento foi identificado com o número 1, o segundo com o número 2, etc.) independente da posição física na sequência (distância entre o início do fragmento e o início do cromossomo) ou da região intergênica associada ao fragmento. A fragmentação das regiões em porções de tamanhos iguais permite que elas contribuam com quantidade de informação equivalente aos seus tamanhos. No total 16.738.447 de fragmentos de 100 bp foram produzidos.

Para cada fragmento, calculamos a cobertura de *hits* (incluindo os dois tipos de *hits*) considerando somente a porção não mascarada dos fragmentos. Portanto, a cada fragmento foi atribuído um valor único de cobertura de *hits*.

Como mencionando anteriormente, as regiões de *breakpoints* tendem a apresentar um percentual de cobertura de *hits* menor do que em regiões que estão fora dos *breakpoints*. Para avaliar isso, fizemos uma verificação da cobertura média apresentada pelos fragmentos de 100 bp dentro e fora dos pontos de quebra identificados pelo `Cassis` com a lista de pares de genes ortólogos entre o homem e o camundongo. A Tabela 6.5 exibe, para cada cromossomo, o número de *breakpoints* encontrados pelo `Cassis` e os tamanhos médios destas regiões, antes e depois do processo de refinamento.

Utilizando as coordenadas dos *breakpoints* em cada um dos cromossomos, separamos os fragmentos de 100 bp em dois grupos: fragmentos dentro de *breakpoints* e fragmentos fora de *breakpoints*. Este processo foi realizado tanto com as coordenadas dos pontos de quebra não refinados como com as coordenadas dos pontos de quebra refinados. Feito isso, para cada conjunto foi calculada a média dos percentuais de cobertura dos fragmentos. A Tabela 6.6 exibe os resultados obtidos.

Analisando a Tabela 6.6 podemos confirmar que, de fato, os fragmentos situados dentro de *breakpoints* apresentam cobertura de *hits*, em geral, bem menor do que os fragmentos situados fora de *breakpoints*. A única exceção observada foi o cromossomo 22 que apresentou cobertura média maior em regiões localizadas no interior de pontos de quebra. O valor é mais do que duas vezes maior quando consideramos os pontos de quebra não refinados. Contudo, os valores são muito próximos quando olhamos para os *breakpoints* refinados. Não encontramos uma explicação clara para esta diferença, que pode estar ligada a uma particularidade deste cromossomo. Uma análise mais minuciosa das características do cromossomo 22 se faz necessária.

Tabela 6.5: *Breakpoints* identificados pelo `Cassis` com base na lista de pares genes ortólogos entre homem e camundongo. A tabela lista o número de *breakpoints* e tamanho médio dos *breakpoints*, em pares de bases, antes e após a segmentação.

	Número de	**Tamanho médio [bp]**	
Cromossomo	***Breakpoints***	**Antes segmentação**	**Após segmentação**
1	22	1.527.431,50	133.899,91
2	24	1.091.900,25	201.740,08
3	25	439.547,12	166.244,20
4	13	659.024,62	183.335,38
5	18	543.419,22	107.058,72
6	18	398.317,56	145.264,39
7	27	353.610,11	81.847,56
8	20	806.776,60	362.154,50
9	16	2.327.709,31	303.640,44
10	19	786.224,89	397.092,58
11	8	456.017,88	168.828,62
12	15	209.333,73	83.651,20
13	14	461.604,29	94.642,79
14	5	241.519,80	32.711,80
15	19	389.397,26	119.303,95
16	17	1.218.853,88	275.076,94
17	22	475.272,77	125.343,55
18	8	1.578.342,75	947.794,00
19	31	605.309,55	89.876,55
20	2	2.395.192,00	2.146.359,50
21	2	64.503,50	64.041,00
22	12	607.264,75	107.086,58
X	12	1.152.936,75	305.686,08
Todos	369	771.402,18	201.252,07

Tabela 6.6: Número de fragmentos de 100 bp e cobertura de *hits* média apresentada pelos fragmentos em regiões dentro e fora dos *breakpoints* identificadas pelo `Cassis` na comparação entre os genomas do homem e do camundongo. Os valores exibidos consideram os *breakpoints* antes e depois da segmentação.

	Antes segmentação				**Depois segmentação**			
	Número de Fragmentos		**Cobertura Média [%]**		**Número de Fragmentos**		**Cobertura Média [%]**	
Crom.	**Dentro**	**Fora**	**Dentro**	**Fora**	**Dentro**	**Fora**	**Dentro**	**Fora**
1	252.850	922.984	2,12	22,91	18.559	1.157.275	2,90	18,69
2	148.655	922.934	10,48	26,00	28.681	1.042.908	3,89	24,40
3	67.185	720.409	14,87	23,41	25.567	762.027	2,82	23,35
4	70.181	1.146.653	14,86	22,93	18.928	1.197.906	4,41	22,75
5	89.166	1.043.096	15,91	25,42	16.057	1.116.205	2,72	24,99
6	46.144	845.681	12,89	24,16	18.205	873.620	1,32	24,04
7	49.806	592.683	13,58	20,91	10.596	631.893	4,82	20,60
8	120.586	783.970	16,77	23,67	38.298	866.258	3,43	23,60
9	295.632	509.400	4,48	23,92	33.900	771.132	5,65	17,27
10	73.049	535.554	13,06	25,37	25.803	582.800	8,83	24,56
11	26.592	725.283	7,83	21,83	9.564	742.311	5,81	21,54
12	27.469	716.487	8,15	21,68	11.049	732.907	4,97	21,42
13	36.956	708.604	22,97	17,70	7.368	738.192	6,52	18,07
14	9.469	667.441	23,38	18,36	1.626	675.284	1,85	18,47
15	53.849	544.066	11,32	17,85	16.212	581.703	4,52	17,61
16	148.264	354.552	3,85	24,49	32.959	469.857	3,47	19,45
17	47.295	289.001	9,77	23,46	15.428	320.868	6,28	22,27
18	87.239	394.081	17,78	24,39	39.850	441.470	12,36	24,17
19	90.212	155.016	7,17	9,82	11.067	234.161	2,60	9,14
20	11.394	265.260	10,16	22,22	9.235	267.419	9,97	22,13
21	610	260.335	4,04	11,17	609	260.336	4,05	11,17
22	26.082	214.280	12,32	5,48	6.418	233.944	6,26	6,22
X	70.234	881.735	6,54	15,12	22.113	929.856	3,68	14,74
Todos	1.848.919	14.199.505	9,39	21,67	418.092	15.630.332	5,20	20,66

Olhando para as sequências refinadas, podemos ver que as médias das coberturas dos fragmentos localizados no interior e no exterior dos *breakpoints* são, respectivamente, 5,20% e 20,66%. Um argumento que poderia ser utilizado contra essa análise é que, como estamos usando *breakpoints* refinados, naturalmente eles possuem menor cobertura já que o processo de segmentação age de forma a estender ao máximo os blocos de sintenia, deixando apenas as regiões com alinhamentos menos significativos para compor os pontos de quebra. Contudo, se observarmos os fragmentos no interior do *breakpoints* não refinados, notaremos que eles também possuem uma cobertura média menor: 9,39% no interior contra 21,67% no exterior. Além disso, podemos notar que, quando refinamos os *breakpoints*, os fragmentos que não fazem mais parte do interior dos pontos de quebra são transferidos para o conjunto de fragmentos localizados no exterior dos pontos de quebra. Se este fragmentos que mudaram de conjunto possuíssem uma cobertura média alta, eles provocariam um aumento da média de cobertura do conjunto de fragmentos situados no exterior de pontos de quebra depois da segmentação. No entanto, o que vemos na maioria dos cromossomos é uma pequena diminuição da média. Isso indica que as regiões que flanqueiam os *breakpoints* também possuem uma cobertura de *hits* média menor do que no resto do genoma.

6.3.2 Janela deslizante

Os dados de cobertura de *hits* podem variar muito ao longo da sequência de um cromossomo apresentando curvas de valores muito ruidosas. Além disso, alguns cromossomos podem apresentar valores baixos de cobertura em alguns trechos e altos em outros e, apesar dessas diferenças, pontos de quebra podem ser localizados em ambos os tipos de regiões. Essa característica torna inviável a análise global da sequência cromossômica. A tarefa de definir um valor (ou um conjunto de valores) de filtro que seja capaz de identificar *breakpoints* em regiões tão distintas é extremamente difícil.

Baseado nessa observação, procuramos desenvolver um método de identificação de *breakpoints* em dois passos. Em um primeiro passo, identificamos regiões de interesse que apresentem um comportamento (curva de cobertura) relativamente uniforme. Em um segundo passo, estas regiões são efetivamente analisadas em busca de *breakpoints*. O primeiro passo será apresentado nesta seção enquanto o segundo passo será apresentado na próxima seção (Seção 6.3.3).

Uma maneira de tentar diminuir o ruído é adotar métodos de janelas deslizantes. Nestes métodos, uma janela de tamanho fixo é utilizada para calcular uma série de valores (média de cobertura, por exemplo) que caracterizem a sequência analisada. A janela define uma região que terá o valor calculado. Feito o cálculo, a janela é então deslocada em uma distância Δ pré-estabelecida e um novo valor é calculado para a nova região coberta por ela. O processo é repetido até que toda a sequência seja percorrida. Conforme a distância de deslocamento, as janelas podem apresentar sobreposição ou não.

Ao invés de deslizar uma janela sobre a sequência do cromossomo, optamos por aplicar a técnica na sua sequência de fragmentos de 100 bp provenientes das regiões intergênicas. Segundo essa abordagem, uma janela tem seu tamanho mensurado pelo número de fragmentos contidos nela. Como nossos fragmentos têm tamanho fixo e não apresentam sobreposições, o tamanho da janela está diretamente relacionado ao número de bases que ela cobre no cromossomo. Note que essas bases não são necessariamente contíguas já que dois fragmentos consecutivos podem conter um gene entre eles.

Se n é o número de fragmentos de um cromossomo C_r do genoma G_r, para um dado fragmento f_i $(1 \leq i \leq n)$, podemos definir uma janela de tamanho $2w + 1$ centralizada no próprio fragmento. Como f_i é o centro da janela, ela cobre todos os fragmentos entre $f_{(i-w)}$ e $f_{(i+w)}$.

A janela é utilizada para calcular o valor de cobertura média de todos os fragmentos contidos em seu interior. O valor calculado é, então, associado ao fragmento f_i que a define. Este procedimento é feito para todos os fragmentos f_i $(1 \leq i \leq n)$ e, portanto, o valor Δ de deslocamento da janela é igual a 1.

O objetivo aqui é utilizar a janela deslizante para a produção de uma curva de cobertura média. Desejamos obter uma curva que reduza o ruído geral e permita a identificação de regiões de interesse. Neste ponto, a definição do tamanho da janela é de extrema importância. Quanto maior o tamanho da janela, mais suave será a curva de cobertura. Por outro lado, quanto maior o comprimento da janela, maior será a quantidade de informações, provenientes de diferentes regiões, incorporada para o cálculo da média. A união de informações de regiões muito distintas podem resultar no mascaramento de características que estão sendo procuradas na curva.

Outro critério a ser definido refere-se à maneira como a curva será analisada para identificação das regiões de interesse. Aqui, devemos lembrar que a meta é obter trechos que apresentem comportamento relativamente uniforme (sem variações

extremas) e que, ao mesmo tempo, auxiliem a identificação de *breakpoints*.

Diante deste aspecto, optamos por utilizar um critério bem simples. O valor médio das médias de cobertura observadas em todas as janelas do cromossomo é calculado. Feito isso, este valor é utilizado como critério de particionamento da curva em regiões que se alternam: regiões que estão acima da média e regiões que estão abaixo da média. Como as regiões de *breakpoints* costumam apresentar, em geral, uma cobertura menor do que em outras regiões, estamos, em um primeiro momento, interessados apenas nas regiões que estão situadas abaixo da média geral do cromossomo.

Para avaliar este algoritmo, realizamos o processamento de todos os cromossomos do genoma humano com diferentes valores para o parâmetro w: 50, 125, 250, 500, 1.000, 1.500, 2.000 e 2.500. A Tabela 6.7 exibe o número de regiões com coberturas abaixo da média obtidas em cada cromossomo conforme o valor utilizado para w.

A Tabela 6.7 mostra que o valor de w tem fator determinante no número de regiões que são definidas quando utilizamos o critério proposto. Com uma janela definida por $w = 50$, um total de 44.519 regiões são identificadas em todo o genoma humano. Este número cai quase pela metade quando aumentamos o valor de w para 125 e pode chegar a apenas 1.773 regiões quando se adota o valor 2.500. Isto é um efeito direto da suavização que as janelas maiores produzem na curva de cobertura: quanto maior a janela, menor a diferença de média de cobertura entre janelas consecutivas.

Um conjunto de regiões adequado é aquele que fornece informações suficientes sobre os pontos de quebra existentes no genoma. Para avaliar isso, comparamos cada um dos conjuntos obtidos com as coordenadas dos *breakpoints* refinados e não refinados definidos com o programa `Cassis`. Contamos todas as regiões que apresentaram algum tipo de "contato" com os pontos de quebra: regiões que cobrem um *breakpoint*, regiões contidas em um *breakpoint* ou regiões que apresentam sobreposições em uma das extremidades dos *breakpoints*. A Tabela 6.8 mostra os números obtidos para cada um dos valores de w.

Os dados da Tabela 6.8 mostram que quanto maior for o valor de w, menor será o número de regiões que apresentam algum contato com *breakpoints* definidos por `Cassis`. Para $w = 50$, quando consideramos os pontos de quebra não refinados, um total de 3.030 regiões estão relacionadas a 365 *breakpoints* resultando em uma taxa de 8,30 regiões por *breakpoint*. Se aumentamos o valor de w para 2.500, obtemos apenas 231 regiões que tocam 303 pontos de quebra (0,76 região por *breakpoint*), o que indica que uma região pode cobrir mais de um *breakpoint*. Apesar das diferenças entre

Tabela 6.7: Número de regiões identificadas, por cromossomo, com o algoritmo de janela deslizante proposto. Os valores de w iguais a 50, 125, 250, 500, 1.000, 1.500, 2.000 e 2.500 foram utilizados para produção de curvas de cobertura. Para cada curva produzida, o valor médio da cobertura de todas as janelas do cromossomo foi calculado. Este valor é utilizado para particionar a sequência e selecionar as regiões que possuem cobertura abaixo da média.

	Parâmetro w – Tamanho da janela deslizante							
Crom.	**50**	**125**	**250**	**500**	**1.000**	**1.500**	**2.000**	**2.500**
1	2.828	1.401	702	381	232	178	134	116
2	3.347	1.681	939	520	281	206	136	123
3	2.437	1.246	704	390	238	144	114	107
4	4.023	2.111	1.224	649	408	322	231	198
5	3.626	1.899	1.124	647	354	271	212	173
6	2.777	1.451	814	413	257	188	147	111
7	1.864	974	539	301	156	107	79	60
8	2.987	1.556	841	506	282	225	153	136
9	1.759	835	493	295	113	71	66	47
10	2.017	1.060	654	362	186	150	109	93
11	2.060	1.042	621	341	186	162	117	68
12	2.350	1.095	618	351	189	131	103	106
13	1.824	881	494	248	170	133	95	82
14	1.586	763	416	223	111	64	44	52
15	1.053	486	243	129	82	45	34	34
16	1.191	536	208	96	63	47	44	30
17	1.071	498	289	153	90	47	52	38
18	1.475	773	447	217	160	120	118	76
19	549	272	171	86	58	35	22	16
20	840	395	221	117	52	23	15	21
21	469	221	125	64	38	30	18	10
22	253	101	57	25	9	5	4	3
X	2.133	1.051	616	350	210	154	117	73
Todos	44.519	22.328	12.560	6.864	3.925	2.858	2.164	1.773

Tabela 6.8: Contagem de regiões que apresentaram algum tipo de sobreposição com os *breakpoints* não refinados e refinados definidos pelo `Cassis`. A coluna **R** indica o número de regiões que apresentaram algum tipo de sobreposição, a coluna **PQ** indica o número de pontos de quebra únicos que foram tocados por ao menos uma região, a coluna **R/PQ** indica a razão entre os valores das colunas **R** e **PQ** e, finalmente, a coluna **%CASSIS** indica qual a porcentagem dos 369 pontos de quebra identificados pelo `Cassis` que foram tocados pelas regiões selecionadas.

	***Breakpoints* não refinados**				***Breakpoints* refinados**			
w	**R**	**PQ**	**R/PQ**	**%CASSIS**	**R**	**PQ**	**R/PQ**	**%CASSIS**
50	3.030	365	8,30	98,92	721	345	2,09	93,50
125	1.612	359	4,49	97,29	499	342	1,46	92,68
250	966	356	2,71	96,48	397	340	1,17	92,14
500	605	344	1,76	93,22	309	326	0,95	88,35
1.000	394	327	1,20	88,62	252	314	0,80	85,09
1.500	313	319	0,98	86,45	228	302	0,75	81,84
2.000	284	312	0,91	84,55	215	297	0,72	80,49
2.500	231	303	0,76	82,11	187	289	0,65	78,32

os conjuntos serem menores, este comportamento também pode ser visto quando consideramos os *breakpoints* refinados. Este comportamento é reflexo do aumento do tamanho médio das regiões indentificadas conforme o valor de w cresce. A Tabela 6.9 mostra, para cada valor de w, os tamanhos médios de todas regiões encontradas e os tamanhos médios das regiões que "tocam" os *breakpoints* identificados pelo `Cassis`.

Tabela 6.9: Número de regiões identificadas com o algoritmo de janela deslizante e tamanho médio apresentado por elas. A coluna **N** indica o número de regiões encontradas e a coluna **Tam. Médio** exibe o tamanho médio das regiões em bp para os conjuntos formados por: 1 - todas as regiões, 2 - regiões que sobrepõem *breakpoints* não refinados e 3 - regiões que sobrepõem *breakpoints* refinados.

	Todas Regiões		**Regiões que sobrepõem *Breakpoints***			
			Não refinados		**Refinados**	
w	**N**	**Tam. Médio**	**N**	**Tam. Médio**	**N**	**Tam. Médio**
50	44.519	34.638,11	3.030	106.013,14	721	218.154,40
125	22.328	70.148,76	1.612	234.889,26	499	450.095,09
250	12.560	125.752,78	966	473.895,49	397	846.493,83
500	6.864	230.082,45	605	906.357,21	309	1.432.671,92
1.000	3.925	396.736,26	394	1.723.743,07	252	2.347.931,66
1.500	2.858	538.159,22	313	2.352.610,77	228	3.012.524,53
2.000	2.164	699.629,30	284	2.740.320,87	215	3.400.745,08
2.500	1.773	847.470,08	231	3.544.929,66	187	4.136.547,95

Os *breakpoints* "perdidos" são segmentos que estão localizadas dentro de regiões que apresentaram cobertura média maior do que a cobertura média de todas as janelas. Neste ponto, nos questionamos se seria possível reduzir essa perda. Mesmo utilizando um valor baixo para o parâmetro w (50), 6,5% dos pontos de quebra refinados não apresentaram sobreposições com as regiões definidas.

Uma solução simples a ser adotada, seria a inclusão das regiões com médias de coberturas maiores do que a média geral do cromossomo. Contudo, optamos por uma abordagem diferente.

A nova abordagem consiste na execução do algoritmo de janela deslizante em dois níveis. No primeiro nível, uma janela de tamanho w_1 é utilizada para processar toda a lista de fragmentos do cromossomo e a média das janelas é utilizada para fragmentar a lista em regiões que estão acima e abaixo da média. A diferença para

a abordagem anterior, está no fato de que não descartamos as regiões com cobertura de *hits* acima da média.

Cada uma das regiões definidas, com coberturas abaixo e acima da média, são submetidas a uma nova janela deslizante de tamanho definido pelo parâmetro w_2. Novamente, calculamos a média das coberturas das janelas, porém, apenas dentro da região e não no cromossomo inteiro. O valor de média calculado é utilizado para particionar a região em trechos com cobertura maior e menor do que a média apresentada por ela. Feito isso, apenas as regiões com cobertura menores do que a média são mantidas.

A intenção desta abordagem é aumentar a fragmentação do cromossomo com o objetivo de potencializar a chance de cobrir todos os *breakpoints*, mas mantendo foco na busca por regiões que, dentro de um contexto local, apresentam um cobertura menos significativa do que as regiões no entorno.

Para avaliar esta nova abordagem, decidimos utilizar o valor $w_1 = 500$ com base em dois fatores principais. O primeiro fator está no número de regiões que foram produzidas na primeira etapa e que serão produzidas na segunda etapa. No genoma humano inteiro, o algoritmo de janela deslizante configurado com $w_1 = 500$ identificou 6.864 regiões de baixa cobertura. Se somarmos a este número as regiões de alta cobertura, teremos um total de 13.705 regiões. O uso da segunda janela deslizante sobre estas regiões provocará uma fragmentação maior. Se escolhermos um valor baixo para w_1, corremos o risco de observar uma explosão de fragmentos muito pequenos que podem dificultar a caracterização do comportamento da cobertura de *hits* ao longo do cromossomo. Em um caso extremo, o uso de um valor muito baixo para w_1 pode resultar em um particionamento formado por regiões com *hits* e regiões sem *hits* (ou com pouquíssimos *hits*).

O segundo fator se refere ao número de *breakpoints* identificados pelo `Cassis` que já apresentaram alguma relação com as regiões de baixa cobertura. O conjunto de regiões obtidos com $w_1 = 500$ teve sobreposição com mais de 90% dos *breakpoints* não refinados e com quase 90% dos *breakpoints* refinados, cifras maiores do que as exibidas pelos conjuntos produzidos com $w_1 > 500$.

Os seguintes valores de w_2 foram utilizados para testar a segunda etapa de janela deslizante: 62, 125, 250, 375, 500, 625 e 750. Procuramos testar valores acima e abaixo do valor de w_1 para avaliar o efeito causado na produção de regiões. A Tabela 6.10 lista o número de regiões identificadas em cada cromossomo.

Comparando as Tabelas 6.7 e 6.10, podemos ver claramente o aumento de regiões

Tabela 6.10: Número de regiões com cobertura abaixo da média identificadas, em cada cromossomo, pelo algoritmo que possui duas etapas de janela deslizante. A primeira etapa utilizou $w_1 = 500$ e identificou regiões com alta e baixa cobertura ao longo do cromossomo. A segunda etapa processou as regiões identificadas pela primeira etapa e identificou sub-regiões com coberturas de *hits* menores do que a apresentada pela região que as contém. Os seguintes valores de w_2 foram testados: 62, 125, 250, 375, 500, 625 e 750.

	Parâmetro w_2 – Tamanho da janela deslizante						
Crom.	**62**	**125**	**250**	**375**	**500**	**625**	**750**
1	3.862	2.869	2.229	2.032	2.105	1.776	1.693
2	4.539	3.416	2.678	2.534	2.709	2.249	2.126
3	3.388	2.492	2.128	1.875	2.054	1.672	1.611
4	5.282	3.991	3.288	3.094	3.372	2.721	2.671
5	5.123	3.889	3.173	2.985	3.320	2.700	2.567
6	3.723	2.787	2.204	2.129	2.225	1.912	1.781
7	2.530	1.924	1.519	1.368	1.526	1.294	1.254
8	4.028	3.092	2.557	2.402	2.611	2.292	2.133
9	2.586	1.919	1.541	1.356	1.428	1.206	1.131
10	2.826	2.230	1.782	1.677	1.841	1.519	1.459
11	2.991	2.182	1.784	1.617	1.681	1.461	1.342
12	3.053	2.290	1.780	1.650	1.755	1.459	1.397
13	2.517	1.870	1.458	1.271	1.449	1.134	1.093
14	2.130	1.517	1.120	1.090	1.185	1.006	961
15	1.589	1.146	787	722	724	623	572
16	1.724	1.194	870	732	686	573	501
17	1.381	1.097	822	744	798	701	638
18	2.023	1.492	1.191	1.097	1.224	1.050	976
19	696	501	403	372	418	330	300
20	1.186	852	588	539	599	506	455
21	605	440	358	305	322	271	257
22	384	237	184	153	158	122	105
X	3.083	2.244	1.837	1.665	1.722	1.431	1.370
Todos	61.249	45.671	36.281	33.409	35.912	30.008	28.393

identificadas. Em relação aos outros aspectos, as tabelas são similares: quanto maior o tamanho da janela, menor o número de regiões criadas.

Reproduzindo a análise feita na Tabela 6.8, a Tabela 6.11 traz a contagem de regiões que apresentaram algum contato com os *breakpoints* refinados e não refinados produzidos pelo `Cassis`. Os resultados mostram que, de fato, o passo adicional não promoveu um aumento na porcentagem de *breakpoints* que possuem alguma sobreposição com as regiões. Isso pode ser uma consequência direta da redução do tamanho médio das regiões de baixa cobertura. As regiões de baixa cobertura identificadas com $w_1 = 500$ possuíam tamanho médio em torno de 230 kbp. Contudo, como podemos observar na Tabela 6.12, as regiões produzidas com as duas janelas deslizantes possuem tamanho médio que variam entre 25 kbp e 57kbp, conforme o valor de w_2 utilizado.

Tabela 6.11: Contagem de regiões, produzidas pelo algoritmo que possui duas etapas de janela deslizante ($w_1 = 500$), que apresentaram algum tipo de sobreposição com os *breakpoints* (não refinados e refinados) listados pelo `Cassis`. A coluna **R** indica o número de regiões que apresentaram algum tipo de sobreposição, a coluna **PQ** indica o número de pontos de quebra únicos que foram tocados por ao menos uma região, a coluna **R/PQ** indica a razão entre os valores das colunas **R** e **PQ** e, finalmente, a coluna **%CASSIS** indica qual a porcentagem dos 369 pontos de quebra identificados pelo `Cassis` que foram tocados pelas regiões selecionadas.

	***Breakpoints* não refinados**				***Breakpoints* refinados**			
w_2	**R**	**PQ**	**R/PQ**	**%CASSIS**	**R**	**PQ**	**R/PQ**	**%CASSIS**
62	4.174	363	11,50	98,37	959	345	2,78	93,50
125	3.088	362	8,53	98,10	752	338	2,22	91,60
250	2.448	354	6,92	95,93	625	325	1,92	88,08
375	2.201	355	6,20	96,21	601	323	1,86	87,53
500	2.348	348	6,75	94,31	595	310	1,92	84,01
625	1.981	340	5,83	92,14	534	302	1,77	81,84
750	1.804	333	5,42	90,24	514	292	1,76	79,13

Os resultados apresentados até aqui não fornecem dados suficientes para a definição de qual estratégia e configuração de parâmetros é mais adequada para a identificação dos *breakpoints*.

Apesar de procurarmos maximizar o número de *breakpoints* identificados pelo

Tabela 6.12: Número de regiões identificadas com o algoritmo que possui duas etapas de janela deslizante ($w_1 = 500$) e tamanho médio apresentado por elas. A coluna **N** indica o número de regiões encontradas e a coluna **Tam. Médio** exibe o tamanho médio das regiões em bp para os conjuntos formados por: 1 - todas as regiões, 2 - regiões que apresentam sobreposições com *breakpoints* não refinados e 3 - regiões que apresentam sobreposições com *breakpoints* refinados.

	Todas Regiões		**Regiões que sobrepõem *Breakpoints***			
			Não refinados		**Refinados**	
w_2	**N**	**Tam. Médio**	**N**	**Tam. Médio**	**N**	**Tam. Médio**
62	61.249	25.258,43	4.174	62.478,06	959	111.322,10
125	45.671	34.654,35	3.088	85.890,55	752	159.834,95
250	36.281	44.730,96	2.448	112.451,47	625	214.511,24
375	33.409	48.171,38	2.201	125.804,81	601	237.013,12
500	35.912	45.410,32	2.348	118.574,32	595	243.054,58
625	30.008	54.149,58	1.981	142.131,92	534	282.901,84
750	28.393	57.784,71	1.804	161.514,70	514	306.651,42

`Cassis` que apresentam alguma sobreposição com as regiões definidas, a identificação de *breakpoints* é, de fato, feita na próxima fase. Assim, os conjuntos produzidos foram utilizados como dados de entrada nos testes realizados com o método que será descrito na próxima seção.

6.3.3 Encadeamento de *hits*

Uma vez que as regiões que devem ser analisadas são identificadas, o próximo passo é verificar como elas se relacionam entre si. Muitas das regiões são consecutivas e colineares ao longo da sequência cromossômica. Contudo, as regiões que estão em torno dos *breakpoints* não apresentam estas propriedades.

Para tentar avaliar a colinearidade das regiões, imaginamos uma abordagem que realiza o encadeamento dos *hits* contidos dentro delas.

O procedimento de seleção de *hits* que serão encadeados funciona da seguinte maneira: Para cada região definida com o algoritmo de janela deslizante, obtemos a lista de *hits* processados (sem sobreposição). Desta lista, apenas os *hits* que possuem $\sigma_o \in \{-1, +1\}$ são considerados. Em um primeiro momento, não desejamos

a influência de *hits* com $\sigma_o = 0$ pois eles indicam regiões com múltiplos *hits* que podem estar relacionadas às duplicações ou às regiões de baixa complexidade e, por isso, podem dificultar a operação de encadeamento.

Se n_{C_r} é o número de regiões definidas pelo algoritmo de janela deslizante para o cromossomo C_r no genoma G_r, para cada região r_i, $(1 \leq i \leq n_{C_r})$, o seguinte procedimento é adotado:

1. Um par (σ_o, C_o) indica uma orientação no cromossomo C_o. Para cada par (σ_o, C_o) existente na região, calculamos qual a porcentagem das bases cobertas por *hits* dentro da região que são associadas a ele (soma de bases de *hits* associado ao par dividido pela soma total de bases cobertas por *hits*);

2. Os diferentes pares (σ_o, C_o) são ordenados em ordem decrescente de porcentagem de cobertura.

3. Se a soma de bases cobertas por *hits* do par (σ_o, C_o) mais representado for menor do que um valor s_{min}, todos os *hits* da região são descartados. Caso contrário, uma análise da composição dos *hits* é feita para decidir quais serão selecionados:

   (a) Se a porcentagem apresentada pelo segundo par da lista ordenada for maior ou igual a $p2_{min}$, os *hits* do primeiro e do segundo par são selecionados para a próxima fase do processo. Caso contrário, apenas os *hits* do primeiro par são selecionados.

O objetivo dos passos descritos acima é promover uma filtragem dos *hits*. Dentro de uma região, *hits* com regiões de diversos cromossomos do genoma G_o podem estar presentes. Muitos destes *hits* podem ser espúrios e nestes casos, estamos assumindo que eles representam uma pequena parcela da quantidade de *hits* total da região.

Em nossos testes, definimos $p2_{min} = 10\%$. Isto significa que, se um único par (σ_o, C_o) responde por mais de 90% da cobertura de *hits* da região, apenas este par nos interessa. Caso contrário, acreditamos que o segundo par mais representado pode fornecer pistas sobre um possível *breakpoint* dentro da região. Se a região não for muito grande, a chance de existirem dois *breakpoints* dentro dela cai muito e, por isso, nessa abordagem optamos por não selecionar o terceiro par (σ_o, C_o) mais representado.

O valor de s_{min} foi arbitrariamente definido em 100. Com este valor, esperamos filtrar *hits* demasiadamente pequenos e, ao mesmo tempo, evitar um descarte exagerado de *hits*.

Depois do processo de filtragem, todos os *hits* selecionados de todas as regiões são posicionados ao longo da sequência do cromossomo C_r. Todos os pares de *hits* que são consecutivos e estão associados à mesma trinca (C_o, I_o, σ_o) são unidos para formar um único *hit*: o novo *hit* tem suas posições inicial e final, respectivamente, iguais a posição inicial do primeiro *hit* e a posição final do segundo *hit*.

Se m_{C_r} é o número de *hits* (após a filtragem) existentes ao longo da sequência do cromossomo C_r do genoma G_r, então para cada *hit* h_i, $(1 < i \leq m_{C_r})$, um valor de distância Δ_i em relação ao *hit* $h_{(i-1)}$ é calculado da seguinte maneira (o *hit* h_1 recebe $\Delta_1 = 0$):

- $\Delta_i = I_{o(i)} - I_{o(i-1)}$, se os cromossomos e as orientações de $h_{(i-1)}$ e h_i são iguais;
- $\Delta_i = \infty$, caso contrário.

Uma vez que os valores de distância Δ foram calculados para todos os *hits*, podemos realizar o encadeamento. Para realizar este procedimento, um valor inteiro positivo Δ_{max} é definido para indicar a máxima distância permitida entre dois *hits* consecutivos e, novamente, a lista de *hits* é percorrida do início ao fim.

Em cada passo da análise, os *hits* $h_{(i-1)}$, h_i e $h_{(i+1)}$ são comparados e o encadeamento ou definição de *breakpoints* são feitos do seguinte modo:

1. Se $|\Delta_i| > \Delta_{max}$, temos um possível candidato a *breakpoint* e uma das seguintes situações podem ocorrer:

   I. $h_{(i-1)}$ e h_i possuem diferentes cromossomos $(\Delta_i = \infty)$;

   II. $h_{(i-1)}$ e h_i possuem mesmo cromossomo C_o, mas suas orientações são diferentes $(\Delta_i = \infty)$;

   III. $h_{(i-1)}$ e h_i possuem mesmo cromossomo C_o e mesma orientação, mas possuem distância maior do que a permitida $(\Delta_i = I_{o(i)} - I_{o(i-1)})$;

   Estas situações são analisadas para identificação de *breakpoints*. As verificações (a) e (b) descritas a seguir são aplicadas nas situações I e II. No caso da situação III, apenas a verificação (b) é necessária.

(a) Se os cromossomos e as orientações dos *hits* $h_{(i-1)}$ e $h_{(i+1)}$ são iguais, consideramos h_i como um *hit* intruso. Ele é eliminado da lista e o valor de $\Delta_{(i+1)}$ é atualizado com a distância entre $h_{(i+1)}$ e $h_{(i-1)}$.

(b) Caso contrário, verificamos se os cromossomos e as orientações dos *hits* h_i e $h_{(i+1)}$ são iguais:

i. Se forem iguais e $|\Delta_{(i+1)}| \leq \Delta_{max}$, isso significa que o *hit* $h_{(i+1)}$ será encadeado com o *hit* h_i e, portanto, eles iniciarão uma nova cadeia. Logo, podemos definir um *breakpoint* localizado entre os *hits* $h_{(i-1)}$ e h_i.

ii. Caso contrário, consideramos h_i como um *hit* intruso. Ele é eliminado da lista e o valor de $\Delta_{(i+1)}$ é atualizado com a distância entre $h_{(i+1)}$ e $h_{(i-1)}$.

2. Caso contrário, os *hits* são encadeados.

O procedimento de encadeamento adotado é bem simples. Combinado com o processo de filtragem de *hits*, nossa expectativa é que ele seja capaz de identificar *breakpoints* baseados nos *hits* contidos nas regiões definidas pelo algoritmo de janela deslizante.

Em nossos testes, adotamos $\Delta_{max} = 10$ para processar a lista de regiões produzidas pelos algoritmos de janela deslizante (um e dois níveis de janela). A Tabela 6.13 mostra o tamanho médio dos *breakpoints* identificados em cada um dos conjuntos quando processados pela nossa metodologia de encadeamento.

A Tabela 6.13 mostra que o método de encadeamento foi capaz de produzir uma forte redução do número de regiões a serem analisadas. Contudo, os números de *breakpoints* determinados variam entre 660 e 854, conforme o conjunto de regiões, definidas pelas janelas deslizantes, que foi processado. Estes números giram em torno do dobro do número de pontos de quebra definidos pelo `Cassis` com o uso de informação de ortologia (369 *breakpoints*).

Um comportamento interessante que podemos observar na Tabela 6.13 é que quando utilizamos o algoritmo composto por apenas uma fase de janela deslizante, quanto maior o valor do parâmetro w_1, maior o tamanho médio dos *breakpoints* produzidos. Isso não é observado quando incluímos a segunda fase de janela deslizante pois o parâmetro w_2 não possui uma relação direta com o tamanho médio dos *breakpoints* obtidos.

Tabela 6.13: Tamanho médio dos *breakpoints* identificados com o método de encadeamento de *hits*. Os *hits* de regiões que foram listadas pelo algoritmos de janela deslizante de uma fase (w_1) e duas fases (w_1, w_2) foram processados pelo método de encadeamento de *hits* com os parâmetros $s_{min} = 100$, $p2_{min} = 10\%$ e $\Delta_{max} = 10$. A tabela lista o número de *breakpoints* encontrados e o tamanho médio deles quando consideramos todos os conjuntos: 1 - todos os *breakpoints*, 2 - *breakpoints* que possuem sobreposição com os *breakpoints* não refinados do `Cassis` e 3 - *breakpoints* que possuem sobreposição com os *breakpoints* refinados do `Cassis`.

		Todas Regiões		**Regiões que sobrepõem *Breakpoints***			
				Não refinados		**Refinados**	
w_1	w_2	**N**	**Tam. Médio**	**N**	**Tam. Médio**	**N**	**Tam. Médio**
50	–	854	462.637,47	395	610.446,33	337	659.993,72
125	–	824	517.042,98	394	666.372,99	330	695.986,28
250	–	792	699.196,58	389	781.108,29	328	748.410,97
500	–	812	952.769,98	392	897.779,15	319	973.166,91
1.000	–	792	1.377.829,18	368	1.227.262,19	312	1.335.607,56
1.500	–	751	1.618.951,81	352	1.337.639,29	300	1.433.680,96
2.000	–	705	1.932.532,93	343	1.618.442,47	296	1.746.004,34
2.500	–	660	2.167.234,75	329	1.781.061,66	286	1.923.092,39
500	62	715	718.495,51	349	1.060.900,32	307	1.170.077,26
500	125	738	681.725,59	370	975.900,14	330	1.067.417,93
500	250	762	630.783,17	382	859.718,71	329	927.535,59
500	375	773	607.706,55	383	811.820,54	330	870.703,02
500	500	773	667.386,95	381	903.025,03	333	920.856,45
500	625	764	637.223,91	375	845.790,24	331	824.368,81
500	750	745	652.436,38	385	805.320,02	335	783.165,42

Os tamanhos médios dos *breakpoints* que possuem algum contato com os *breakpoints* não refinados do `Cassis` variam entre 610 kbp e 1,78 Mbp para os conjuntos de dados que utilizam apenas uma fase de janela deslizante e entre 805 kbp e 1,06 Mbp para os conjuntos que utilizam duas fases. Estes valores são muito maiores do que o tamanho médio de 200 kbp dos *breakpoints* refinados do `Cassis`, mas alguns deles são próximos dos 771 kbp de tamanho médio apresentado pelos pontos de quebra não refinados. Olhando os *breakpoints* que sobrepõem os pontos de quebra refinados do `Cassis`, o tamanho médio varia entre 659 kbp e 1,92 Mbp para os conjuntos que utilizam uma fase de janela deslizante e 783 kbp e 1,17 Mbp para os conjuntos que utilizam duas fases de janela deslizante.

A Tabela 6.14 mostra os resultados da contagem de *breakpoints* que apresentam sobreposição com pontos de quebra refinados e não refinados do `Cassis`. Uma análise rápida dos dados desta tabela mostra que os *breakpoints* produzidos apresentaram sobreposições com uma boa porcentagem dos pontos de quebra do `Cassis`. De todos os conjuntos testados, o que apresentou melhor resultado foi a execução do algoritmo de janela deslizante em duas fases configurado com parâmetros $w_1 = 500$ e $w_2 = 125$: sobreposições com 97,02% dos pontos de quebra não refinados e com 94,85% dos pontos de quebra refinados.

A Tabela 6.14 foi construída da mesma forma que as Tabelas 6.8 e 6.11 e apenas verifica a questão da sobreposição das sequências. Ela não nos fornece nenhuma informação adicional que nos permita verificar se os *breakpoints* definidos por nossa metodologia são válidos.

Para averiguar esse aspecto, fizemos uma nova contagem de *breakpoints* para caracterizar diversas situações. Para facilitar a descrição, o conjunto de *breakpoints* do `Cassis` será denominado `C` e o conjunto de *breakpoints* definido com nossa metodologia será denominado `E`:

- **P**: Número de *breakpoints* de `C` que foram "perdidos", isto é, que não foram detectados no conjunto `E`;
- **CE1**: Número de *breakpoints* de `C` que possuem em seu interior um único ponto de quebra de `E` (*breakpoint* de `C` cobre inteiramente um único *breakpoint* de `E`);
- **EC1**: Número de *breakpoints* de `E` que possuem em seu interior um único ponto de quebra de `C` (*breakpoint* de `C` é coberto inteiramente por um único *breakpoint* de `E`);

Tabela 6.14: Contagem de *breakpoints* definidos com os algoritmos de janela deslizante que sobrepõem os pontos de quebra do `Cassis`. A coluna **R** indica o número de regiões que apresentaram algum tipo de sobreposição, a coluna **PQ** indica o número de pontos de quebra únicos que foram tocados por ao menos um *breakpoint*, a coluna **R/PQ** indica a razão entre os valores das colunas **R** e **PQ** e, finalmente, a coluna **%CASSIS** indica qual a porcentagem dos 369 pontos de quebra identificados pelo `Cassis` que foram tocados pelos *breakpoints* de cada conjunto testado.

		***Breakpoints* não refinados**				***Breakpoints* refinados**			
w_1	w_2	**R**	**PQ**	**R/PQ**	**%CASSIS**	**R**	**PQ**	**R/PQ**	**%CASSIS**
50	–	395	346	1,14	93,77	337	336	1,00	91,06
125	–	394	347	1,14	94,04	330	332	0,99	89,97
250	–	389	341	1,14	92,41	328	330	0,99	89,43
500	–	392	341	1,15	92,41	319	329	0,97	89,16
1.000	–	368	342	1,08	92,68	312	336	0,93	91,06
1.500	–	352	340	1,04	92,14	300	332	0,90	89,97
2.000	–	343	341	1,01	92,41	296	335	0,88	90,79
2.500	–	329	332	0,99	89,97	286	328	0,87	88,89
500	62	349	354	0,99	95,93	307	344	0,89	93,22
500	125	370	358	1,03	97,02	330	350	0,94	94,85
500	250	382	355	1,08	96,21	329	345	0,95	93,50
500	375	383	348	1,10	94,31	330	341	0,97	92,41
500	500	381	348	1,09	94,31	333	340	0,98	92,14
500	625	375	348	1,08	94,31	331	345	0,96	93,50
500	750	385	350	1,10	94,85	335	347	0,97	94,04

- **CEN**: Número de *breakpoints* de `C` que possuem em seu interior mais de um *breakpoint* de `E` (*breakpoint* de `C` cobre inteiramente mais de um *breakpoint* de `E`);
- **#CEN**: Número de *breakpoints* de `E` que são cobertos pelos *breakpoints* de `C` que foram contados em **CEN**;
- **ECN**: Número de *breakpoints* de `E` que possuem em seu interior mais de um *breakpoint* de `C` (*breakpoint* de `E` cobre inteiramente mais de um *breakpoint* de `C`);
- **#ECN**: Número de *breakpoints* de `C` que são cobertos pelos *breakpoints* de `E` que foram contados em **ECN**;
- **Extra**: Número de *breakpoints* que foram listados em `E` mas que não possuem qualquer relação de sobreposição com os *breakpoints* de `C`.

Os valores descritos acima foram coletados para todos os conjunto de testes considerando as coordenadas dos *breakpoints* não refinados e refinados do `Cassis`. A Tabela 6.15 exibe os resultados obtidos com os pontos de quebra não refinados e a Tabela 6.16 lista os valores computados com os pontos de quebra refinados.

Estes números foram coletados com o objetivo de melhor caracterizar os *breakpoints* do conjunto `E`: identificar perdas de *breakpoints* em relação ao conjunto `C` e identificar *breakpoints* que sejam equivalente entre os dois conjuntos.

As contagens **CE1**, **EC1**, **CEN** e **ECN** consideram apenas os casos em que *breakpoints* estão contidos completamente dentro do *breakpoint* do outro conjunto. Assim, a tabela não mostra diretamente uma contagem de *breakpoint* que se sobrepõem apenas nas extremidades.

Com os valores **CE1** e **EC1** procuramos identificar as situações onde os *breakpoints* dos dois conjuntos estão "cercando" uma mesma região no genoma. Os valores **CEN** e **ECN** foram computados para detectar situações em que um *breakpoint* de um conjunto pode possuir um possível rearranjo interno (pontos de quebra grandes o suficiente para cobrirem outro possíveis *breakpoints*).

O nosso objetivo é, principalmente minimizar as perdas (**P**) e maximizar os *breakpoints* equivalentes (**CE1** e **EC1**).

A Tabela 6.15 mostra que quando utilizamos as coordenadas dos *breakpoints* não refinados do conjunto `C`, as perdas de *breakpoints* (**P**) estão entre 22 e 37 para os

Tabela 6.15: Relações entre *breakpoints* não refinados identificados pelo `Cassis` e os *breakpoints* identificados com o processo de encadeamento de *hits* de regiões identificadas com o algoritmo de janela deslizante. Para facilitar a descrição, o conjunto de *breakpoints* do `Cassis` será denominado `C` e o conjunto de *breakpoints* produzido segundo nossa metodologia será denominado `E`. A coluna **P** indica o número de *breakpoints* de `C` que não foram detectados em `E`. A coluna **CE1** mostra o número de *breakpoints* de `C` que contém em seu interior um único *breakpoint* de `E`. A coluna **EC1** indica o número de *breakpoints* de `E` que contém em seu interior um único *breakpoint* de `C`. A coluna **1-1** representa a soma dos valores das colunas **CE1** e **EC1** e o valor dentro dos parênteses indica a porcentagem dos *breakpoints* do conjunto `C` (369 *breakpoints*) que são cobertos por esta soma. A coluna **CEN** exibe o número de *breakpoints* de `C` que contém mais de um *breakpoint* de `E` em seu interior e a coluna **#CEN** indica qual a quantidade de pontos de quebra de `E` afetados. Analogamente, a coluna **ECN** exibe o número de *breakpoints* de `E` que contém mais de um *breakpoint* de `C` em seu interior e a coluna **#ECN** indica qual a quantidade de pontos de quebra de `C` afetados. Finalmente, a coluna **Extra** indica o número de *breakpoints* do conjunto `E` que não possuem qualquer "contato" com *breakpoints* de `C`.

w_1	w_2	**P**	**CE1**	**EC1**	**1-1 (%)**	**CEN**	**#CEN**	**ECN**	**#ECN**	**Extra**
50	–	23	58	74	132 (35,8)	19	48	7	21	459
125	–	28	52	73	125 (33,9)	17	45	11	27	420
250	–	27	46	68	114 (30,9)	12	28	19	46	424
500	–	22	57	76	133 (36,0)	13	32	8	24	430
1.000	–	29	44	69	113 (30,6)	13	29	21	50	399
1.500	–	28	46	74	120 (32,5)	9	21	22	56	362
2.000	–	28	57	74	131 (35,5)	13	37	11	26	403
2.500	–	37	39	72	111 (30,1)	10	24	26	57	331
500	62	21	52	87	139 (37,7)	9	35	12	30	392
500	125	15	40	97	137 (37,1)	4	15	17	43	366
500	250	21	56	82	138 (37,4)	7	28	10	27	389
500	375	19	55	80	135 (36,6)	8	27	12	30	360
500	500	11	43	90	133 (36,0)	6	16	13	36	368
500	625	14	45	87	132 (35,8)	10	32	15	41	380
500	750	21	49	89	138 (37,4)	10	31	12	30	390

Tabela 6.16: Relações entre *breakpoints* refinados identificados pelo `Cassis` e os *breakpoints* identificados com o processo de encadeamento de *hits* de regiões identificadas com o algoritmo de janela deslizante. Para facilitar a descrição, o conjunto de *breakpoints* do `Cassis` será denominado `C` e o conjunto de *breakpoints* produzido segundo nossa metodologia será denominado `E`. A coluna **P** indica o número de *breakpoints* de `C` que não foram detectados em `E`. A coluna **CE1** mostra o número de *breakpoints* de `C` que contém em seu interior um único *breakpoint* de `E`. A coluna **EC1** indica o número de *breakpoints* de `E` que contém em seu interior um único *breakpoint* de `C`. A coluna **1-1** representa a soma dos valores das colunas **CE1** e **EC1** e o valor dentro dos parênteses indica a porcentagem dos *breakpoints* do conjunto `C` (369 *breakpoints*) que são cobertos por esta soma. A coluna **CEN** exibe o número de *breakpoints* de `C` que contém mais de um *breakpoint* de `E` em seu interior e a coluna **#CEN** indica qual a quantidade de pontos de quebra de `E` afetados. Analogamente, a coluna **ECN** exibe o número de *breakpoints* de `E` que contém mais de um *breakpoint* de `C` em seu interior e a coluna **#ECN** indica qual a quantidade de pontos de quebra de `C` afetados. Finalmente, a coluna **Extra** indica o número de *breakpoints* do conjunto `E` que não possuem qualquer "contato" com *breakpoints* de `C`.

w_1	w_2	**P**	**CE1**	**EC1**	**1-1 (%)**	**CEN**	**#CEN**	**ECN**	**#ECN**	**PFP**
50	–	33	10	203	213 (57,7)	2	4	12	33	517
125	–	40	20	173	193 (52,3)	1	2	21	49	493
250	–	33	19	167	186 (50,4)	1	2	26	62	480
500	–	37	15	190	205 (55,6)	0	0	14	37	494
1.000	–	37	20	162	182 (49,3)	1	2	31	75	451
1.500	–	34	21	160	181 (49,1)	1	2	32	78	409
2.000	–	39	17	186	203 (55,0)	1	2	15	34	464
2.500	–	41	24	152	176 (47,7)	1	2	38	84	374
500	62	29	8	220	228 (61,8)	0	0	17	41	440
500	125	25	2	199	201 (54,5)	0	0	35	81	408
500	250	24	10	214	224 (60,7)	0	0	15	39	433
500	375	22	7	204	211 (57,2)	0	0	20	46	410
500	500	19	6	211	217 (58,8)	0	0	23	58	408
500	625	24	6	213	219 (59,3)	0	0	20	51	433
500	750	28	13	217	230 (62,3)	0	0	18	43	443

conjuntos que utilizam apenas uma janela deslizante e entre 11 e 21 para os conjuntos que utilizam duas janelas. Utilizando as coordenadas dos *breakpoints* refinados (Tabela 6.16), podemos ver que estes números aumentam tanto para os conjuntos que utilizaram uma janela como para os conjuntos que utilizaram duas janelas. Contudo, os conjuntos processados por apenas uma janela apresentam um aumento levemente maior: os valores estão entre 33 e 41 para os conjuntos que usam uma janela e entre 19 e 29 para os conjuntos que usam duas janelas. Assim, em relação ao parâmetro que mede as perdas de *breakpoints*, os algoritmos que utilizaram duas janelas possuem uma pequena vantagem.

Como mencionado anteriormente, os valores **CE1** e **EC1** tentam identificar situações em que ambos os métodos de detecção de *breakpoints* apontam para uma mesma região. Neste caso vale lembrar que o método `Cassis` mostra uma melhor resolução, sendo capaz de identificar *breakpoints* muito mais estreitos (principalmente após o refinamento). Assim, os números apresentados por **CE1** devem ser menores do que os apresentados por **EC1** devido a uma maior probabilidade de um *breakpoint* do conjunto `E` cobrir um *breakpoint* do conjunto `C`.

As Tabelas 6.15 e 6.16 mostram que em relação ao parâmetro **CE1** os conjuntos que utilizam apenas uma janela possuem resultado levemente melhores do que os conjuntos que adotam duas janelas. Em relação ao parâmetro **EC1** o contrário é observado: os conjuntos que utilizam duas janelas apresentam vantagem sobre o os conjuntos que utilizam apenas uma janela.

Se considerarmos a soma das colunas **CE1** e **EC1** (coluna **1-1**), podemos ver que, quando observamos os *breakpoints* não refinados do conjunto `C`, entre 30,1% e 37,7% dos 369 pontos de quebra do conjunto `C` possuem um único correspondente no conjunto `E` (conforme a configuração de w_1 e w_2). Este número sobe para valores entre 47,7% e 62,3% quando consideramos as coordenadas dos *breakpoints* refinados.

Aqui, devemos lembrar que apenas contamos casos em que os *breakpoints* estão completamente contidos em um *breakpoint* do outro conjunto. Assim, casos em que os *breakpoints* apresentam sobreposição apenas nas extremidades não foram computados. O aumento dos valores da coluna **1-1** (**CE1** + **EC1**) apresentados pela Tabela 6.16 em relação aos valores da Tabela 6.15 é uma consequência direta dos refinamento das coordenadas dos pontos de quebra produzidos pelo `Cassis`. Uma vez que as coordenadas dos *breakpoints* do conjunto `C` foram definidas com mais resolução, muitos dos casos de sobreposições nas extremidades foram convertidos para situações onde um *breakpoint* do conjunto `E` cobre inteiramente um *breakpoint* do con-

junto `C`. A observação deste comportamento indica que, de fato, a nossa metodologia foi capaz de encontrar uma porcentagem razoável dos *breakpoints* do `Cassis`: apesar de possuírem menor resolução e possuírem coordenadas deslocadas em relação aos *breakpoints* identificados no conjunto `C`, os pontos de quebra do conjunto `E` apontam para regiões similares no genoma.

Os valores **CEN** e **ECN** indicam a ocorrência de casos onde um ponto de quebra de um conjunto cobre mais de um *breakpoint* do outro. Isso pode indicar que um método não foi capaz de identificar rearranjos internos dentro dos *breakpoints* que ele definiu. Novamente, devido a melhor resolução apresentada pelo método `Cassis`, os valores da coluna **CEN** devem ser menores do que na coluna **ECN**. Isto é observado tanto na Tabela 6.15 como na Tabela 6.16. A Tabela 6.16, em especial, nos mostra que após o refinamento, praticamente não existem *breakpoints* deste conjunto que cobrem mais de um *breakpoint* do conjunto produzido por nossos métodos. Em relação à coluna **ECN**, a mesma observação não pode ser feita. Uma quantidade significativa de *breakpoints* do conjunto `C` estão contidas dentro de um *breakpoint* maior identificado pelo conjunto `E` (coluna (#ECN)).

Finalmente a coluna **Extra** aponta para *breakpoints* que foram definidos por nossa metodologia mas que não encontraram correspondência nos *breakpoints* produzidos pelo `Cassis`. Os números apresentados por esta coluna são altos, na maioria das vezes superior aos 369 *breakpoints* identificados pelo `Cassis` (especialmente na Tabela 6.16 que considera os *breakpoints* refinados na comparação). Os números desta coluna mostram uma pequena vantagem para os conjuntos produzidos com o uso de duas janelas deslizantes.

Considerando que `Cassis` usou informação de ortologia para definir seus blocos, acreditamos que a maioria dos *breakpoints* listados no campo `Extra` são na realidade falsos positivos. Contudo, uma análise mais detalhada das regiões no entorno de cada um dos *breakpoints* precisa ser feita para confirmar esta suspeita.

6.4 Conclusão

Neste capítulo propomos uma nova metodologia de identificação de *breakpoints* baseada apenas na informação de similaridade encontrada entre as sequências cromossômicas de dois genomas que estão sendo comparados.

A metodologia é bem simples e possui duas fases: análise dos *hits* dos alinhamen-

tos com um algoritmo de janela deslizante para identificação de regiões de interesse e encadeamento dos *hits* das regiões identificadas no primeiro passo para definição dos *breakpoints*.

Os resultados obtidos em nossos testes mostraram uma leve vantagem para o método que utiliza duas janelas deslizantes. Essa pequena vantagem pode ser explicada pela maior quantidade de informação que esta abordagem utiliza em relação àquela que utiliza apenas uma janela.

Enquanto o método de janela deslizante que utiliza apenas uma janela processa apenas os *hits* de regiões que apresentaram cobertura abaixo da média global do cromossomo, o método de duas janelas utiliza sub-regiões, identificadas pela segunda janela, provenientes tanto das regiões com baixa cobertura como das regiões com alta cobertura, definidas com a primeira janela.

Conforme a configuração utilizada, o nosso método foi capaz de encontrar um *breakpoint* equivalente (*breakpoint* de um conjunto que contém um único *breakpoint* inteiro do outro conjunto em seu interior) para uma porcentagem razoável dos *breakpoints* do produzidos pelo `Cassis`: entre 47,7% e 62,3% se considerarmos as coordenadas dos pontos de quebra refinados. Porém, o nosso método não é capaz de produzir *breakpoints* com resolução tão alta quanto o do método `Cassis`.

Uma maior desvantagem do nosso método está no número de *breakpoints* produzidos que não possuem qualquer correspondência com os pontos de quebra identificados pelo `Cassis`. Uma análise mais detalhada para avaliar a características destes pontos de quebra deve ser feita. Apesar de existir a possibilidade de alguns destes pontos de quebra serem rearranjos verdadeiros, acreditamos que a maioria deles podem ser falsos positivos.

A metodologia desenvolvida ao longo deste capítulo mostra potencial para a detecção de pontos de quebra. Apesar dos resultados estarem distantes dos resultados obtidos pelo `Cassis`, eles poderão ser melhorados através da evolução dos mecanismos criados tanto para a fase de seleção de regiões de interesse como na fase de encadeamento de *hits*.

Por exemplo, ao invés de considerar apenas as regiões de baixa cobertura, poderíamos considerar apenas as regiões de alta cobertura ou mesmo considerar as duas regiões na análise. Em relação ao encadeamento, alternativas mais robustas para detectar a ocorrência de um *breakpoint* devem ser consideradas.

Uma vantagem de nossa metodologia é que ela é independente de informação de ortologia e, dessa maneira, ela é adequada para análise de genomas que ainda não

dispõem deste tipo de informação.

Acreditamos que os resultados produzidos até aqui apontam para um embrião de uma nova metodologia que poderá ser desenvolvida e refinada para melhora dos resultados obtidos.

Considerações finais

Durante o trabalho desenvolvido nesta tese abordamos dois problemas distintos ligados ao estudo de rearranjo de genomas. Assim, o texto da tese foi dividido em duas partes.

O primeiro problema, abordado na Parte I da tese, foi a Enumeração de *Traces* de soluções para o problema de ordenação de permutações por reversões orientadas. A partir da análise das características das soluções propostas por Braga *et al.* [29–31] produzimos dois resultados principais.

O primeiro resultado foi apresentado no Capítulo 2 e refere-se a adoção de uma nova estrutura de dados para o processo de enumeração de *traces*. Mudando a abordagem de análise da árvore de *traces*, através da troca da busca em largura pela busca em profundidade, conseguimos produzir um algoritmo que utiliza uma simples estrutura de pilha e é capaz de processar grandes permutações utilizando uma quantidade de memória principal muito menor do que a exibida pelo algoritmo que utiliza a estrutura original desenvolvida por Braga.

Além de possuir um complexidade de tempo melhor que a do algoritmo original, como foi provado por Badr, Swenson e Sankoff [7], nosso algoritmo não necessita da realização de acessos a discos como no caso da estrutura original. Porém, ele apresenta a desvantagem de não poder ser aplicado em conjunto com restrições biológicas que não produzem *traces* perfeitos.

Badr, Swenson e Sankoff produziram recentemente uma nova abordagem para a enumeração de *traces* a partir de uma adaptação no nosso algoritmo e no algoritmo proposto por Braga [7]. Durante a enumeração, eles realizam o agrupamento de conjuntos de *i-traces* de acordo com a permutação a que eles estão associados no nível i da árvore. Isto evita a necessidade de se explorar mais de uma vez a sub-árvore de um *i-trace* relacionado a uma dada permutação do nível i. Os resultados obtidos por Badr, Swenson e Sankoff mostram uma redução de 50% no tempo de execução

do nosso algoritmo. Contudo, uma desvantagem da solução proposta por eles é que, para grandes permutações, uma grande quantidade de memória será necessária para manutenção do conjunto de permutações do nível i e de seus i-*traces* associados.

O estudo realizado durante a criação da estrutura foi apresentado oralmente e publicado nos anais do 25^{th} *Symposium on Applied Computing* (`ACM SAC 2010`) sob o título "*An Improved Algorithm to Enumerate All Traces that Sort a Signed Permutation by Reversals*" [14].

O segundo resultado do nosso estudo foi apresentado no Capítulo 3 e refere-se a uma nova abordagem para o problema de enumeração de *traces*. Ao invés de tentarmos enumerar todos os *traces*, procuramos produzir técnicas que permitam a enumeração parcial de *traces*. Esta abordagem foi criada com o objetivo de oferecer aos biólogos um conjunto de soluções alternativas para grandes permutações que não podem ser processadas pelo algoritmo de enumeração total. Três algoritmos para a enumeração parcial de *traces* que executam durante um limite de tempo pré-estabelecido foram propostos: algoritmo que utiliza a estrutura de pilha limitado por tempo (`Stack`), algoritmo de geração de *traces* aleatórios (`Random`) e algoritmo que explora "caminhos" de permutações utilizando janelas deslizantes de tamanho w (`Window w`).

A proposição destes algoritmos é a fase inicial do estudo para produção de enumeração parcial de *traces*. Apenas enumerar *traces* não é a solução ideal para os biólogos que precisam, na verdade, de pistas de soluções que sejam interessantes do ponto de vista biológico. Assim, como um trabalho futuro desta estratégia, prevemos a criação de um sistema de classificação de *traces*. A partir de um conjunto de *traces* obtidos, os elementos deste conjunto seriam ordenados de acordo com critérios biológicos pré-definidos. Desta forma, tanto a enumeração parcial como a enumeração total de *traces* se tornariam ferramentas mais úteis aos biólogos que poderiam concentrar esforços em algumas soluções específicas que atendem a critérios definidos por eles.

Os resultados produzidos até o momento sobre o problema de enumeração parcial de *traces* foram apresentados em formato de pôster na 14^{th} *International Conference on Research in Computational Molecular Biology* (`RECOMB 2010`) sob o título "Partial enumeration of traces of solutions for the problem of sorting by signed reversals" [15].

Um estudo intermediário, realizado durante a análise das características do algoritmo proposto por Braga *et al.* também resultou em uma publicação. Realizamos um estudo de um cenário evolutivo envolvendo seis α-proteobactérias do gênero

Rickettsia. Através da análise dos *traces* e da árvore filogenética dos membros do grupo, conseguimos inferir a ordem cronológica de eventos de reversão que ocorreram durante a evolução destes membros sem a necessidade de análises extensivas das sequências, dos marcadores e dos blocos conservados existentes entre eles. Este estudo foi apresentado oralmente e publicado nos anais do simpósio *ACM International Symposium on Biocomputing 2010* (`ISB 2010`) sob o título "*Chronological order of reversal events on Rickettsia genus*" [13].

Na Parte II da tese, tratamos do problema de identificação de *breakpoints* a partir da análise comparativa de genomas.

Na primeira fase deste estudo (Capítulo 5), realizamos a implementação do método proposto por Lemaitre *et al*. para a realização da identificação de pontos de quebra com base na lista de pares de genes ortólogos existentes entre duas espécies [100,102]. O método é composto por uma fase de identificação de blocos de sintenia confiáveis com base na lista de genes ortólogos. A partir destes blocos, os *breakpoints* são identificados e podem ser refinados na segunda fase do método. O refinamento é feito através da segmentação de uma curva calculada com base nos *hits* de alinhamentos entre três sequências de interesse: S_r proveniente do genoma de referência, S_{oA} proveniente do genoma G_o e ortóloga a porção inicial de S_r e S_{oB} também proveniente do genoma G_o, mas ortóloga a porção final de S_r. O método conta ainda com um procedimento estatístico para verificar se o refinamento produzido está realmente associado a uma sequência estruturada em três segmentos: segmento de S_r ortólogo a S_{oA}, segmento equivalente ao ponto de quebra e segmento de S_r ortólogo a S_{oB}.

Além de realizar a implementação dos métodos conforme proposto por Lemaitre, também adicionamos a opção da realização de refinamento de *breakpoints* identificados com outros métodos de detecção de blocos de sintenia. Para avaliar esta opção realizamos um estudo com blocos de sintenia disponíveis na plataforma `Ensembl` [85] (banco de dados `Compara`) e com blocos produzidos pelo programa `MAUVE` [49]. Nossos resultados mostram que os métodos implementados são capazes de refinar *breakpoints* definidos com outras metodologias.

Os métodos foram implementados em um pacote denominado `Cassis`. Este pacote, desenvolvido em `Perl` [46] e `R` [138], está disponível sob licensa `GNU GPL License` no endereço `http://pbil.univ-lyon1.fr/software/Cassis/`. Um *Application Notes* sobre este pacote foi publicado na revista *Bioinformatics* sob o título "Cassis: precise detection of genomic rearrangement breakpoints" [16]. Um pôster com mesmo título foi apresentado na 9^{th} *European Conference on Computational*

Biology (`ECCB 2010`) [17].

O Capítulo 6 apresentou uma nova metodologia para identificação de *breakpoints* baseada apenas na informação de similaridade entre regiões intergênicas de dois genomas que estão sendo comparados.

Esta metodologia foi desenvolvida com base em observações de que as regiões que estão dentro e no entorno dos *breakpoints* possuem um grau de similaridade, com suas sequências ortólogas, menor do que o observado em outras regiões do genoma. Isto pode ser um possível efeito dos micro-rearranjos e duplicações em *tandem* que costumam ser observados no interior dos pontos de quebra.

Em uma primeira fase, nossa metodologia realiza a busca por regiões de interesse: segmentos de sequência cromossômica que apresentem, localmente, uma média de cobertura de *hits* menor do que no seu entorno. Isto é feito através de um algoritmo que usa janela deslizante, em um ou dois níveis, para procurar por regiões que possuem cobertura abaixo da média observada em todo o cromossomo (primeiro nível) ou dentro de uma região (segundo nível) identificada na análise de primeiro nível.

A segunda fase é responsável pela identificação de *breakpoints* através do encadeamento de *hits* contidos no interior das regiões definidas na primeira etapa. As cadeias são produzidas e todo espaço entre duas cadeias consecutivas são marcados como pontos de quebras.

Realizamos testes para avaliar esta metodologia. Como os pontos de quebras obtidos pelo `Cassis` utilizam informação de ortologia, eles são mais confiáveis e, por isso, os utilizamos para a validar os resultados produzidos por nosso método.

Os primeiros testes mostraram que a metodologia tem potencial pois foi capaz de encontrar, conforme a configuração utilizada, até 62% dos *breakpoints* identificados pelo `Cassis`.

Contudo, alguns problemas foram observados durante os testes. A resolução é menor do que a oferecida pelo `Cassis` e, assim, os pontos de quebras possuem médias de tamanhos maiores. Além disso, o método produz um grande número de *breakpoints* que não possuem correspondência com os pontos de quebra obtidos com o `Cassis`. Apesar de existir a possibilidade de alguns destes *breakpoints* serem pontos de quebra verdadeiros, acreditamos que muito deles são falsos positivos e uma análise mais minuciosa é necessária para caracterizá-los.

Esta nova metodologia ainda requer um trabalho de refinamento. Apesar de ela não produzir resultados tão bons quanto os obtidos pelo `Cassis` ela tem a vantagem de não utilizar informações de ortologia. Dessa maneira, ela é adequada para estudo

de genomas que ainda não possuem este tipo de informação.

Durante o desenvolvimento desta tese, realizamos um amplo estudo de dois aspectos ligados a evolução. No primeiro aspecto, estudamos as inúmeras combinações que são possíveis para determinar uma sequência de reversões que ditaram a evolução de um grupo de espécies. No segundo aspecto, estudamos as regiões onde ocorrem as quebras que delimitam os blocos rearranjados. Ambos os tópicos ainda possuem muito a ser explorado e, com o estudo realizado nesta tese, esperamos ter produzido uma pequena contribuição para auxiliar o entendimento dos mecanismos envolvidos na evolução.

Referências Bibliográficas

[1] G. Achaz, F. Boyer, C. P. E. Rocha, A. Viari, and E. Coissac. Repseek, a tool to retrieve approximate repeats from large DNA sequences. *Bioinformatics*, 23(1):119–121, 2007.

[2] M. Alekseyev and P. A. Pevzner. Are there rearrangement hotspots in the human genome? *PLoS Computational Biology*, 3(11):e209, 2007.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.

[4] L. Andersson, A. Archibald, M. Ashburner, S. Audun, W. Barendse, J. Bitgood, C. Bottema, T. Broad, S. Brown, D. Burt, et al. Comparative genome organization of vertebrates. *Mammalian Genome*, 7(10):717–734, 1996.

[5] I Auger and C. Lawrence. Algorithms for the optimal identification of segments neighborhoods. *Bulletin of Mathematical Biology*, 51:39–54, 1989.

[6] D. A. Bader, B. M. E. Moret, and M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology*, 8(5):483–491, 2001.

[7] G. Badr, K. Swenson, and D. Sankoff. Listing all parsimonius reversal sequences: New algorithms and perspectives. In E. Tannier, editor, *Proceedings of the 8th Annual RECOMB Satellite Workshop on Comparative Genomics (RECOMB-CG 2010)*, volume 6398 of *Lecture Notes in Bioinformatics*, pages 39–49, Ottawa, Canada, October 2010. Springer-Verlag Berlin Heidelberg.

[8] V. Bafna and P. A. Pevzner. Sorting by reversals: Genome rearrangements in plant organelles and evolutionary history of X chromosome. *Molecular Biology and Evolution*, 12(2):239–246, 1995.

[9] V. Bafna and P. A. Pevzner. Sorting by transpositions. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 614–623, San Francisco, USA, January 1995.

[10] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.

[11] V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, May 1998.

[12] A. Bashir, S. Volik, C. Collins, V. Bafna, and B. J. Raphael. Evaluation of paired-end sequencing strategies for detection of genome rearrangements in cancer. *PLoS Computational Biology*, 4(4):e1000051, 2008.

[13] C. Baudet and Z. Dias. Chronological order of reversal events on *Rickettsia* genus. In *Proceedings of the ACM International Symposium on Biocomputing 2010 (ISB 2010)*, Calicut, Kerala, India, February 2010. 5 pages.

[14] C. Baudet and Z. Dias. An improved algorithm to enumerate all traces that sort a signed permutation by reversals. In *Proceedings of the 25th Symposium On Applied Computing (ACM SAC 2010)*, Sierre, Switzerland, March 2010. 5 pages, Bioinformatics Track.

[15] C. Baudet and Z. Dias. Partial enumeration of traces of solutions for the problem of sorting by signed reversals. In *Proceedings of the 14th International Conference on Research in Computational Molecular Biology (RECOMB 2010)*, page A11, Lisbon, Portugal, August 2010. Poster.

[16] C. Baudet, C. Lemaitre, Z. Dias, E. Tannier, C. Gautier, and M.-F. Sagot. Cassis: precise detection of genomic rearrangement breakpoints. *Bioinformatics*, 26(15):1897–1898, 2010. Application Notes.

[17] C. Baudet, C. Lemaitre, Z. Dias, E. Tannier, C. Gautier, and M.-F. Sagot. Cassis: precise detection of genomic rearrangement breakpoints. In *Proceedings*

of the 9th European Conference on Computational Biology (ECCB 2010), page B25, Ghent, Belgium, September 2010. Poster.

[18] A. Bergeron. A very elementary presentation of the Hannenhalli-Pevzner theory. *Discrete Applied Mathematics*, 146(2):134–145, 2005.

[19] A. Bergeron, C. Chauve, T. Hartman, and K. Saint-Onge. On the properties of sequences of reversals that sort a signed permutation. In *Proceedings of the JOBIM 2002*, pages 99–108, Saint Malo, 2002.

[20] A. Bergeron, S. Corteel, and M. Raffinot. The algorithmic of gene teams. In R. Guigó and D. Gusfield, editors, *Proceedings of the Workshop on Algorithms in Bioinformatics 2002 (WABI 2002)*, volume 2452 of *Lecture Notes in Computer Science*, pages 464–476, Rome, Italy, September 2002. Springer-Verlag Berlin Heidelberg.

[21] A. Bergeron, J. Mixtacki, and J. Stoye. On sorting by translocations. *Journal of Computational Biology*, 13(2):567–578, 2006.

[22] P. Berman and S. Hannenhalli. Fast sorting by reversal. In D. S. Hirschberg and E. W. Myers, editors, *Proceedings of Combinatorial Pattern Matching (CPM'96), 7th Annual Symposium*, volume 1075 of *Lecture Notes in Computer Science*, pages 168–185, Laguna Beach, USA, January 1996. Springer.

[23] P. Berman, S. Hannenhalli, and M. Karpinski. 1.375-approximation algorithm for sorting by reversals. In *Proceedings of the 10th European Symposium on Algorithms (ESA'2002)*, Lecture Notes in Computer Science, Rome, Italy, September 2002. Springer.

[24] G. Blanc, H. Ogata, C. Robert, S. Audic, K. Suhre, G. Vestris, J.-M. Claverie, and D. Raoult. Reductive genome evolution from the mother of *Rickettsia*. *PLos Genetics*, 3(1):103–114, January 2007.

[25] M. Blanchette, T. Kunisawa, and D. Sankoff. Parametric genome rearrangement. *Journal of Computational Biology*, 172:11–17, 1996.

[26] G. Bourque, E. M. Zdobnov, P. Bork, P. A. Pevzner, and G. Tesler. Comparative architectures of mammalian and chicken genomes reveal highly variable

rates of genomic rearrangements across different lineages. *Genome Research*, 15(1):98–110, 2005.

[27] M. D. V. Braga. *Exploring the Solution Space of Sorting by Reversals When Analyzing Genome Rearrangements*. PhD thesis, Université Lyon 1, France, 2008.

[28] M. D. V. Braga. `baobabLuna`: the solution space of sorting by reversals. *Bioinformatics*, 25(14):1833–1835, April 2009. Applications Notes.

[29] M. D. V. Braga, C. Gautier, and M.-F. Sagot. An asymmetric approach to preserve common intervals while sorting by reversals. *Algorithms for Molecular Biology*, 4(1):16, 2009.

[30] M. D. V. Braga, M.-F. Sagot, C. Scornavacca, and E. Tannier. The solution space of sorting by reversals. In *Proceedings of the International Symposium on Bioinformatics Research and Applications 2007 (ISBRA 2007)*, volume 4463 of *Bioinformatics Research and Applications*, pages 293–304. Springer Berlin / Heidelberg, 2007.

[31] M. D. V. Braga, M.-F. Sagot, C. Scornavacca, and E. Tannier. Exploring the solution space of sorting by reversals with experiments and an application to evolution. *Transactions on Computational Biology and Bioinformatics*, 5(3):348–356, 2008.

[32] N. Bray, I. Dubchak, and L. Pachter. AVID: A global alignment program. *Genome Research*, 13(1):97–102, 2003.

[33] M. Brudno, M. Chapman, B. Göttgens, S. Batzoglou, and B. Morgenstern. Fast and sensitive multiple alignment of large genomic sequences. *BMC Bioinformatics*, 4(66):11 pages, 2003.

[34] P. P. Calabrese, S. Chakravarty, and T. J. Vision. Fast identification and statistical evaluation of segmental homologies in comparative maps. *Bioinformatics*, 19(Suppl 1):i74–i80, 2003.

[35] S. B. Cannon, A. Kozik, B. Chan, R. Michelmore, and N. D. Young. DiagHunter and GenoPix2D: programs for genomic comparisons, large-scale homology discovery and visualization. *Genome Biology*, 4(10):R68, 2003.

[36] A. Caprara. Sorting permutations by reversals and eulerian cycle decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91–110, February 1999.

[37] A. Caprara and G. Lancia. Experimental and statistical analysis of sorting by reversals. In D. Sankoff and J. H. Nadeau, editors, *Comparative Genomics: Empirical and Analyitical Approaches to Gene Order Dynamics, Map Alignment and Evolution of Gene Families*. Kluwer Academic Publishers, Le Chantecler, Canada, September 2000.

[38] P. Cartier and D. Foata. *Problèmes combinatoires de commutation et réarrangements*. Number 85 in Lecture Notes in Mathematics. Springer-Verlag, Berlin, 1969.

[39] V. Choi, C. Zheng, Q. Zhu, and D. Sankoff. Algorithms for the extraction of synteny blocks from comparative maps. In R. Giancarlo and S. Hannenhalli, editors, *Proceedings of the Workshop on Algorithms in Bioinformatics 2007 (WABI 2007)*, volume 4645 of *Lecture Notes in Computer Science*, pages 277–288, Philadelphia, PA, USA, September 2007. Springer-Verlag Berlin Heidelberg.

[40] D. A. Christie. Sorting permutations by block-interchanges. *Information Processing Letters*, 60(4):165–169, November 1996.

[41] D. A. Christie. A 3/2-approximation algorithm for sorting by reversals. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 244–252, San Francisco, USA, January 1998.

[42] D. A. Christie. *Genome Rearrangement Problems*. PhD thesis, Glasgow University, 1998.

[43] A. J. Clutterbuck and S. J. O'Brien. *Genetics Maps: Locus Maps of Complex Genomes*. Cold Spring Harbor Laobratory Press, New York, 6 edition, 1993.

[44] A. Coghlan, E. E. Eichler, S. G. Oliver, A. H. Paterson, and L. Stein. Chromosome evolution in eukaryotes: a multi-kingdom perspective. *Trends in Genetics*, 21(12):673–682, 2005.

[45] O. Couronne, A. Poliakov, N. Bray, T. Ishkhanov, D. Ryaboy, E. Rubin, L. Pachter, and I. Dubchak. Strategies and tools for whole-genome alignments. *Genome Research*, 13(1):73–80, 2003.

[46] CPAN – Comprehensive Perl Archive Network, September 2002. http://www.cpan.org.

[47] V. Curwen, E. Eyras, T. D. Andrews, L. Clarke, E. Mongin, S. M. J. Searle, and M. Clamp. The Ensembl Automatic Gene Annotation System. *Genome Research*, 14:942–950, 2004.

[48] E. Darai-Ramqvist, A. Sandlund, S. Müller, G. Klein, S. Imreh, and M. Kost-Alimova. Segmental duplications and evolutionary plasticity at tumor chromosome break-prone regions. *Genome Research*, 18:370–379, 2008.

[49] A. C. E. Darling, B. Mau, F. R. Blattner, and N. T. Perna. Mauve: Alignment of conserved genomic sequence with rearrangements. *Genome Research*, 14:1394–1403, 2004.

[50] R. W. DeBry and M. F. Seldin. Human/mouse homology relationships. *Genomics*, 33(3):337–351, 1996.

[51] T. Derrien, C. André, F. Galibert, and C. Hitte. AutoGRAPH: an interactive web server for automating and visualizing comparative genome maps. *Bioinformatics*, 23(4):498–499, 2007.

[52] C. N. Dewey and L. Pachter. Evolution at the nucleotide level: the problem of multiple whole-genome alignment. *Human Molecular Genetics*, 15(Review issue 1):R51–R56, 2006.

[53] C. N. Dewey and L. Pachter. Mercator: Multiple whole-genome orthology map construction. webpage, 2006. http://bio.math.berkeley.edu/mercator/.

[54] Z. Dias and J. Meidanis. Genome rearrangements distance by fusion, fission, and transposition is easy. In *Proceedings of the String Processing and Information Retrieval (SPIRE'2001)*, pages 250–253, Laguna de San Rafael, Chile, November 2001. IEEE Computer Society.

[55] Y. Diekmann, M.-F. Sagot, and E. Tannier. Evolution under reversals: parsimony and conservation of common intervals. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(2):301–309, 2005.

[56] T. Dobzhansky and A. H. Sturtevant. Inversions in the third chromosome of wild races of *Drosophila pseudoobscura*, and their use in the study of the history of the species. *Proceedings of the National Academy of Science, USA*, 22:448–450, July 1936.

[57] T. Dobzhansky and A. H. Sturtevant. Inversions in the chromosomes of *Drosophila pseudoobscura*. *Genetics*, 23(1):28, 1938.

[58] J.-F. Dufayard, L. Duret, S. Penel, M. Gouy, F. Rechenmann, and G. Perrière. Tree pattern matching in phylogenetic trees: automatic search for orthologs or paralogs in homologous gene sequence databases. *Bioinformatics*, 21(11):2596–2603, 2005.

[59] B. Dutrillaux. Comment évoluent les chromosomes de mammifères. *La Recherche*, 296:70–75, 1997.

[60] R. C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 2004.

[61] J. Ehrlich, D. Sankoff, and J. H. Nadeau. Synteny conservation and chromosome rearrangements during mammalian evolution. *Genetics*, 147(1):289–296, 1997.

[62] J. A. Eisen, J. F. Heidelberg, O. White, and S. L. Salzberg. Evidence for symmetric chromosomal inversions around the replication origin in bacteria. *Genome Biology*, 1(6):11.1–11.9, 2000.

[63] N. El-Mabrouk and D. Sankoff. Genome rearrangement. In T. Jiang, T. Smith, Y. Xu, and M. Zhang, editors, *Current Topics in Computational Biology*. MIT Press, Cambridge, 2001.

[64] I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. In *Proceedings of 5th Workshop on Algorithms in Bioinformatics (WABI'05)*, volume 3692 of *Lecture Notes in Bioinformatics*, pages 204–215, 2005.

[65] N. Eriksen. $(1 + \epsilon)$-approximation of sorting by reversals and transpositions. *Theoretical Computer Science*, 289(1):517–529, October 2002.

[66] M. A. Ferguson-Smith and V. Trifonov. Mammalian karyotype evolution. *Nature Reviews Genetics*, 8:950–962, 2007.

[67] P. A. Fujita, B. Rhead, A. S. Zweig, A. S. Hinrichs, D. Karolchik, M. S. Cline, M. Goldman, G. P. Barber, H. Clawson, A. Coelho, M. Diekhans, T. R. Dreszer, B. M. Giardine, R. A. Harte, J. Hillman-Jackson, F. Hsu, V. Kirkup, R. M. Kuhn, K. Learned, C. H. Li, L. R. Meyer, A. Pohl, B. J. Raney, K. R. Rosenbloom, K. E. Smith, D. Haussler, and W. J. Kent. The UCSC Genome Browser database: update 2011. *Nucleic Acids Research*, pages 1–7, October 2010. First published online (doi:10.1093/nar/gkq963).

[68] L. Gordon, S. Yang, M. Tran-Gyamfi, D. Baggott, M. Christensen, A. Hamilton, R. Crooijmans, M. Groenen, S. Lucas, I. Ovcharenko, et al. Comparative analysis of chicken chromosome 28 provides new clues to the evolutionary fragility of gene-rich vertebrate regions. *Genome Research*, 17:1603–1613, 2007.

[69] Q.-P. Gu and K. Iwata. A heuristic algorithm for genome rearrangements, December 1997. Poster in The Eighth Workshop on Genome Informatics (GIW'97).

[70] Q.-P. Gu, S. Peng, and H. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theoretical Computer Science*, 210(2):327–339, January 1999.

[71] B. J. Haas, A. L. Delcher, J. R. Wortman, and S. L. Salzberg. DAGchainer: a tool for mining segmental genome duplications and synteny. *Bioinformatics*, 20(18):3643–3646, 2004.

[72] S. Hampson, A. McLysaght, B. Gaut, and P. Baldi. LineUp: statistical detection of chromosomal homology with application to plant comparative genomics. *Genome Research*, 13(5):999–1010, 2003.

[73] S. Hannenhalli. Polynomial-time algorithm for computing translocation distance between genomes. *Discrete Applied Mathematics*, 71:137–151, 1996.

[74] S. Hannenhalli, C. Chappey, E. V. Koonin, and P. A. Pevzner. Genome sequence comparison and scenarios for gene rearrangements: a test case. *Genomics*, 30:299–311, 1995.

[75] S. Hannenhalli and P. A. Pevzner. Towards a computational theory of genome rearrangements. *Lecture Notes in Computer Science*, 1000:184–200, 1995.

[76] S. Hannenhalli and P. A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 581–592, Los Alamitos, USA, October 1995. IEEE Computer Society Press.

[77] S. Hannenhalli and P. A. Pevzner. To cut... or not to cut (applications of comparative physical maps in molecular evolution). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 304–313, Atlanta, USA, January 1996.

[78] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 46(1):1–27, January 1999.

[79] R. S. Harris. *Improved pairwise alignment of genomic DNA*. PhD thesis, The Pennsylvania State Univeristy, 2007. 84 pages.

[80] L. S. Heath and J. P.Vergara. Sorting by bounded block-moves. *Discrete Applied Mathematics, Second Special Issue on Computational Biology*, 88:181–206, 1998.

[81] L. S. Heath and J. P. Vergara. Sorting by short block-moves. *Algorithmica*, June 2000. Online publication.

[82] S. Heber and J. Stoye. Finding all common intervals of k permutations. In *Proceeding of the 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 207–218, 2001.

[83] H. Hinsch and S. Hannenhalli. Recurring genomic breaks in independent lineages support genomic fragility. *BMC Evolutionary Biology*, 6:90, 2006.

[84] R. Hoberman, D. Sankoff, and D. Durand. The statistical analysis of spatially clustered genes under the maximum gap criterion. *Journal of Computational Biology*, 12(8):1083–1102, 2005.

[85] T. J. P. Hubbard, B. L. Aken, S. Ayling, B. Ballester, K. Beal, E. Bragin, S. Brent, Y. Chen, P. Clapham, L. Clarke, G. Coates, S. Fairley, S. Fitzgerald, J. Fernandez-Banet, L. Gordon, S. Graf, S. Haider, M. Hammond, R. Holland, K. Howe, A. Jenkinson, N. Johnson, A. Kahari, D. Keefe, S. Keenan, R. Kinsella, F. Kokocinski, E. Kulesha, D. Lawson, I. Longden, K. Megy, P. Meidl, B. Overduin, A. Parker, B. Pritchard, D. Rios, M. Schuster, G. Slate, D. Smedley, W. Spooner, G. Spudich, S. Trevanion, A. Vilella, J. Vogel, S. White, S. Wilder, A. Zadissa, E. Birney, F. Cunningham, V. Curwen, R. Durbin, X. M. Fernandez-Suarez, J. Herrero, A. Kasprzyk, G. Proctor, J. Smith, S. Searle, and P. Flicek. Ensembl 2009. *Nucleic Acids Research*, 37(Suppl 1):D690–D697, 2009.

[86] T. J. P. Hubbard, B. L. Aken, K. Beal, B. Ballester, M. Caccamo, Y. Chen, L. Clarke, G. Coates, F. Cunningham, T. Cutts, et al. Ensembl 2007. *Nucleic Acids Research*, 35(Database issue):D610–D617, 2007.

[87] M. Jerrum. The complexity of finding minimum-length generator sequences. *Theoretical Computer Science*, 36:265–289, 1985.

[88] H. Kaplan, R. Shamir, and R. E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal on Computing*, 29(3):880–892, January 2000.

[89] H. Kaplan and E. Verbin. Efficient data structures and a new randomized approach for sorting signed permutations by reversals. In *Proceeding of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *Lecture Notes in Computer Science*, pages 170–185, Michoacán, Mexico, June 2003. Springer-Verlag.

[90] J. D. Kececioglu and R. Ravi. Of mice and men: Algorithms for evolutionary distances between genomes with translocation. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms*, pages 604–613, New York, USA, January 1995. ACM Press.

[91] J. D. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13:180–210, January 1995.

[92] H. Kehrer-Sawatzki and D. N. Cooper. Molecular mechanisms of chromosomal rearrangement during primate evolution. *Chromosome Research*, 16(1):41–56, 2008.

[93] J. W. Kent. BLAT – the BLAST-like alignment tool. *Genome Research*, 12(4):656–664, 2002.

[94] W. J. Kent, R. Baertsch, A. Hinrichs, W. Miller, and D. Haussler. Evolution's cauldron: duplication, deletion, and rearrangement in the mouse and human genomes. *Proceedings of the National Academy of Sciences, USA*, 100(20):11484–11489, 2003.

[95] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at UCSC. *Genome Research*, 12(6):996–1006, 2002.

[96] L. Klasson, T. Walker, M. Sebaihia, M. J. Sanders, M. A. Quail, A. Lord, S. Sanders, J. Earl, S. L. O'Neill, N. Thomson, S. P. Sinkins, and J. Parkhill. Genome evolution of *Wolbachia* strain wpip from the *Culex pipiens* group. *Molecular Biology and Evolution*, 25(9):1877–1887, September 2008.

[97] J. O. Korbel, A. E. Urban, J. P. Affourtit, B. Godwin, F. Grubert, J. F. Simons, P. M. Kim, D. Palejev, N. J. Carriero, L. Du, et al. Paired-end mapping reveals extensive structural variation in the human genome. *Science*, 318(5849):420–426, 2007.

[98] B. T. Lahn and D. C. Page. Four evolutionary strata on the human X chromosome. *Science*, 286:964–967, 1999.

[99] C. Ledergerber and C. Dessimoz. Alignments with non-overlapping moves, inversions and tandem duplications in $O(n^4)$ time. *Journal of Combinatorial Optimization*, 16(3):263–278, 2007.

[100] C. Lemaitre. *Réarrangements chromosomiques dans les génomes de mammifères: caractérisation des points de cassure*. PhD thesis, Université Claude Bernard – Lyon 1, Novembre 2008. 180 pages, In French.

[101] C. Lemaitre and M.-F. Sagot. A small trip in the untranquil world of genomes: A survey on the detection and analysis of genome rearrangement breakpoints. *Theoretical Computer Science*, 395(2-3):171–192, 2008.

[102] C. Lemaitre, E. Tannier, C. Gautier, and M.-F. Sagot. Precise detection of rearrangement breakpoints in mammalian chromosomes. *BMC Bioinformatics*, 9:286, June 2008. 15 pages.

[103] G. Li, X. Qi, X. Wang, and B. Zhu. *Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, chapter A Linear-Time Algorithm for Computing Translocation Distance between Signed Genomes, pages 323–332. Springer Berlin / Heidelberg, 2004.

[104] H. Li et al. TreeSoft: Softwares for Phylogenetic Trees, November 2010. http://treesoft.sourceforge.net/index.shtml.

[105] L. Li, C. J. Stoeckert, and D. S. Roos. OrthoMCL: Identification of ortholog groups for eukaryotic genomes. *Genome Research*, 13:2178–2189, 2003.

[106] Z. Li and L. Wang. Algorithmic approaches for genome rearrangement: A review. *IEEE Transactions on Systems, Man, and Cybernetics*, 36(5):636–648, September 2006.

[107] G.-H. Lin and G. Xue. Signed genome rearrangement by reversals and transpositions: Models and approximations. In *Fifth Annual International Computing and Combinatorics Conference (COCOON'99)*, pages 71–80, 1999.

[108] B. Ma, J. Tromp, and M. Li. Pattern hunter: faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.

[109] J. Ma, L. Zhang, B. B. Suh, B. J. Raney, R. C. Burhans, W. J. Kent, M. Blanchette, D. Haussler, and W. Miller. Reconstructing contiguous regions of an ancestral genome. *Genome Research*, 16:1557–1565, 2006.

[110] P. Mackiewicz, D. Mackiewicz, M. Kowalczuk, and S. Cebrat. Flip-flop around the origin and terminus of replication in prokariotic genomes. *Genome Biology*, 2(12):1004.1–1004.4, 2001.

[111] J. Meidanis, M. E. M. T. Walter, and Z. Dias. A lower bound on the reversal and transposition diameter. Technical Report IC-00-16, Institute of Computing - University of Campinas, October 2000.

[112] W. J. Murphy, D. M. Larkin, A. E. van der Wind, G. Bourque, G. Tesler, L. Auvil, J. E. Beever, B. P. Chowdhary, F. Galibert, L. Gatzke, et al. Dynamics of mammalian chromosome evolution inferred from multispecies comparative maps. *Science*, 309(5734):613–617, 2005.

[113] J. H. Nadeau and D. Sankoff. The lengths of undiscovered conserved segments in comparative maps. *Mammalian Genome*, 9(6):491–495, 1998.

[114] J. H. Nadeau and B. A. Taylor. Lengths of chromosomal segments conserved since divergence of man and mouse. *Proceedings of the National Academy Science, USA*, 81:814–818, 1984.

[115] V. Navratil. GeM (Genomic Mapping) Website. webpage, 2005. http://pbil.univ-lyon1.fr/gem/gem_home.php.

[116] V. Navratil. *Modélisation des connaissances en génomique par une approche de cartographie comparée: Application à la détection et à l'analyse des SNPs exoniques chez les vertébrés*. PhD thesis, Université Claude Bernard – Lyon 1, 2005.

[117] T. L. Newman, E. Tuzun, V. A. Morrison, K. E. Hayden, M. Ventura, S. D. McGrath, M. Rocchi, and E. E. Eichler. A genome-wide survey of structural variation between human and chimpanzee. *Genome Research*, 15(10):1344–1356, 2005.

[118] R. A. Notebaart, M. A. Huynen, B. Teusink, R. J. Siezen, and B. Snel. Correlation between sequence conservation and the genomic context after gene duplication. *Nucleic Acids Research*, 33(19):6164–6171, 2005.

[119] K. P. O'Brien, M. Remm, and E. L. L. Sonnhammer. Inparanoid: a comprehensive database of eukaryotic orthologs. *Nucleic Acids Research*, 33(Database issue):D476–D480, 2005.

[120] S. Ohno. *Sex chromosomes and sex-linked genes*. Springer, Berlin, 1967.

[121] R. D. Page and M. A. Charleston. From gene to organismal phylogeny: reconciled trees and the gene tree/species tree problem. *Molecular Phylogenetics and Evolution*, 7(2):231–240, 1997.

[122] J. D. Palmer and L. A. Herbon. Plant mitochondrial DNA evolves rapidly in structure, but slowly in sequence. *Journal of Molecular Evolution*, 27:87–97, 1988.

[123] X. Pan, L. Stein, and V. Brendel. SynBrowse: a synteny browser for comparative sequence analysis. *Bioinformatics*, 21(17):3461–3468, 2005.

[124] E. Passarge, B. Horsthemke, and R. A. Farber. Incorrect use of the term synteny. *Nature Genetics*, 23(4):387, 1999.

[125] G. Pavesi, G. Mauri, F. Iannelli, C. Gissi, and G. Pesole. GeneSyn: a tool for detecting conserved gene order across genomes. *Bioinformatics*, 20(9):1472–1474, 2004.

[126] Q. Peng, P. A. Pevzner, and G. Tesler. The fragile breakage versus random breakage models of chromosome evolution. *PLoS Computational Biology*, 2(2):e14, 2006.

[127] P. Peterlongo. *Filtrage de séquences d'ADN pour la recherche de longues répétitions multiples*. PhD thesis, Université de Marne-la-Vallée, 2006. In French.

[128] P. Peterlongo, N. Pisanti, F. Boyer, A. Pereira do Lago, and M.-F. Sagot. Lossless filter for multiple repetitions with hamming distance. *Journal of Discrete Algorithms*, 6(3):497–509, 2008.

[129] P. Peterlongo, G. A. T. Sacomoto, A. Pereira do Lago, N. Pisanti, and M.-F. Sagot. Lossless filter for multiple repeats with bounded edit distance. *Algorithms for Molecular Biology*, 4(3):20 pages, 2009.

[130] P. A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000.

[131] P. A. Pevzner and G. Tesler. Genome rearrangements in mammalian evolution: lessons from human and mouse genomes. *Genome Research*, 13:37–45, 2003.

[132] P. A. Pevzner and G. Tesler. Human and mouse genomic sequences reveal extensive breakpoint reuse in mammalian evolution. *Proceedings of the National Academy of Sciences, USA*, 100(13):7672–7677, 2003.

[133] S. C. Potter, L. Clarke, V. Curwen, S. Keenan, E. Mongin, S. M. J. Searle, A. Stabenau, R. Storey, and M. Clamp. The Ensembl Analysis Pipeline. *Genome Research*, 14:934–941, 2004.

[134] K. D. Pruitt, J. Harrow, R. A. Harte, C. Wallin, M. Diekhans, D. R. Maglott, S. Searle, C. M. Farrell, J. E. Loveland, B. J. Ruef, E. Hart, M. M. Suner, M. J. Landrum, B. Aken, S. Ayling, R. Baertsch, J. Fernandez-Banet, J. L. Cherry, V. Curwen, M. Dicuccio, M. Kellis, J. Lee, M. F. Lin, M. Schuster, A. Shkeda, C. Amid, G. Brown, O. Dukhanina, A. Frankish, J. Hart, B. L. Maidak, J. Mudge, M. R. Murphy, T. Murphy, J. Rajan, B. Rajput, L. D. Riddick, C. Snow, C. Steward, D. Webb, J. A. Weber, L. Wilming, W. Wu, E. Birney, D. Haussler, T. Hubbard, J. Ostell, R. Durbin, and D. Lipman. The consensus coding sequence (CCDS) project: Identifying a common protein-coding gene set for the human and mouse genomes. *Genome Research*, 19(7):1316–1323, 2009.

[135] D. Rasko, G. Myers, and J. Ravel. Visualization of comparative genomic analyses by BLAST score ratio. *BMC Bioinformatics*, 6(1):2, 2005.

[136] RicBase. Rickettsia genome database, March 2009. http://www.igs.cnrs-mrs.fr/mgdb/Rickettsia/.

[137] M. T. Ross et al. The DNA sequence of the human X chromosome. *Nature*, 434:325–337, 2005.

[138] The R project for statistical computing, October 2010. http://www.r-project.org.

[139] D. Sankoff. Edit distance for genome comparison based on non-local operations. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings of the Third Annual Symposium of the Combinatorial Pattern Matching (CPM'92)*, number 664 in Lecture Notes in Computer Science, pages 121–135, Tucson, USA, 1992. Springer-Verlag.

[140] D. Sankoff. The signal in the genomes. *PLoS Computational Biology*, 2(4):e35, 2006.

[141] D. Sankoff, G. Leduc, N. Antoine, B. Paquin, B. F. Lang, and R. Cedergren. Gene order comparisons for phylogenetic inference: Evolution of the mitochondrial genome. *Proceedings of the National Academy Science, USA*, 89:6575–6579, 1992.

[142] D. Sankoff and J. H. Nadeau. Chromosome rearrangements in evolution: From gene order to genome sequence and back. *Proceedings of the National Academy of Sciences, USA*, 100(20):11188–11189, 2003.

[143] D. Sankoff and P. Trinh. Chromosomal breakpoint reuse in genome sequence rearrangement. *Journal of Computational Biology*, 12(6):812–821, 2005.

[144] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller. Human-mouse alignments with BLASTZ. *Genome Research*, 13(1):103–107, 2003.

[145] A. C. Siepel. An algorithm to enumerate sorting reversals. *Journal of Computational Biology*, 10(3–4):575–597, 2003.

[146] A. U. Sinha and J. Meller. Cinteny: flexible analysis and visualization of synteny and genome rearrangements in multiple organisms. *BMC Bioinformatics*, 8:82, 2007.

[147] H. Skaletsky et al. The male-specific region of the human Y chromosome is a mosaic of discrete sequence classes. *Nature*, 423:825–837, 2003.

[148] D. Smedley, S. Haider, B. Ballester, R. Holland, D. London, G. Thorisson, and A. Kasprzyk. BioMart - biological queries made easy. *BMC Genomics*, 10(1):22 pages, 2009.

[149] A. F. A. Smit, R. Hubley, and P. Green. Repeatmasker, 2009. http://www.repeatmasker.org.

[150] Y. Sun and J. Buhler. Choosing the best heuristic for seeded alignment of DNA sequences. *BMC Bioinformatics*, 7(1):133, 2006.

[151] K. M. Swenson, G. Badr, and D. Sankoff. Listing all sorting reversals in quadratic time. In V. Moulton and M. Singh, editors, *Proceedings of the 10th Workshop in Algorithms in Bioinformatics (WABI 2010)*, volume 6293 of *Lecture Notes in Bioinformatics*, Liverpool, UK, September 2010. Springer-Verlag.

[152] K. M. Swenson, Y. Lin, V. Rajan, and B. M. Moret. Hurdles hardly have to be heeded. In *Proceedings of the International Workshop on Comparative Genomics (RECOMB-CG'08)*, volume 5267 of *Lecture Notes in Computer Science*, pages 241–251, Paris, 2008.

[153] K. M. Swenson, V. Rajan, Y. Lin, and B. M. E. Moret. Sorting signed permutations by inversions in $O(n \log n)$ time. *Journal of Computational Biology*, 17(3):489–501, 2010.

[154] F. Swidan, E. P. C. Rocha, M. Shmoish, and R. Y. Pinter. An integrative method for accurate comparative genome mapping. *PLoS Computational Biology*, 2(8), 2006. 20 pages.

[155] E. Tannier, A. Bergeron, and M.-F. Sagot. Advances on sorting by reversals. *Discrete Applied Mathematics*, 155:881–888, April 2007.

[156] R. L. Tatusov, E. V. Koonin, and D. J. Lipman. A genomic perspective on protein families. *Science*, 278(5338):631–637, 1997.

[157] P. Trinh, A. McLysaght, and D. Sankoff. Genomic features in the breakpoint regions between syntenic blocks. *Bioinformatics*, 20(Suppl. 1):I318–I325, 2004.

[158] UCSC Genome Bioinformatics, 2002. http://genome.ucsc.edu/.

[159] A. E. van der Wind, S. R. Kata, M. R. Band, M. Rebeiz, D. M. Larkin, R. E. Everts, C. A. Green, L. Liu, S. Natarajan, T. Goldammer, et al. A 1463 gene cattle-human comparative map with anchor points defined by human genome sequence coordinates. *Genome Research*, 14(7):1424–1437, 2004.

[160] K. Vandepoele, Y. Saeys, C. Simillion, J. Raes, and Y. V. D Peer. The automatic detection of homologous regions (ADHoRe) and its application to microcolinearity between *Arabidopsis* and rice. *Genome Research*, 12(11):1792–1801, 2002.

[161] A. F. Vellozo, C. E. R. Alves, and A. P. do Lago. Alignment with non-overlapping inversions in $O(n^3)$-time. In *Algorithms in Bioinformatics*, volume 4175 of *Lecture Notes in Computer Science*, pages 186–196. Springer-Verlag Berlin Heidelberg, 2006. Proceedings of the WABI 2006.

[162] A. J. Vilella, J. Severing, A. Ureta-Vidal, L. Heng, R. Durbin, and E. Birney. EnsemblCompara GeneTrees: Complete, duplication-aware phylogenetic trees in vertebrates. *Genome Research*, 19(2):327–335, 2009.

[163] M. E. M. T. Walter, Z. Dias, and J. Meidanis. Reversal and transposition distance of linear chromosomes. In *Proceedings of the String Processing and Information Retrieval (SPIRE'98)*, 1998.

[164] M. E. M. T. Walter, Z. Dias, and J. Meidanis. A new approach for approximating the transposition distance. In *Proceedings of the String Processing and Information Retrieval (SPIRE'2000)*, September 2000.

[165] L. Wang, D. Zhu, X. Liu, and S. Ma. An $O(n^2)$ algorithm for signed translocation. *Journal of Computer and System Sciences*, 70(3):284–299, 2005.

[166] L.-S. Wang. Exact-IEBP: A new technique for estimating evolutionary distances between whole genomes. *Lecture Notes in Computer Science*, 2149:175–188, 2001.

[167] L.-S. Wang and T. Warnow. Estimating true evolutionary distances between genomes. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 637–646, New York, NY, USA, 2001. ACM.

[168] R. H. Waterston, K. Lindblad-Toh, E. Birney, J. Rogers, J. F. Abril, P. Agarwal, R. Agarwala, R. Ainscough, M. Alexandersson, P. An, et al. Initial sequencing and comparative analysis of the mouse genome. *Nature*, 420(6915):520–562, 2002.

[169] L. G. Wilming, G. R. Gilbert, K. Howe, S. Trevanion, T. Hubbard, and J. L. Harrow. The vertebrate genome annotation (Vega) database. *Nucleic Acids Research*, 36(Suppl. 1):D753–D760, 2008.

[170] Y. Yue, B. Grossmann, M. Ferguson-Smith, F. Yang, and T. Haaf. Comparative cytogenetics of human chromosome 3q21.3 reveals a hot spot for ectopic recombination in hominoid evolution. *Genomics*, 85(1):36–47, 2005.

[171] J. J. Yunis, J. R. Sawyer, and K. Dunham. The striking resemblance of high-resolution g-banded chromosomes of man and chimpanzee. *Science*, 208(4448):1145–1148, 1980.

[172] E. M. Zdobnov, C. von Mering, I. Letunic, D. Torrents, M. Suyama, R. R. Copley, G. K. Christophides, D. Thomasova, R. A. Holt, G. M. Subramanian, et al. Comparative genome and proteome analysis of anopheles gambiae and drosophila melanogaster. *Science*, 298(5591):149–159, 2002.

[173] Z. Zhang, B. Raghavachari, R. C. Hardison, and W. Miller. Chaining multiple-alignment blocks. *Journal of Computational Biology*, 1(3):217–226, 1994.

[174] C. Zheng and D. Sankoff. Rearrangement of noisy genomes. In V. N. Alexandrov et al., editors, *Proceedings of the International Conference on Computational Science 2006 (ICCS 2006)*, volume 3992 of *Lecture Notes in Computer Science*, pages 791–798, Reading, UK, May 2006. Springer-Verlag Berlin Heidelberg.

[175] D. M. Zhu and S. H. Ma. Improved polynomial-time algorithm for computing translocation distance between genomes. *Chinese J. Comput.*, 25(2):189–196, 2002. In Chinese.