Montagem de Fragmentos de DNA pelo Método "Ordered Shotgun Sequencing" (OSS)

Lin Tzy Li

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Montagem de Fragmentos de DNA pelo Método "Ordered Shotgun Sequencing" (OSS)

Lin Tzy Li[1]

Fevereiro de 2001

Banca Examinadora:

- João Meidanis (Orientador)
- Marco Dimas Gubitoso
  IME - USP
- João Paulo Kitajima
  CBMEG - UNICAMP
- Arnaldo Vieira Moura (Suplente)
  IC - UNICAMP

[1]**Projeto financiado pela FAPESP, sob número de processo 97/11628-6.**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Lin, Tzy Li

L63m Montagem de fragmentos de DNA pelo método "Ordered Shotgun Sequencing" (OSS) / Lin Tzy Li -- Campinas, [S.P. :s.n.], 2001.

Orientador : João Meidanis

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Algoritmos. 2. Biologia molecular. 3. Teoria da computação. I. Meidais, João. II. Universidade Estadual de Campinas. Instituto de Computação . III. Título.

Montagem de Fragmentos de DNA pelo Método "Ordered Shotgun Sequencing" (OSS)

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Lin Tzy Li e aprovada pela Banca Examinadora.

Campinas, 23 de fevereiro de 2001.

João Meidanis (Orientador)

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

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 23 de fevereiro de 2001, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Marco Dimas Gubitoso
USP

João Paulo J. W. Kitajima

Prof. Dr. João Paulo Kitajima
CBMEG – UNICAMP

Prof. Dr. João Meidanis
IC - UNICAMP

"Descobrir consiste em olhar para o que todo mundo está vendo e pensar uma coisa diferente."
Albert Szent-Györgyi

"Nada é mais perigoso do que uma idéia quando ela é a única que você tem."
Emile Chartier

"Transportai um punhado de terra todos os dias, e fareis uma montanha."
Confúcio

Agradecimentos

"... para criaturas pequenas como nós, a vastidão só é suportável através do amor."
Carl Sagan, *Contato*

Eu gostaria de agradecer a todos os que me ajudaram a tornar essa dissertação possível, seja através do convívio cotidiano, seja por incentivos morais, espirituais, sentimentais ou financeiros.

Não poderia deixar de registrar aqui minha gratidão à FAPESP pelo financiamento desde o início; a todos os meus colegas e amigos de convívio e apoio diário no LBI (Laboratório de Bioinformática), principalmente aqueles que acompanham meu caminho desde antes do mestrado: Guilherme Pimentel, João Carlos Setubal, Maria Emília Walter, Marília D. V. Braga, Vagner Katsumi e Zanoni Dias; ao LAD (Laboratório de Alto Desempenho) por permitir rodar os testes feitos para este trabalho em sua máquina mais poderosa, em especial ao professor Jorge Stolfi pela paciência em administrar a máquina e se dispor a resolver eventuais problemas; ao LBI pela infraestrutura do laboratório, pela oportunidade de presenciar de perto a correria que é participar de projetos genomas e pelo apoio e compreensão de João Meidanis e João Paulo Kitajima às necessidades de tempo para terminar esta dissertação.

Um sonoro obrigado aos amigos que souberam ouvir e deram seus incentivos nos momentos difíceis, tornando os obstáculos mais fáceis de se transpor; aos meus amigos de moradia estudantil pela calorosa acolhida nos últimos anos; ao meu orientador João Meidanis pelas idéias, debates e críticas bastante elucidativas, pelo empenho em dar todo o suporte necessário a fim de me "convencer" a usar o LaTeX para editar esta dissertação e, acima de tudo, por não ter perdido as esperanças de algum dia ver este trabalho concluído.

Por fim, gostaria de agradecer à minha família: mãe, irmão, tios e primos que esperaram tanto por este momento; e à minha quase segunda família pela torcida e conselhos, em especial ao Zanoni Dias pelas críticas, direções e idéias sempre iluminadas e valiosas para o meu trabalho, à Lara pela paciente leitura e correção dos capítulos inicias e à D. Antônia pelo carinho com que sempre me recebeu em sua casa.

Resumo

Esta dissertação é na área de Biologia Computacional e se propõe a estudar a montagem de fragmentos de DNA pelo método *Ordered Shotgun Sequencing* (OSS).

A montagem de fragmentos de DNA é uma das etapas de um projeto de seqüenciamento total do genoma de um organismo. Com o crescimento do tamanho dos genomas a serem seqüenciados e do aumento da complexidade dos organismos estudados, os problemas para se montar um DNA, como trechos repetidos no genoma, também aumentaram.

Até os anos noventa, o método mais popular de montagem de fragmentos era o Shotgun, que quebra a molécula de DNA em fragmentos de mais ou menos 500 pares de bases que serão desvendados em laboratórios de seqüenciamento. Conhecendo-se os fragmentos, é a vez de se descobrir como eles se encaixam, através de sobreposições, e ordená-los. No final, espera-se que os algoritmos nos retornem a seqüência do DNA.

Em 1993, dois biólogos propuseram a montagem de fragmentos OSS, que quebra a molécula de DNA em intervalos bem maiores, resultando em fragmentos que chamaremos de clones, e seqüencia apenas as suas pontas, deixando a parte central desconhecida. Agora, além da seqüência dos fragmentos, teremos a informação de ligação e distância entre as pontas do clone. Com essas informações e das sobreposições entre as pontas, tenta-se encontrar as posições relativas dos clones no DNA. Um bloco ordenado de fragmentos de DNA por causa das sobreposições e das ligações de clone, será denominado *scaffold*. As suas partes desconhecidas serão desvendadas em outras iterações, que é uma característica da montagem de fragmentos de DNA OSS.

Em 1998, ano que nossos estudos começaram, não haviam trabalhos computacionais que tirassem proveito dessa técnica, até que em 2000 surgiram trabalhos relatando projetos genoma que utilizaram a montagem OSS com sucesso.

Em nosso trabalho, propusemos dois algoritmos para encontrar scaffolds e um ambiente de simulação para executar as iterações feitas pelo método OSS e montar genomas artificialmente gerados. Foram feitos vários testes para medir o desempenho dos algoritmos propostos em termos de tempo total de processamento em CPU, e número de iterações e de clones seqüenciados para terminar a montagem OSS. Por último, identificamos várias questões importantes sobre os algoritmos que merecem um estudo mais aprofundado.

Abstract

This is a dissertation in the area of Computational Biology. We propose to study the DNA fragment assembly problem using the *Ordered Shotgun Sequencing* (OSS) method.

The DNA fragment assembly problem is one step of a complete genome sequencing project. With the growing size of the sequenced genomes and the increasing complexity of the studied organisms, problems such as repeated regions increased as well. The OSS method is an attempt to mitigate those problems.

Until the nineties, the most popular method of DNA fragment assembly was Shotgun, which breaks the DNA molecule into pieces of about 500 base pairs that are revealed by sequencing laboratories. Given the fragments, it is necessary to find out what their relative position is in the original DNA molecule by using overlap information to order them. In the end, the algorithms are supposed to give us the DNA sequence.

In 1993 two biologists proposed the OSS method for fragment assembly, which breaks the DNA into quite bigger pieces, resulting in fragments called clones in this dissertation. Just the ends of the clones are sequenced, leaving the central part unknown. Besides the fragment sequences, we know how far one end is from the other, and their relative orientation. Based on this information and on the overlaps between clone ends, an ordered block of DNA fragments called *scaffold* can be formed. The unknown part will be revealed by subsequent iterations, an intrinsic characteristic of OSS DNA fragment assembly.

Our studies began in 1998 when there was no computational work exploring the OSS technique. The first papers reporting on the successful use of OSS assembly in genome projects appeared in 2000.

In this work, we propose two scaffold-finding algorithms and present a simulation environment built to test them. The environment generates artificial data, promotes the iterations, and keeps track of the amount of sequencing, number of iterations, and processing time for the algorithm being tested. Finally, we identified many important questions about the algorithms which deserve further studies.

Conteúdo

Agradecimentos viii

Resumo ix

Abstract x

1 Introdução 1
- 1.1 Conceitos . . . 3
- 1.2 Montagem de Fragmentos de DNA . . . 5
- 1.3 O método OSS - Ordered Shotgun Sequencing . . . 8
- 1.4 Modelagem dos Dados . . . 11
- 1.5 Formalização do problema . . . 13

2 Algoritmos Scaffold 17
- 2.1 Algoritmo 1 - O Guloso . . . 17
- 2.2 Algoritmo 2 - A Maior Componente Conexa OSS . . . 21
  - 2.2.1 Adaptações para o caso do genoma circular . . . 28

3 O Ambiente de Simulação 31
- 3.1 Gerador de Dados . . . 31
- 3.2 Gerenciador de Iterações . . . 34
  - 3.2.1 Parâmetros . . . 35
  - 3.2.2 Arquivos necessários para rodar cada iteração . . . 35
  - 3.2.3 Arquivos modificados a cada iteração . . . 35
  - 3.2.4 Arquivos criados . . . 36
  - 3.2.5 O processamento de cada iteração por submódulos . . . 37
- 3.3 Os submódulos do gerenciador . . . 38
  - 3.3.1 Comparação de seqüências . . . 38
  - 3.3.2 Construção do grafo de sobreposição OSS . . . 40
  - 3.3.3 À procura do *Scaffold* . . . 47

3.3.4 O mapa dos dados 48
3.3.5 Seqüenciador 50

4 Testes e Resultados **51**
4.1 Geração e Teste com Dados Artificiais 51
4.2 Resultados dos testes 52
4.2.1 Resultados com o Algoritmo 1 53
4.2.2 Resultados com o Algoritmo 2 53
4.3 Avaliação dos dois algoritmos Scaffold 58
4.4 Avaliação do gerenciador de iterações por algoritmo 59
4.4.1 Tempo total do gerenciador 60
4.4.2 Gerenciador de iterações módulo a módulo 60
4.5 Considerações finais 61

5 Conclusões **63**
5.1 Contribuições 64
5.2 Trabalhos futuros 67

A Tabelas **69**
A.1 Resultados com Algoritmo 1 69
A.2 Resultados com Algoritmo 2 75

B Gráficos **81**
B.1 Gráficos: tempo dos algoritmos scaffold 81
B.2 Gráficos: total de iterações executadas 88
B.3 Gráficos: esforço de seqüenciamento 95
B.4 Gráficos: tempo total da montagem OSS 102
B.5 Gráficos: percentual de tempo dos módulos do gerenciador com algoritmo 1 108
B.6 Gráficos: percentual de tempo dos módulos do gerenciador com algoritmo 2 114

Bibliografia **121**

Lista de Tabelas

2.1 Tabela das principais diferenças entre EstendeX() e EstendeY() 21

4.1 Diagnóstico dos resultados das tabelas para o algoritmo 2. 55

A.1 Resultados com algoritmo 1 para conjunto DNA100K_13c40K_p500. 69
A.2 Resultados com algoritmo 1 para conjunto DNA200K_25c40K_p500. 70
A.3 Resultados com algoritmo 1 para conjunto DNA300K_38c40K_p500. 70
A.4 Resultados com algoritmo 1 para conjunto DNA400K_50c40K_p500. 71
A.5 Resultados com algoritmo 1 para conjunto DNA500K_63c40K_p500. 71
A.6 Resultados com algoritmo 1 para conjunto DNA600K_75c40K_p500. 72
A.7 Resultados com algoritmo 1 para conjunto DNA700K_88c40K_p500. 72
A.8 Resultados com algoritmo 1 para conjunto DNA800K_100c40K_p500. 73
A.9 Resultados com algoritmo 1 para conjunto DNA900K_113c40K_p500. 73
A.10 Resultados com algoritmo 1 para conjunto DNA1000K_125c40K_p500. 74
A.11 Resultados com algoritmo 1 para conjunto DNA3M_375c40K_p450. 74
A.12 Resultados com algoritmo 2 para conjunto DNA100K_13c40K_p500. 75
A.13 Resultados com algoritmo 2 para conjunto DNA200K_25c40K_p500. 76
A.14 Resultados com algoritmo 2 para conjunto DNA300K_38c40K_p500. 76
A.15 Resultados com algoritmo 2 para conjunto DNA400K_50c40K_p500. 77
A.16 Resultados com algoritmo 2 para conjunto DNA500K_63c40K_p500. 77
A.17 Resultados com algoritmo 2 para conjunto DNA600K_75c40K_p500. 78
A.18 Resultados com algoritmo 2 para conjunto DNA700K_88c40K_p500. 78
A.19 Resultados com algoritmo 2 para conjunto DNA800K_100c40K_p500. 79
A.20 Resultados com algoritmo 2 para conjunto DNA900K_113c40K_p500. 79
A.21 Resultados com algoritmo 2 para conjunto DNA1000K_125c40K_p500. 80
A.22 Resultados com algoritmo 2 para conjunto DNA3M_375c40K_p450. 80

Lista de Figuras

1.1 Sumário: do nucleotídeo ao cromossomo 4
1.2 Esquema da estrutura molecular da fita dupla de DNA. 6
1.3 Problema da falta de cobertura. 8
1.4 Exemplo de uma montagem OSS. 9
1.5 Exemplo ilustrativo da iteração OSS. 10
1.6 Esquema da leitura de um fragmento no método OSS. 11
1.7 Ilustração da definição de clone como par de cadeias. 12
1.8 Representação gráfica de um clone seqüenciado nas extremidades. 13
1.9 Esquema gráfico versus representação em grafo. 14

2.1 Representação gráfica do algoritmo guloso 18
2.2 Corpo principal do algoritmo guloso: SCAFFOLD1. 19
2.3 Possibilidades para estender o scaffold à direita. 19
2.4 Possibilidades para estender o scaffold à esquerda. 19
2.5 Subrotina do SCAFFOLD1: ESTENDEY. 20
2.6 Algoritmo SCAFFOLD2. 22
2.7 Mapeamento equivalente dos vértices v, w, x. 23
2.8 Algoritmo MAPEÁVEL e sua subrotina DFS. 25
2.9 Escolha do fragmento de maior coordenada: fragmento D. 26
2.10 Algoritmo MAPEÁVEL2 e sua subrotina DFS2. 28
2.11 O mapa de distribuição dos fragmentos de um genoma circular. 29
2.12 Ligações eliminadas (marcado com 'X') para tratar o caso do genoma circular. 30

3.1 Esquema geral do ambiente de simulação OSS. 32
3.2 Visão geral do gerador de dados. 33
3.3 Visão geral do gerenciador de iterações. 39
3.4 Gráfico de evolução do tempo dos módulos no Monta Grafo antigo. 44
3.5 Gráfico de comparação de tempo das duas versões do Monta Grafo. 45
3.6 Tabela de dados das duas versões do Monta Grafo. 46
3.7 Exemplo do mapa de distribuição dos dados (apenas de 143333–200000). . 49

4.1 Mapa da montagem final DNA100K_13c40K_p500 com 1 X (editado). 56
4.2 Mapa da montagem do DNA100K_13c40K_p500 com 7 X (editado). 56
4.3 Mapa da montagem do DNA200K_25c40K_p500 com 4 X (editado). 57
4.4 Mapa da montagem do DNA700K_88c40K_p500 com 3 X (editado). 57

5.1 Nova formulação teórica. 66

B.1 Tempo dos algoritmos scaffold para conjunto DNA100K_13c40K_p500. 81
B.2 Tempo dos algoritmos scaffold para conjunto DNA200K_25c40K_p500. 82
B.3 Tempo dos algoritmos scaffold para conjunto DNA300K_38c40K_p500. 82
B.4 Tempo dos algoritmos scaffold para conjunto DNA400K_50c40K_p500. 83
B.5 Tempo dos algoritmos scaffold para conjunto DNA500K_63c40K_p500. 83
B.6 Tempo dos algoritmos scaffold para conjunto DNA600K_75c40K_p500. 84
B.7 Tempo dos algoritmos scaffold para conjunto DNA700K_88c40K_p500. 84
B.8 Tempo dos algoritmos scaffold para conjunto DNA800K_100c40K_p500. 85
B.9 Tempo dos algoritmos scaffold para conjunto DNA900K_113c40K_p500. 85
B.10 Tempo dos algoritmos scaffold para conjunto DNA1000K_125c40K_p500. 86
B.11 Tempo dos algoritmos scaffold para conjunto DNA3M_375c40K_p450. 86
B.12 Tempo total dos algoritmos scaffold para os conjuntos. 87
B.13 Total de iterações para DNA100K_13c40K_p500. 88
B.14 Total de iterações para DNA200K_25c40K_p500. 89
B.15 Total de iterações para DNA300K_38c40K_p500. 89
B.16 Total de iterações para DNA400K_50c40K_p500. 90
B.17 Total de iterações para DNA500K_63c40K_p500. 90
B.18 Total de iterações para DNA600K_75c40K_p500. 91
B.19 Total de iterações para DNA700K_88c40K_p500. 91
B.20 Total de iterações para DNA800K_100c40K_p500. 92
B.21 Total de iterações para DNA900K_113c40K_p500. 92
B.22 Total de iterações para DNA1000K_125c40K_p500. 93
B.23 Total de iterações para DNA3M_375c40K_p450. 93
B.24 Total geral de iterações para DNA de até 1Mbp. 94
B.25 Esforço de seqüenciamento para DNA100K_13c40K_p500. 95
B.26 Esforço de seqüenciamento para DNA200K_25c40K_p500. 96
B.27 Esforço de seqüenciamento para DNA300K_38c40K_p500. 96
B.28 Esforço de seqüenciamento para DNA400K_50c40K_p500. 97
B.29 Esforço de seqüenciamento para DNA500K_63c40K_p500. 97
B.30 Esforço de seqüenciamento para DNA600K_75c40K_p500. 98
B.31 Esforço de seqüenciamento para DNA700K_88c40K_p500. 98
B.32 Esforço de seqüenciamento para DNA800K_100c40K_p500. 99

B.33 Esforço de seqüenciamento para DNA900K_113c40K_p500. 99
B.34 Esforço de seqüenciamento para DNA1000K_125c40K_p500. 100
B.35 Esforço de seqüenciamento para DNA3M_375c40K_p450. 100
B.36 Esforço de seqüenciamento total por conjunto. 101
B.37 Tempo da montagem OSS para DNA100K_13c40K_p500. 102
B.38 Tempo da montagem OSS para DNA200K_25c40K_p500. 103
B.39 Tempo da montagem OSS para DNA300K_38c40K_p500. 103
B.40 Tempo da montagem OSS para DNA400K_50c40K_p500. 104
B.41 Tempo da montagem OSS para DNA500K_63c40K_p500. 104
B.42 Tempo da montagem OSS para DNA600K_75c40K_p500. 105
B.43 Tempo da montagem OSS para DNA700K_88c40K_p500. 105
B.44 Tempo da montagem OSS para DNA800K_100c40K_p500. 106
B.45 Tempo da montagem OSS para DNA900K_113c40K_p500. 106
B.46 Tempo da montagem OSS para DNA1000K_125c40K_p500. 107
B.47 Tempo da montagem OSS para DNA3M_375c40K_p450. 107
B.48 Tempo percentual para DNA100K_13c40K_p500 (algoritmo 1). 108
B.49 Tempo percentual para DNA200K_25c40K_p500 (algoritmo 1). 109
B.50 Tempo percentual para DNA300K_38c40K_p500 (algoritmo 1). 109
B.51 Tempo percentual para DNA400K_50c40K_p500 (algoritmo 1). 110
B.52 Tempo percentual para DNA500K_63c40K_p500 (algoritmo 1). 110
B.53 Tempo percentual para DNA600K_75c40K_p500 (algoritmo 1). 111
B.54 Tempo percentual para DNA700K_88c40K_p500 (algoritmo 1). 111
B.55 Tempo percentual para DNA800K_100c40K_p500 (algoritmo 1). 112
B.56 Tempo percentual para DNA900K_113c40K_p500 (algoritmo 1). 112
B.57 Tempo percentual para DNA1000K_125c40K_p500 (algoritmo 1). 113
B.58 Tempo percentual para DNA3M_375c40K_p450 (algoritmo 1). 113
B.59 Tempo percentual para DNA100K_13c40K_p500 (algoritmo 2). 114
B.60 Tempo percentual para DNA200K_25c40K_p500 (algoritmo 2). 115
B.61 Tempo percentual para DNA300K_38c40K_p500 (algoritmo 2). 115
B.62 Tempo percentual para DNA400K_50c40K_p500 (algoritmo 2). 116
B.63 Tempo percentual para DNA500K_63c40K_p500 (algoritmo 2). 116
B.64 Tempo percentual para DNA600K_75c40K_p500 (algoritmo 2). 117
B.65 Tempo percentual para DNA700K_88c40K_p500 (algoritmo 2). 117
B.66 Tempo percentual para DNA800K_100c40K_p500 (algoritmo 2). 118
B.67 Tempo percentual para DNA900K_113c40K_p500 (algoritmo 2). 118
B.68 Tempo percentual para DNA1000K_125c40K_p500 (algoritmo 2). 119
B.69 Tempo percentual para DNA3M_375c40K_p450 (algoritmo 2). 119

Capítulo 1

Introdução

A montagem de fragmentos é uma das etapas de um projeto de seqüenciamento total do genoma de um organismo.

Apresentaremos aqui um método de montagem de fragmentos chamado *Ordered Shotgun Sequencing* (OSS), proposto em 1993 por E. Chen e seus colegas [4]. Usando as idéias apresentadas por eles, Myers e Weber propuseram o método *whole-genome shotgun* [13, 11], que veio a ser empregado na montagem do genoma da *Drosophila*, cujos resultados foram publicados em 2000 [12]. No final de 2000, um relatório técnico foi escrito por Setubal e Werneck [17] sobre um programa usado no projeto genoma da *Xanthomonas axonopodis* pv. *citri* [14], utilizando o método proposto por Myers e Weber.

Na época em que começamos a estudar o método do ponto de vista computacional (Jan/1998), não se sabia de um software para montagem que tirasse vantagem da estratégia OSS. Por isso, resolvemos estudá-la, propondo algoritmos e produzindo uma ferramenta de simulação da montagem OSS para testá-los e avaliá-los.

Para entender melhor os problemas mencionados é preciso saber que as máquinas existentes para seqüenciar o DNA conseguem ler, no máximo, 1000 pares de bases. Por isso, faz-se necessário quebrar o DNA, de milhares ou até milhões de pares de bases, em pedaços de tamanhos legíveis pelas seqüenciadoras — os fragmentos ou *reads* — para depois serem remontados, alinhados e posicionados relativamente entre si através das sobreposições encontradas entre eles.

A seqüência do DNA reconstruída é conhecida como *consenso* e o processo de reconstrução, como *montagem de fragmentos*, na qual o trabalho computacional se tornou fundamental à medida que os organismos escolhidos para se estudar eram mais complexos, portanto com genomas maiores e mais complicados.

Numa tentativa de amenizar os problemas de montagem como as repetições — problemas crescentes, devido à complexidade dos genomas estudados — é que foi proposto o método Ordered Shotgun Sequencing (OSS) em 1993. Nesta época, e ainda hoje, o

método mais empregado para a montagem de fragmentos de DNA era o *Shotgun*.

A novidade da montagem de fragmentos OSS é o clone, que representa um fragmento do DNA muito maior do que um seqüenciador pode ler, mais de 2 mil até centenas de milhares de pares de bases. Inicialmente, as únicas partes seqüenciadas do clone são suas extremidades (pontas), com tamanhos em torno de 500 a 1000 pares de bases.

A vantagem de se ter um clone grande é que há grandes chances deste isolar em si cópias individuais de regiões repetidas do genoma e diminuir os problemas com repetições na montagem.

Uma outra característica da montagem de fragmentos OSS é a iteratividade. A cada iteração, é procurado o esqueleto da montagem da região que mais se expande, fundamentado nas ligações por sobreposição entre as pontas e pela ligação natural dos pares de pontas pertencentes ao mesmo clones. A esse "esqueleto" formado, dá-se o nome de *scaffold*. Um scaffold é, então, um conjunto de clones orientados, posicionados relativamente uns aos outros, onde se conhece o tamanho aproximado dos buracos entre os fragmentos [12].

Após isso, um pequeno grupo de clones que expande a região do scaffold são completamente seqüenciados e montados, resultando num consenso parcial. Este é inserido na montagem principal e usado na filtragem e remoção de todos os clones que caiam inteiramente na região seqüenciada, bem como dos reads que fizeram parte do scaffold anterior. Somente assim, uma nova iteração se inicia, com nova informação inserida e um conjunto de dados mais enxuto. As iterações vão acontecendo até que não haja mais novas sobreposições para encontrar novos scaffolds. O resultado, após a última iteração, será um consenso ou vários *contigs* [13] — porções reconstruídas e contíguas do DNA, mas que não se sobrepõem por falta de fragmentos nas regiões que as ligariam.

Com os principais conceitos apresentados, podemos explicar que o programa de Setubal e Werneck [17] utiliza os resultados de um software de montagem de fragmentos de DNA, por exemplo o `phrap` [10], para construir um scaffold que se baseia nas informações de ligação das pontas dos clones para juntar e ordenar as seqüências dos contigs formados pelo software. Em contra partida, nós vislumbramos um programa que faça a montagem OSS desde o início.

A nossa idéia foi construir um ambiente de simulação da montagem OSS, com o objetivo inicial de testar e medir o desempenho dos algoritmos propostos para encontrar o melhor scaffold a cada iteração. A ferramenta produzida é capaz de gerar e gerenciar os mais variados conjuntos de dados para os testes.

Mediu-se o desempenho dos algoritmos com parâmetros como número de iterações necessárias para montar o DNA, o percentual montado a cada iteração, o número de clones que foi preciso seqüenciar e a quantidade de bases que isto representou e o tempo de processamento.

1.1 Conceitos

Nosso trabalho é na área de Biologia Computacional, que basicamente consiste em desenvolver e usar as técnicas matemáticas e computacionais para ajudar a resolver os problemas de uma subárea da Biologia, a Biologia Molecular [16].

No Brasil, essa área teve sua importância finalmente reconhecida no país com a atuação decisiva e bastante elogiada da chamada Bioinformática no Projeto Genoma da *Xylella fastidiosa*. Os resultados deste projeto foram publicados no ano 2000 [18].

Para quem trabalha com Biologia Computacional, é essencial conhecer alguns conceitos básicos da área de Biologia Molecular. Por isso, daremos uma visão geral de alguns conceitos para melhor entendimento dos problemas tratados na dissertação.

Inicialmente, versaremos sobre o genoma de um organismo, aqui definido como sendo o conjunto de todos os cromossomos de sua célula. A quantidade de cromossomos no genoma varia de espécie para espécie [5, 21].

Um cromossomo é composto por uma molécula de DNA (ácido desoxirribonucléico) que, por sua vez, é uma fita dupla helicoidal. Cada fita simples é constituída por uma seqüência de nucleotídeos. É no DNA onde estão armazenadas todas as informações genéticas do organismo.

Um cromossomo pode ser demarcado, ao longo de seu comprimento, em milhares de regiões chamadas de genes [5], que codificam informações para a construção de todas as proteínas necessárias a um determinado organismo vivo [16].

O nucleotídeo é formado por três componentes: uma base nitrogenada, um carboidrato (açúcar) e um grupo fosfato.

São quatro as bases nitrogenadas que formam os nucleotídeos de DNA: Adenina, Citosina, Guanina, Timina, representadas por A, C, G, T respectivamente. Os conceitos aqui apresentados estão esquematizados na figura 1.1.

Havíamos dito que o DNA é formado por uma fita dupla e, sabendo-se o conteúdo de uma fita, automaticamente se saberá a codificação da outra, pois cada base A de uma fita se liga à base T da outra fita, na posição correspondente, e vice-versa. Da mesma forma, C sempre se liga à base G na outra fita; por isso, dizemos que uma é o complemento da outra [16, 21].

Além das duas fitas do DNA serem complementares, elas são antiparalelas, ou seja, uma fita complementa a outra e estão em direção oposta, portanto, uma fita é o complemento reverso da outra [16, 19].

O conceito de direção surge por causa das ligações da molécula de açúcar do nucleotídeo que formam a espinha dorsal (a fita) do DNA. A molécula de açúcar (2'-desoxiribose) é constituída por cinco átomos de carbono rotulados de 1' a 5'. O carbono 3' de um açúcar se liga ao grupo fosfato que, por sua vez, se liga ao carbono 5' do carboidrato do outro

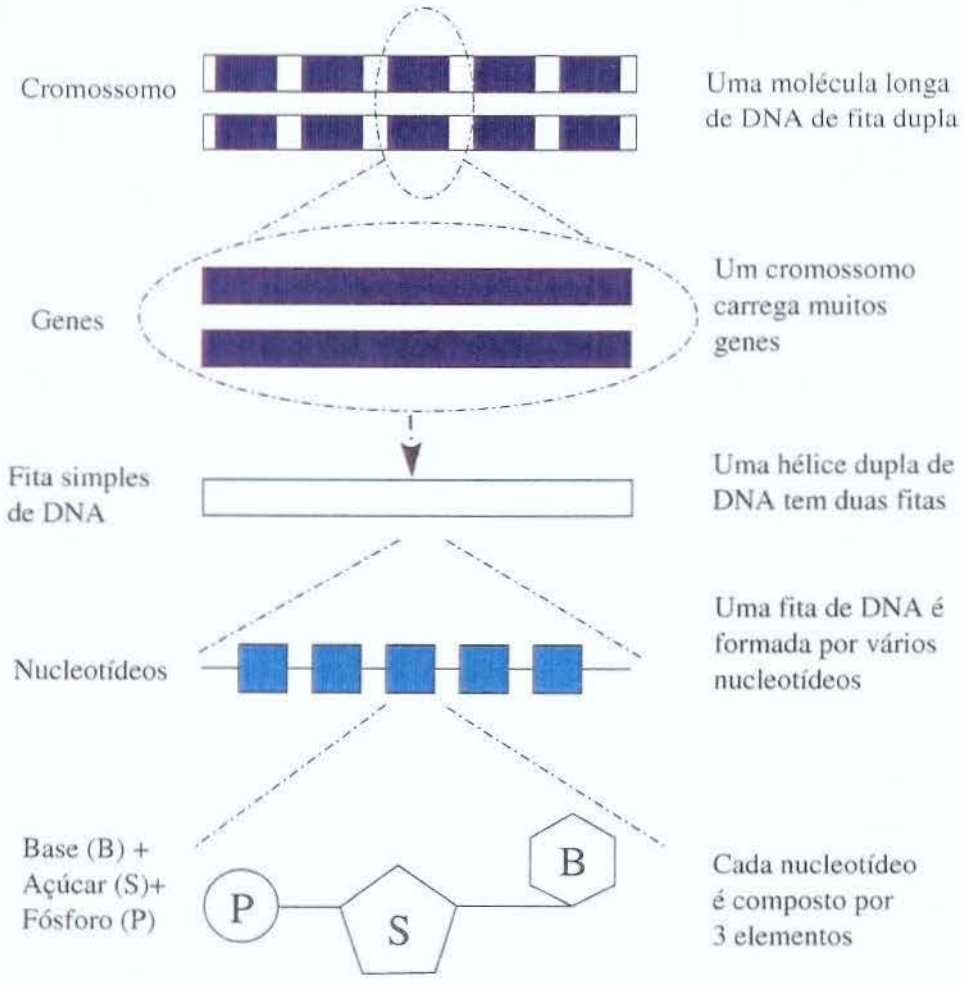

Figura 1.1: Sumário: do nucleotídeo ao cromossomo

nucleotídeo. Por convenção, a espinha dorsal começa na extremidade 5' e termina na 3'. Esta é a direção canônica [16, 21, 7] e é denotada por direção 5' $\rightarrow$ 3'. A estrutura molecular pode ser melhor compreendida observando-se a figura 1.2.

Ao dizer que queremos seqüenciar um DNA estamos falando em determinar a seqüência das bases de nucleotídeos que compõem o DNA. Medimos o tamanho da fita de DNA pelo número de nucleotídeos que esta possui, no entanto, dificilmente ouviremos que um DNA possui x nucleotídeos, mas sim que um DNA possui x *pares de bases*, ou abreviadamente, x *bp* (do inglês *base pair*) [16]. Outras formas de quantificação são feitas em milhares de pares de bases, cuja abreviação é *kbp*, ou milhões de pares de bases (*Mbp*).

Como o genoma é o conjunto de todos os cromossomos de uma célula, fica claro que o tamanho do genoma é medido pela soma do número de bases de todos os cromossomos.

Estudar um genoma é basicamente mapear os genes após desvendar a seqüência dos pares de bases que formam o DNA. O primeiro passo para se atingir tal meta é o seqüenciamento do DNA [16].

Para se ter uma idéia do problema que se enfrenta, imaginemos o tamanho do genoma da levedura (*Saccharomyces cerevisiae*), de 10 milhões de bp, e o do genoma humano estimado em, aproximadamente, 10^9 bp. Levando-se em consideração que o maior segmento de DNA que pode ser seqüenciado diretamente nos laboratório é de mais ou menos 700 bp, concluiremos que temos uma incompatibilidade entre o que podemos realmente fazer e o que queremos, numa ordem de 100.000 vezes [16].

A solução encontrada pelos biólogos para resolver tal problema foi a de quebrar o DNA de forma variada, em pedaços que eles possam manipular e depois reconstruir a solução final através das soluções parciais obtidas, com a ajuda de técnicas computacionais. É esta a área de *montagem de fragmentos de DNA* [16, 5].

1.2 Montagem de Fragmentos de DNA

O método mais empregado pelos laboratórios para a quebra da molécula de DNA em vários pedaços menores é o *Shotgun*. Neste método, basicamente, segue-se alguns passos bem definidos [13, 2]:

1. obtém-se muitas cópias do DNA a ser seqüenciado para poder manipulá-los posteriormente;

2. particiona-se cada cópia aleatoriamente através de métodos físicos, o que resulta em um conjunto de *fragmentos*;

3. os fragmentos são selecionados conforme um tamanho médio definido. Fragmentos grandes ou pequenos demais são desprezados;

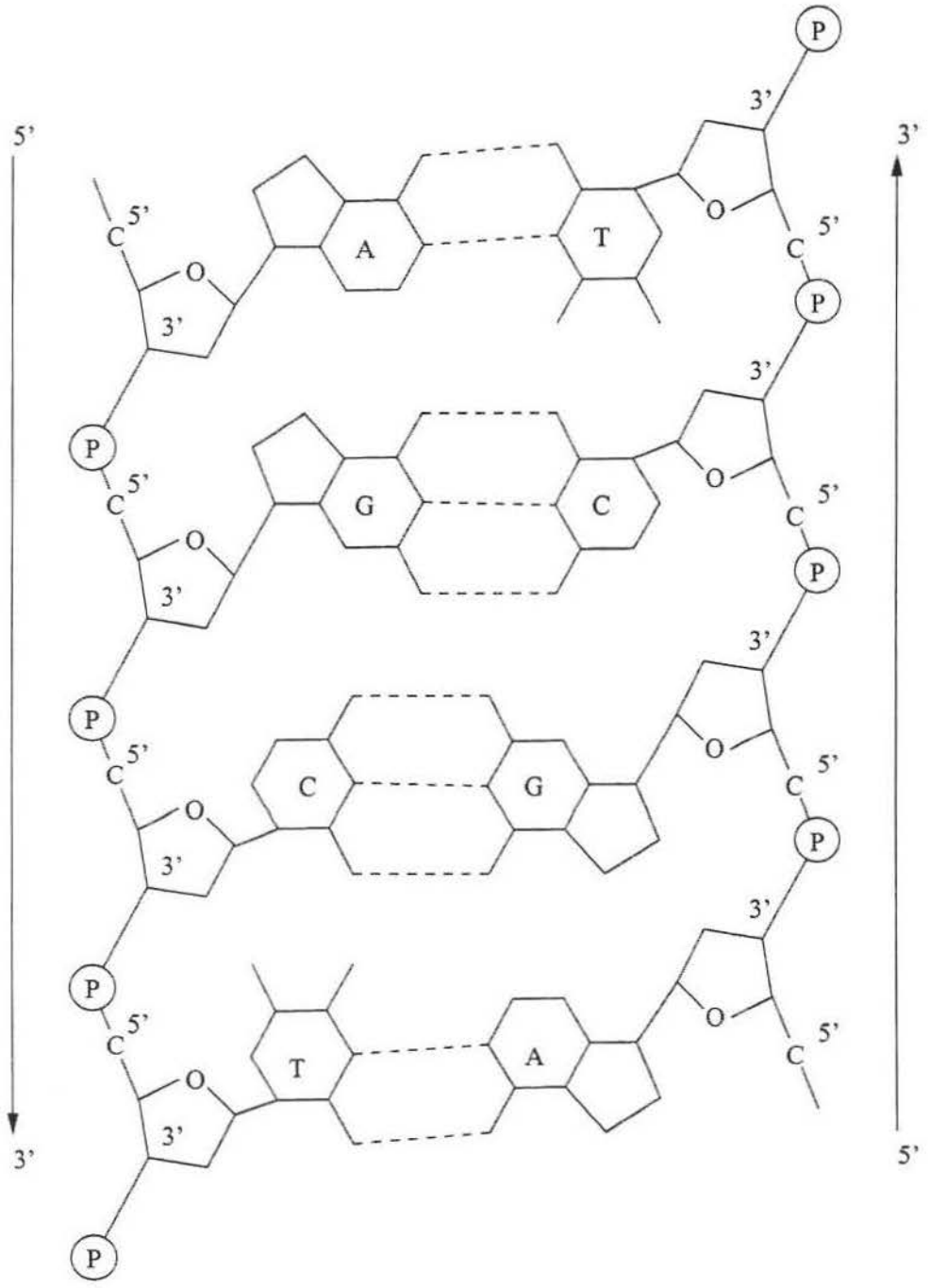

Figura 1.2: Esquema da estrutura molecular da fita dupla de DNA.

4. estes fragmentos selecionados são submetidos à clonagem em laboratório resultando em conjunto de *insertos*;

5. este conjunto de insertos é, então, seqüenciado diretamente em laboratório. As seqüências obtidas recebem o nome de *reads*.

No decorrer deste documento, iremos usar os termos *fragmentos*, *insertos*, *clones* e *reads* com o mesmo significado, embora designem coisas diferentes em cada fase do Shotgun.

A partir dos reads é que se começa a montagem de fragmentos propriamente dita. A seqüência resultante da montagem é o que chamamos de consenso, que se espera ser o mais próximo possível da seqüência original da molécula de DNA.

Para ser possível encontrar um consenso de boa qualidade, é necessário que, ao se quebrar a molécula original, obtenham-se fragmentos que cubram toda a molécula e que tenham boa sobreposição entre si. Além disso, para assegurar que problemas ocorridos na prática não prejudiquem a montagem, faz-se necessário obter um número suficiente de fragmentos, de forma que seja possível detectar a existência de tais problemas e tentar tratá-los.

Suponha os seguintes reads [16]: `ACCGT`, `CGTGC`, `TTAC`, `TACCGT`

A montagem procura a sobreposição entre eles:	`--ACCGT--`
	`----CGTGC`
	`TTAC-----`
	`-TACCGT--`
Obtém-se o consenso:	`TTACCGTGC`

Esse exemplo é o caso ideal, embora os tamanhos não sejam realistas. Os fragmentos cobrem inteiramente a molécula original, sobrepõem-se uns aos outros e estão livres de erros que podem ocorrer no processo de fragmentação e seqüenciamento.

A ausência de erros é pouco provável, por isso mostraremos a seguir os complicadores do processo de montagem de fragmentos [16].

Os fatores complicadores na montagem de fragmentos são:

1. *erros* nos reads introduzidos pelo sequenciador ou mesmo na duplicação do DNA, como fragmentos quiméricos ou contaminação com DNA dos vetores usados na duplicação do DNA estudado;

2. *regiões repetidas* ou *repetições* são regiões que aparecem duas ou mais vezes na molécula original, podendo causar ambigüidades nas soluções;

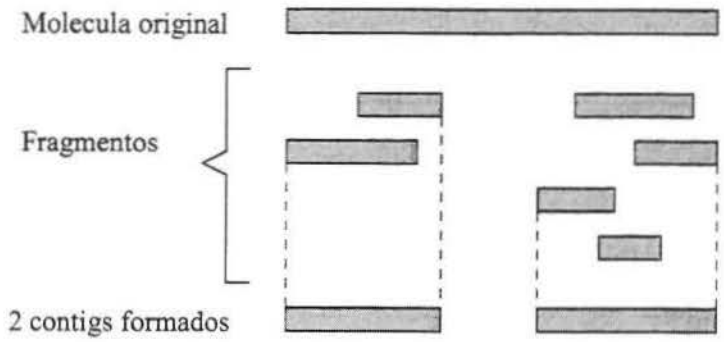

Figura 1.3: Problema da falta de cobertura.

3. *falta de cobertura* acontece quando os reads não são suficientes para montar o consenso único do DNA, deixando buracos (*gaps*) na montagem por falta de reads para cobrir aquela região, formando blocos que não se conectam, chamados *contigs*. Veja exemplo ilustrativo da figura 1.3.

1.3 O método OSS - Ordered Shotgun Sequencing

Conforme descrito na introdução desse capítulo, o método foi proposto por Chen e colegas em 1993 [4].

No método de seqüenciamento e montagem de fragmentos de DNA *Ordered Shotgun Sequencing*, os clones são amostrados aleatoriamente como no método Shotgun, mas aqui são maiores e terão inicialmente somente as pontas seqüenciadas.

A técnica consiste basicamente em algumas etapas que se repetem [13, 4, 20]:

1. encontrar o maior contig possível com base nas sobreposições dos fragmentos seqüenciados nas extremidades. Este "contig" é apenas a espinha dorsal da montagem, ou o *scaffold*, pois posicionamos alguns clones relativamente entre si, mas não temos a sua seqüência toda, só a distância aproximada entre as partes conhecidas;

2. seqüenciar a parte faltante do scaffold selecionado;

3. nesta fase, estarão disponíveis dados que ajudarão a eliminar alguns fragmentos que faziam parte da região seqüenciada, mas que não se sobrepuseram com nenhum dos outros na fase anterior. Por exemplo, na figura 1.4 o fragmento F seria eliminado nessa fase, já que ele faz parte da região seqüenciada;

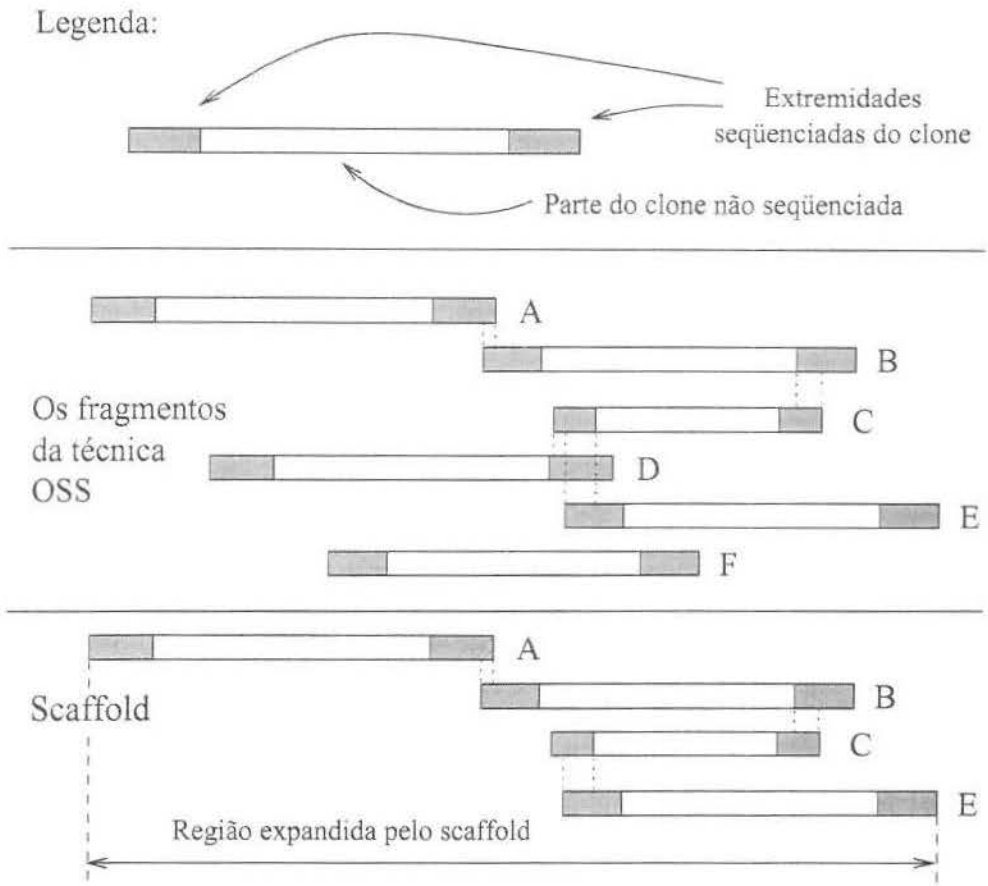

Figura 1.4: Exemplo de uma montagem OSS.

4. eliminar os fragmentos que entraram na composição do contig selecionado na etapa 1 e, no lugar, adicionar o "contig seqüenciado" ao conjunto de fragmentos a entrar no processo de montagem da próxima iteração;

5. processar novamente a partir do passo 1.

Esse processo se repete até que reste apenas um contig (nesse caso, chegamos ao consenso) ou até não encontrarmos mais nenhuma sobreposição para formar um novo contig. Veja figura 1.5 ilustrando o processo iterativo do método OSS.

O interessante é que, durante a montagem, nem todos os fragmentos são necessariamente utilizados ou seqüenciados para se descobrir o consenso. Qualquer buraco remanescente é completado por seqüenciamento direto ou outro método qualquer [13, 4].

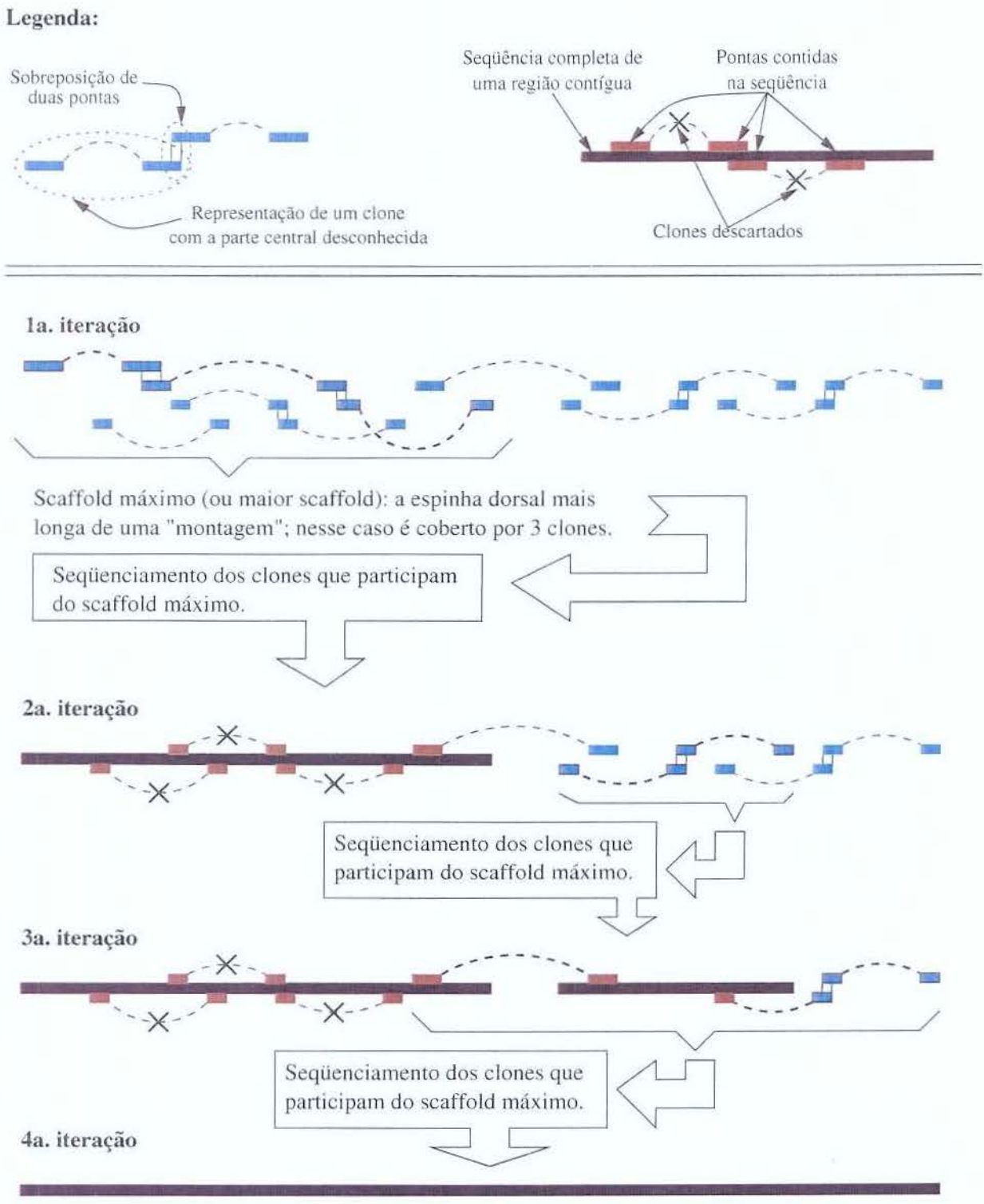

Figura 1.5: Exemplo ilustrativo da iteração OSS.

1.4 Modelagem dos Dados

Em Biologia Computacional, costuma-se trabalhar com *cadeias* sobre um alfabeto de quatro letras $\{\mathtt{A}, \mathtt{T}, \mathtt{C}, \mathtt{G}\}$, pois uma molécula de DNA pode ser vista como duas cadeias, s e $\overline{s}$, sobre aquele alfabeto, em que a segunda cadeia é o complemento reverso da primeira.

Dada uma cadeia s, o complemento reverso $\overline{s}$ será obtido pela simples mudança das ocorrências dos caracteres A, T, C e G pelos respectivos caracteres T, A, G e C e pela reversão da ordem da seqüência obtida. Veja exemplo a seguir.

cadeia s	tatcccggtgagcaccc
o complemento de s	atagggccactcgtggg
o complemento reverso $\overline{s}$	gggtgctcaccgggata

Assim, sempre que falarmos de cadeias, estaremos considerando-as sobre o alfabeto apresentado. Da mesma forma que qualquer letra com barra em cima representará o complemento reverso da cadeia representada pela letra. Por exemplo, $\overline{s}$ representa o complemento reverso de s.

Podemos visualizar cada extremidade dos clones gerados pelo método OSS como uma cadeia s sobre o alfabeto $\{\mathtt{A}, \mathtt{T}, \mathtt{C}, \mathtt{G}\}$, cujo tamanho pode variar (embora em nossos testes tenhamos usado tamanho fixo). Cada fragmento será representado por um par de cadeias, s e t, separado por uma distância d, correspondente ao tamanho da parte do fragmento desconhecida.

Devemos ter cuidado ao representar o clone a partir dos pares, pois a leitura das extremidades do fragmento do DNA se faz sempre de 5' $\rightarrow$ 3', o que significa que o par de cadeias que representa o clone não é da mesma fita. Portanto, para representar o clone, devemos ter, em uma das extremidades, o complemento reverso da cadeia. Para elucidar esta questão, observe o esquema da figura 1.6.

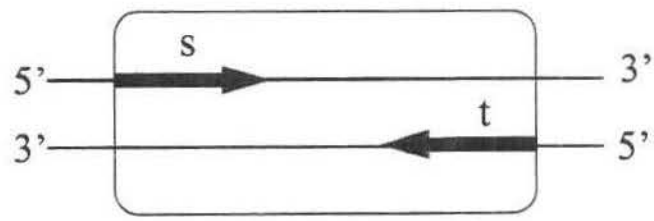

Figura 1.6: Esquema da leitura de um fragmento no método OSS.

Portanto, simplesmente dizer que s e t representam as extremidades lidas de um determinado clone não seria muito apropriado. O melhor seria dizer que o par s e $\overline{t}$ ou

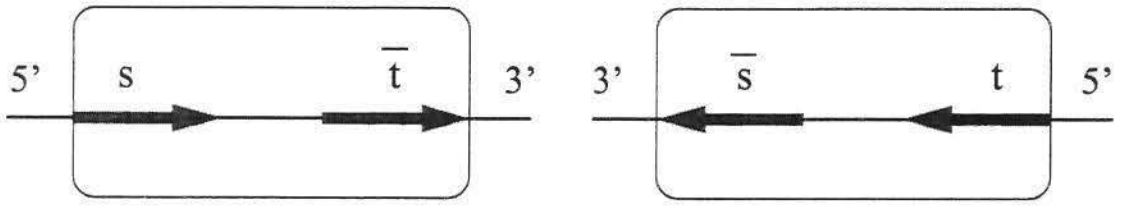

Figura 1.7: Ilustração da definição de clone como par de cadeias.

o par t e $\overline{s}$ representa o clone do método OSS, já que nunca sabemos de qual fita um fragmento foi lido. Baseado no esquema anterior, o que queremos representar pode ser melhor compreendido pela ilustração da figura 1.7.

Sobreposição

Dadas duas cadeias s e t, dizemos que s se sobrepõe a t quando os k caracteres finais de s são iguais aos k caracteres iniciais de t para um certo $k \geq 1$. Ou seja, se k-fim(s) representa a subcadeia final de s de tamanho k e k-ini(t) representa a subcadeia inicial de t de tamanho k, observamos sobreposição de s com t quando k-fim(s) = k-ini(t) [8]. Neste caso, podemos dizer também que s e t têm sobreposição k.

Exemplo: Seja $k = 5$,

cadeia x	k-ini(x)	k-fim(x)
s = acgtccggttca	acgtc	gttca
t = gttcaccgaacc	gttca	gaacc

Como k-fim(s) = k-ini(t), s se sobrepõe a t por 5 caracteres, portanto s e t possuem sobreposição 5. Para facilitar, podemos visualizar uma sobreposição como os biólogos costumam fazer:

```
s  acgtccggttca
t        gttcaccgaacc
```

No exemplo acima, há uma sobreposição de 5 caracteres de s com t.

Contudo, devemos ficar atentos quanto à diferença de se dizer "t se sobrepõe a s em k caracteres", pois neste caso nos referimos a k-fim(t)=k-ini(s) e não k-fim(s) = k-ini(t); ou seja, estaríamos falando erroneamente em:

```
s        acgtccggttca
t  gttcaccgaacc
```

No decorrer desse trabalho, poderemos usar também a palavra inglesa *overlap* para designar sobreposição.

1.5 Formalização do problema

Podemos resumir o problema da montagem de fragmentos OSS à procura do melhor scaffold a cada iteração. Computacionalmente, podemos vê-lo como um problema em grafos, modelado da seguinte maneira:

Sejam s_i, $\overline{t_i}$, com $i = 1..n$, os pares de cadeias que formam as duas extremidades dos nossos n clones e seja d_i o tamanho da parte central desconhecida do clone conforme mostra a figura 1.8.

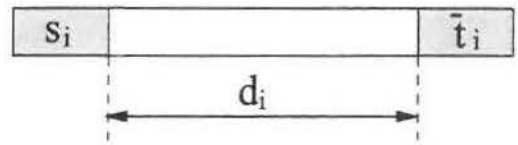

Figura 1.8: Representação gráfica de um clone seqüenciado nas extremidades.

Seja um grafo $G = (V, A)$ orientado, com peso associado às arestas definido como segue. Cada vértice $v \in V$ corresponde a uma extremidade dos clones. Portanto, os vértices de G são: $s_i, t_i, \overline{s_i}, \overline{t_i}$ com $1 \leq i \leq n$. Note-se que $|V| = 4n$.

Existe uma aresta $(u, v) \in A$ em duas situações:

1. quando há **sobreposição** entre as extremidades de dois clones. Vale lembrar que se existe sobreposição entre as extremidades u e v, então certamente haverá sobreposição entre seus complementos reversos, $\overline{v}$ e $\overline{u}$. Nesta situação, a aresta será conhecida como *aresta de sobreposição*;

2. quando há **pareação** entre duas extremidades, ou melhor, quando duas extremidades pertencem ao mesmo clone. Arestas deste tipo serão chamadas de *aresta de ligação*.

Existindo uma aresta $(u, v) \in A$, obrigatoriamente deve existir a aresta simétrica $(v, u) \in A$. A diferença entre elas está no seu peso associado.

O peso $p(u, v)$ associado à aresta $(u, v) \in A$ representa a distância entre u e v. Devemos esclarecer que aqui empregamos "distância entre u e v" como o número de caracteres que existem entre o primeiro caractere da cadeia u e o primeiro caractere da cadeia v.

Se a aresta (u, v) existe em virtude de uma sobreposição k, entre os fragmentos u e v, de tamanhos $|u|$ e $|v|$, então:

$$p(u, v) = |u| - k$$

$$p(v,u) = -|u| + k.$$

Se a aresta (u,v) existe por causa de uma pareação entre as pontas u e v, de tamanhos $|u|$ e $|v|$, separados por d_i bp desconhecidos, então:

$$\begin{aligned} p(u,v) &= |u| + d_i \\ p(v,u) &= -|u| - d_i. \end{aligned}$$

Esses conceitos podem ser melhor compreendidos pela figura a seguir (figura 1.9) :

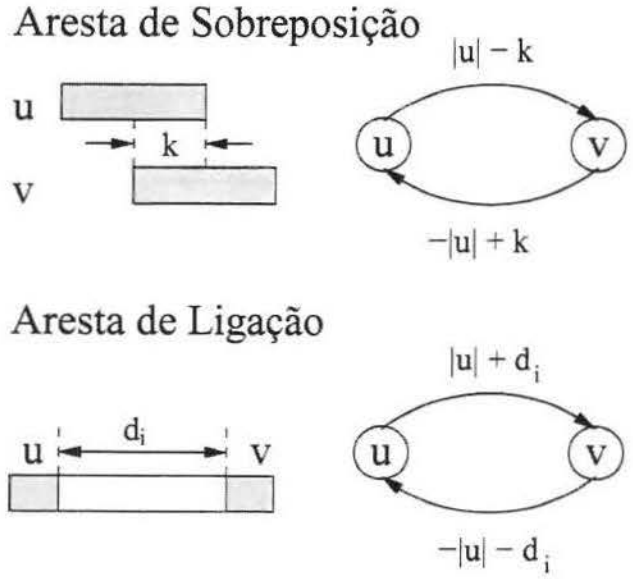

Figura 1.9: Esquema gráfico versus representação em grafo.

O grafo assim definido será chamado de *grafo de sobreposição OSS*, ou simplesmente, *grafo OSS* no decorrer de nosso trabalho.

Um *scaffold* é um caminho em G, onde cada clone aparece no máximo uma vez. Isto significa que se um certo clone foi incluído num caminho, seu complemento reverso não pode aparecer.

O *peso* de um caminho é a soma dos pesos das suas arestas. Em termos biológicos, se o caminho for um scaffold, o peso nos dá a distância relativa entre os extremos deste caminho.

A *extensão* de um caminho é o peso deste mais o tamanho do fragmento representado pelo último vértice desse caminho. Em termos de biologia, se o caminho for um scaffold, a extensão será o tamanho da região fisicamente compreendida entre os extremos do caminho. O nosso problema é encontrar o **scaffold de extensão máxima** que equivale

a encontrar um caminho de extensão máxima num grafo orientado com pesos positivos e negativos associados às arestas, podendo conter ciclos e com a restrição de não permitirmos que s e $\overline{s}$ estejam no mesmo caminho. Não estudamos profundamente este problema, porém o problema relacionado de encontrar o caminho mais longo em um grafo qualquer é NP-Completo [1, 9].

No caso ideal, depois de várias iterações, o grafo terá duas componentes conexas e haverá um caminho ligando os fragmentos em uma das componentes; por sua vez, a extensão da montagem nos dará o tamanho do consenso. A outra componente resultará no complemento reverso da montagem, pois estamos modelando o problema com orientação desconhecida e incluímos vértices que representam o complemento reverso dos clones.

Supomos, como caso ideal, a cobertura completa da molécula original e a ausência de repetições e erros. Para o caso do problema de falta de cobertura, mas sem repetições ou erros, o grafo terá mais do que duas componentes conexas. A nossa montagem resultará em metade de contigs de quantos forem as componentes conexas, pelo mesmo motivo anterior.

Os próximos capítulos descreverão os algoritmos propostos para tentar encontrar um scaffold que melhor expanda a montagem OSS, o ambiente de simulação usado para testá-los, os resultados desses testes e sua análise e as nossas conclusões.

Capítulo 2

Algoritmos Scaffold

Os algoritmos propostos neste capítulo trabalham com o grafo de sobreposição OSS definido no capítulo anterior.

Como vimos, num grafo de sobreposição OSS, ou grafo OSS, há dois tipos de arestas: *aresta de sobreposição* e *aresta de ligação*. No entanto, o grafo OSS pode ter vértices representando fragmentos que não são pontas de um clone, mas um clone todo seqüenciado. Neste caso, a aresta de ligação é da forma (u, u) e tem peso igual a zero.

Lembramos que o peso associado às arestas do grafo corresponde ao número de bases existentes entre o começo de uma ponta e o começo da outra. Além disso, para cada aresta do grafo, existe uma outra aresta com peso de valor oposto e orientação contrária. Por fim, associado aos vértices, há o tamanho da seqüência representada por cada um deles.

Apresentaremos dois algoritmos para tentar encontrar um scaffold que melhor expanda uma montagem de fragmentos. O primeiro é um ***algoritmo guloso***, a ser chamado de algoritmo 1, e o segundo, a ***maior componente conexa OSS***, que será referenciado como algoritmo 2.

2.1 Algoritmo 1 - O Guloso

O algoritmo guloso recebe o grafo de sobreposição OSS e age da seguinte forma. A partir do maior clone que o algoritmo guloso encontra no grafo, o próximo clone escolhido para compor o scaffold será aquele que se sobrepõe ao primeiro e que mais estende a montagem.

O clone seguinte será aquele que prolonga e se liga à extremidade do scaffold formado anteriormente e assim por diante, até que não haja clones desmarcados que estendam o scaffold. O algoritmo primeiro prolonga à direita e depois à esquerda.

Um clone é representado no grafo como uma aresta de ligação e ele só se incorpora no scaffold se o seu complemento reverso ou o próprio clone não foram utilizados ainda na

composição do scaffold.

Na figura 2.1, temos uma representação do algoritmo e na figura 2.2 o seu código.

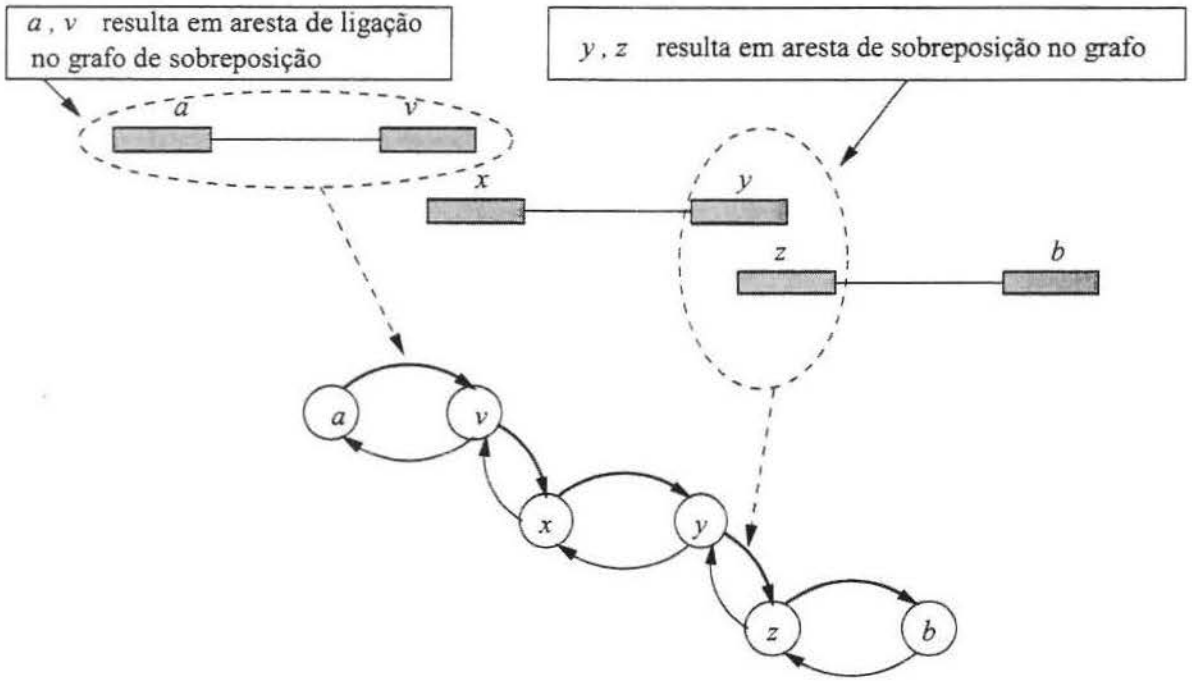

Figura 2.1: Representação gráfica do algoritmo guloso

No algoritmo mostrado nas figuras 2.2 e 2.5, são usadas as rotinas MARQUE(C) para marcar todos os vértices de C, bem como os correspondentes ao complemento reverso deles, e MARCADO(u) para saber se u é um vértice marcado. Com as marcas, o algoritmo evita escolher os mesmos clones, ou o complemento reverso de clones que já compõem o scaffold em formação.

As rotinas para estender o scaffold à esquerda e à direita são bastante parecidas, por isso apresentamos apenas o ESTENDEY (figura 2.5), que estende à direita, e ressaltaremos a diferença entre eles no decorrer do texto. Antes, vamos entender como o algoritmo tenta estender um scaffold.

Há duas possibilidades de se estender um scaffold à direita, conforme mostra a figura 2.3. Na rotina ESTENDEY, a possibilidade **A**, mostrada na figura em questão, é testada na linha 8 da rotina (figura 2.5) e a hipótese **B**, na linha 14. Usando a mesma idéia, o ESTENDEX irá estender à esquerda conforme duas possibilidades mostradas na figura 2.4.

Um detalhe importante entre as duas rotinas é que em ESTENDEY, o ponto inicial para alongar o scaffold é a partir do início da ponta y, pois o peso das arestas que saem deste vértice é o número de bases entre o início de y e o início da outra ponta, conforme definição de pesos do grafo OSS do capítulo 1 e mostrado na figura 1.9. Assim, a rotina

Entrada: G é o grafo de sobreposição OSS.
Saída: um scaffold C.
SCAFFOLD1(G)
1 Todos os vértices começam desmarcados;
2 **Seja** (x, y) a maior aresta de ligação do grafo G;
3 $C \leftarrow \{(x, y)\}$;
4 MARQUE(C);
5 $extensão\_total \leftarrow peso(x, y) + tam(y)$;
6 $C \leftarrow EstendeY(C, y)$;
7 $C \leftarrow EstendeX(C, x)$;
8 **return** C;

Figura 2.2: Corpo principal do algoritmo guloso: SCAFFOLD1.

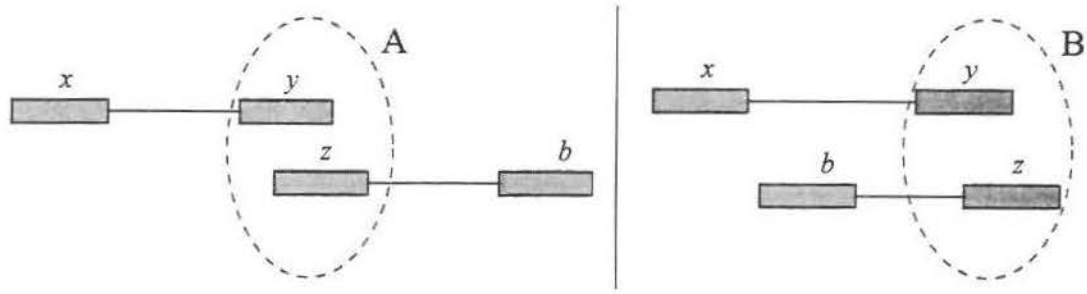

Figura 2.3: Possibilidades para estender o scaffold à direita.

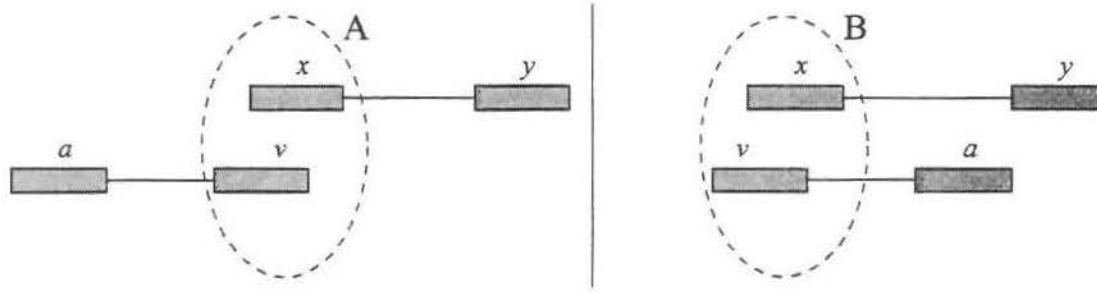

Figura 2.4: Possibilidades para estender o scaffold à esquerda.

Entrada: Caminho encontrado até o momento e vértice y a estender.
ESTENDEY(C, y)

```
 1  deu_pra_estender ← true;
 2  while deu_pra_estender
 3  do deu_pra_estender ← false;
 4     extensão ← tam(y);
 5     for each z tal que (y, z) é aresta de sobreposição
 6     do if not MARCADO(z)
 7           then Determine b tal que (z, b) seja uma aresta de ligação;
 8                proposta_ext ← peso(y, z) + peso(z, b) + tam(b);
 9                if extensão < proposta_ext
10                  then deu_pra_estender ← true;
11                       novo_y ← b;
12                       pedaco_caminho ← {(y, z), (z, b)};
13                       extensão ← proposta_ext;
14                proposta_ext ← peso(y, z) + tam(z);
15                if extensão < proposta_ext
16                  then deu_pra_estender ← true;
17                       novo_y ← z;
18                       extensão ← proposta_ext;
19                       pedaco_caminho ← {(y, z)};
20     if deu_pra_estender
21       then MARQUE(pedaco_caminho)
22            C ← C + pedaco_caminho;
23            extensão_total ← extensão_total + extensão − tam(y);
24            y ← novo_y;
25  return C
```

Figura 2.5: Subrotina do SCAFFOLD1: ESTENDEY.

só consegue estender o scaffold se existir uma proposta de extensão maior que o tamanho da ponta y. Já para ESTENDEX, o ponto de partida é zero, pois o peso das arestas que chegam ao vértice x representa a quantidade de bases até o início da ponta x.

Como a extensão total vai até o final da ponta y e a proposta de extensão parte do início de y, ao computar a nova extensão, resultante da soma dos dois, o tamanho de y é subtraído da nova extensão (linha 23 da figura 2.5). Com a ponta x não teremos este problema, pois o tamanho de x não é computado na proposta de extensão.

As principais diferenças estão apontadas na tabela 2.1, onde indicamos o número da linha em ESTENDEY e a mudança necessária na rotina ESTENDEX.

A vantagem do algoritmo é que usamos uma idéia simples, com as desvantagens conhe-

Nº da Linha	Expressão substituta em ESTENDEX()
4	$extensão = 0$
8	$proposta\_ext = peso(v, x) + peso(a, v);$
14	$proposta\_ext = peso(v, x);$
22	$C = pedaco\_caminho + C;$
23	$extensão\_total = extensão\_total + extensão;$

Tabela 2.1: Tabela das principais diferenças entre ESTENDEX() e ESTENDEY()

cidas: ser guloso, ou melhor, escolher o maior clone em todo passo não necessariamente resultará no maior scaffold, pois um clone menor poderia ser a conexão de um clone maior mais pra frente.

Para um genoma circular, o scaffold encontrado pode representar mais de uma volta no genoma. Em outras palavras, o scaffold pode conter clones que cubram a mesma região do genoma, pois o algoritmo não testa se o círculo do genoma fechou, apenas vai estendendo o scaffold até que não haja mais nenhum clone desmarcado que possa fazê-lo. Isso pode resultar em sequenciamento exagerado, como veremos no capítulo 4.

Em caso de empate no tamanho do clone a ser escolhido, o primeiro da lista é o escolhido para compor o scaffold, neste caso, dependendo da ordem de entrada dos clones, o scaffold encontrado pode ser diferente. Portanto, o algoritmo pode ser sensível à ordem dos dados de entrada.

A complexidade para o nosso algoritmo guloso é $O(m+n)$. A razão para isso é que os vértices são marcados quando usados, e vértices já marcados ou arestas que saem deles não são visitados novamente. Assim, cada vértice e cada aresta é processada um número constante de vezes.

2.2 Algoritmo 2 - A Maior Componente Conexa OSS

Este algoritmo usa a idéia de conjuntos disjuntos [6] com a intenção de juntar o maior número de clones num conjunto e procurar o melhor scaffold nele. Podemos dizer que este algoritmo baseia-se na procura das componentes conexas do grafo.

Inicialmente, cada clone é um conjunto de duas pontas, portanto, no primeiro instante teremos conjuntos que contêm uma aresta, ou um par de vértices. Com base nas arestas de sobreposição, ordenadas pelo peso, os conjuntos são unidos dois a dois.

A união de dois conjuntos deve obedecer à restrição de que os vértices de um conjunto não podem ser o complemento reverso de algum vértice do outro conjunto e vice-versa. Em outras palavras, se o complemento reverso de algum vértice do conjunto A estiver em B, então A e B não devem ser unidos. Desta maneira, garantimos que os conjuntos formados no final apenas contenham vértices e, conseqüentemente, arestas válidas na

composição do scaffold.

O algoritmo pode ser descrito pelo SCAFFOLD2, mostrado na figura 2.6.

Entrada: G é o grafo de sobreposição OSS.
Saída: Um scaffold C.
SCAFFOLD2(G)

```
1  S ← conjunto de arestas de sobreposição do grafo G,
2   ordenados por tamanho da sobreposição;
3  C ← 0;
4  for each (x, y) tal que (x, y) é aresta de ligação
5  do Crie na coleção C um conjunto com os vértices x e y;
6  while S ≠ ∅
7  do (u, v) ← REMOVE_MAIOR_ARESTA(S);
8      L_u ← ConjDe(u);
9      L_v ← ConjDe(v);
10     if L_u' ∩ L_v = ∅
11        then C ← (C ∪ {L_u ∪ L_v}) − {L_u, L_v};
12 L ← maior conjunto de C;
13 C ← o "melhor" caminho encontrado no subgrafo G(L),
14 induzido pelos vértices do conjunto L;
15 return C;
```

Figura 2.6: Algoritmo SCAFFOLD2.

Esse algoritmo aposta que quanto maior o número de fragmentos agrupados, maior a chance do scaffold encontrado estender mais a montagem. Contudo, esta suposição pode ser falsa, como veremos no capítulo 4. Desta forma, a última etapa do algoritmo é procurar o scaffold C no maior agrupamento do grafo.

O passo mais crítico neste algoritmo é o último passo, que está na linha 12 do algoritmo: como definir o "melhor" caminho? Considerando que um scaffold é um conjunto de clones orientados, posicionados relativamente uns aos outros, uma idéia seria construir um **mapa** dos clones, ou seja, uma representação pictorial dos fragmentos, cada um ocupando seu lugar fisicamente correto.

O lugar dos clones no mapa é determinado pelas coordenadas de cada um num eixo linear. A atribuição das coordenadas aos clones pode ser feita por uma busca. Escolhemos um fragmento qualquer e lhe conferimos a coordenada 0 (zero). A partir daí, é feita uma busca, por exemplo, em profundidade, dando a cada novo fragmento visitado a coordenada resultante da posição do fragmento de onde veio, mais o peso da aresta navegada.

Dois problemas podem aparecer:

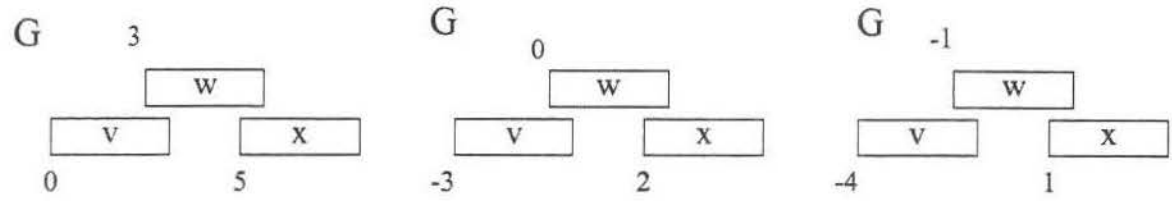

Figura 2.7: Mapeamento equivalente dos vértices v, w, x.

Problema(1) : partindo de diferentes vértices iniciais não obtemos o mesmo resultado;

Problema(2) : ao re-visitar um vértice, a coordenada já atribuída não é igual à coordenada calculada neste momento.

Antes de tudo, vamos definir o significado de "mesmo resultado". O resultado desejado para construção do mapa é um vetor de coordenadas *coord* para cada fragmento. Um vetor terá dado o mesmo resultado que outro, não quando dois vetores são idênticos, mas sim quando são *equivalentes*, como definiremos a seguir.

Dois vetores de coordenadores *coordA* e *coordB* sobre o mesmo grafo G são *equivalentes* quando existe uma constante c tal que:

$$coordB(v) - coordA(v) = c$$

para todo $v \in$ grafo G.

Para exemplificar, seja os vetores de coordenadas para os vértices do mesmo grafo v, w, x, respectivamente:

$$\begin{aligned} coordA &= (0, 3, 5) \\ coordB &= (-3, 0, 2) \\ coordC &= (-4, -1, 1). \end{aligned}$$

As atribuições de coordenadas variam, mas são equivalentes, pois resultam no mesmo mapa, conforme ilustra a figura 2.7.

Os grafos que apresentam problemas do tipo 1 são os mesmos que apresentam o problema do tipo 2, que por sua vez são aqueles que possuem ciclos de peso total não nulo. Nestes grafos, qualquer vértice inicial e ordem nas listas de adjacências resultará num sistema de coordenadas não equivalentes. Chamaremos estes grafos de *não mapeáveis*.

Teorema 2.2.1 *Seja G um grafo orientado com pesos nas arestas e tal que para cada aresta (u, v) existe a aresta (v, u) com peso oposto ao peso de (u, v). Então G apresenta o problema (1), se, e somente se, apresenta o problema (2) e se, e somente se, possui um ciclo orientado de peso não nulo.*

Prova: Para cada aresta (u, v) existe a aresta (v, u) e

$$p(v, u) = -p(u, v)$$

Isto significa que para cada caminho $P = u_1 u_2 \ldots u_k$ de peso total

$$p(P) = p(u_1, u_2) + \ldots + p(u_{k-1}, u_k)$$

temos um caminho inverso $P' = u_k u_{k-1} \ldots u_1$ obedecendo

$$p(P') = -p(P)$$

Agora suponha que o grafo G apresente o problema (2). Então, utilizando diferentes ordens nas listas de adjacências, conseguimos como resultados a lista de coordenadas $coordA$ e $coordB$ não equivalentes, mesmo começando no mesmo vértice v, ou seja, $coordA(v) = 0$ e $coordB(v) = 0$, mas existe um vértice w em G tal que $coordA(w) \neq coordB(w)$.

Como $coordA$ é construído a partir de uma busca em profundidade, temos que existe um caminho P de v a w com peso total $p(P) = coordA(w)$. Analogamente, existe um caminho Q de v a w com $p(Q) = coordB(w)$. Mas compondo P com o inverso Q' de Q, temos $p(PQ') \neq 0$ e PQ' é um passeio fechado. Ora, se há um passeio fechado de peso diferente de 0 é fácil concluir que há um ciclo de peso não nulo (formado por parte do passeio, sem repetições de vértices). Logo, se há problema (2) existe um ciclo de peso não nulo.

Por outro lado, se há um ciclo de peso não nulo, por exemplo, $C = u_1 u_2 \ldots u_k u_1$, então veremos que o grafo tem o problema (2). Basta notar que os caminhos $P = u_1 u_2$ e $Q = u_1 u_k u_{k-1} \ldots u_3 u_2$, ambos de u_1 a u_2, têm pesos distintos (senão o ciclo teria peso nulo) e podemos definir ordens nas listas de adjacência de G para que, iniciado em u_1, uma ordem A atribua para $coordA(u_2) = p(P)$ e a outra ordem B atribua $coordB(u_2) = p(Q)$. Como $p(P) \neq p(Q)$, aparece aí o problema (2).

Isto mostra que o problema (2) é equivalente à existência de ciclos não nulos. Agora vejamos o problema (1).

Se ocorre o problema (1), é porque iniciando-se em v e w obtém-se $coordA$ e $coordB$ não equivalentes. Sabemos que $coordA(v) = 0 = coordB(w)$. Se são não equivalentes, existe um vértice x tal que $coordB(x) - coordA(x) \neq coordB(v)$. Mas cada valor de $coord$ é o peso de um caminho, logo:

$$p(P) - p(Q) \neq p(R),$$

onde P é um caminho de w a x, Q é um caminho de v a x e R é um caminho de w a v. Isto significa que o passeio RQP', onde P' é o inverso de P, tem peso total não nulo e é fechado, ou seja, temos um ciclo de peso não nulo.

Reciprocamente, dado um ciclo de peso não nulo $C = u_1u_2 \ldots u_ku_1$ podemos fazer as listas de adjacência de tal modo que, começando em u_1 chegamos a $coordA(u_2) = p(u_1, u_2)$ e, começando em u_2 temos $coordB(u_1) = p(u_2 \ldots u_ku_1)$. Como $p(u_1, u_2) \neq -p(u_2 \ldots u_ku_1)$, não são equivalentes $coordA$ e $coordB$. □

O algoritmo proposto para atribuir coordenadas aos fragmentos também detectará se no grafo há ciclo não nulo, portanto não mapeável, como especificado no algoritmo MAPEÁVEL, mostrado na figura 2.8.

Entrada: Grafo de sobreposição OSS.
Saída: *true*, se grafo mapeável; *false*, não mapeável.

```
MAPEÁVEL(G)
1  ok ← true;
2  for each v in V(G)
3  do coord[v] ← NIL;
4  Escolha v ∈ V(G);
5  coord[v] ← 0;
6  DFS(v);
7  return ok;
```

Entrada: vértice visitado

```
DFS(w)
1  for each x in Adj(w)
2  do if coord[x] = NIL
3        then coord[x] ← coord[w] + desloca(w, x);
4             DFS(x);
5        else if coord[x] ≠ coord[w] + desloca(w, x)
6                then ok ← false ;
```

Figura 2.8: Algoritmo MAPEÁVEL e sua subrotina DFS.

A questão de grafo não mapeável será tratada mais tarde. Agora mostraremos como encontrar o caminho que representa o melhor scaffold num grafo mapeável.

A partir do mapa, obtém-se os fragmentos que estão nas duas extremidades do mapa. Eles representam as pontas inicial e final do scaffold procurado. Portanto, com o mapa, identificamos os vértices inicial e final, s e t, do caminho do grafo de sobreposição OSS representando o melhor scaffold.

Os fragmentos das extremidades são aqueles com o menor e a maior coordenadas no mapa. O de menor coordenada é aquele que começa na coordenada de valor mais baixo no mapa, enquanto o de maior coordenada é aquele que termina na coordenada de maior valor. Em outras palavras, quando nos referimos ao fragmento r de maior coordenada,

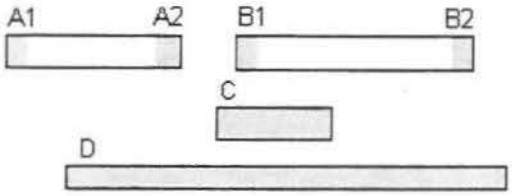

Figura 2.9: Escolha do fragmento de maior coordenada: fragmento D.

estamos falando daquele que tem o maior valor M,

$$M = coord(r) + tamanho(r),$$

onde $coord(r)$ é a coordenada inicial do fragmento r e $tamanho(r)$ é o seu tamanho. Na figura 2.9 está desenhada a situação que nos fez decidir ver o fragmento de maior coordenada como acabamos de explicar. Tomando o exemplo da figura, a lista ordenada de fragmentos de menores coordenadas seria $(A1, D, A2, C, B1, B2)$ e a de maiores coordenadas conforme nossa definição seria $(D, B2, C, B1, A2, A1)$.

O melhor scaffold, além de expandir bem a montagem, requer menos trabalho de seqüenciamento. Para isso, o número de partes desconhecidas que entram neste scaffold deve ser o menor possível e elas são representadas pelas arestas de ligação. O que se deseja é encontrar um caminho entre os vértices s e t no grafo de sobreposição OSS que use o menor número de arestas de ligação.

Podemos aplicar o algoritmo de Dijkstra [6] para procurar o caminho de menor peso do vértice s até t, colocando custo 1 nas arestas de ligação e custo 0 nas arestas de sobreposição. O custo total do caminho dará a quantidade de clones usados e a serem seqüenciados para a próxima iteração.

Voltemos agora para a discussão dos problemas do grafo não mapeável, ou seja, os que têm ciclos de peso total não nulo.

Então existem dois tipos de grafos: aqueles com todos os ciclos de peso nulo, nos quais qualquer vértice inicial e qualquer ordem nas listas de adjacência vai dar um sistema de coordenadas equivalentes, resultando num grafo mapeável; e aqueles onde há ciclos de peso não nulo, onde não há sistema de coordenadas compatível com todas as arestas.

Dois problemas surgem aqui:

- Como reconhecer os grafos mapeáveis?
- O que fazer se uma coleção de fragmentos produzir um grafo não mapeável?

Reconhecer grafos mapeáveis, como visto no algoritmo para atribuição de coordenadas, pode ser feito com o algoritmo MAPEÁVEL apresentado na página 25.

A questão quanto à atitude a ser tomada se o grafo OSS não for mapeável é mais complicada e não será tratada com profundidade nesta dissertação. As hipóteses para explicar o fato de ele não ser mapeável são:

1. genoma circular;
2. pequenos erros de deslocamento;
3. arestas falsas.

Um genoma circular não significa realmente que o grafo não seja mapeável, mas apenas que o mapa será circular também. Para resolver isso, podemos substituir a condição do algoritmo MAPEÁVEL (figura 2.8),

$$coord(x) = coord(w) + desloca(w, x)$$

que deve valer para todas as arestas, por uma condição específica para genomas circulares:

$$coord(x) \equiv coord(w) + desloca(w, x) \pmod{g}$$

onde g é o tamanho do genoma circular.

O mesmo algoritmo, com esta condição modificada, servirá aos nossos propósitos. Um problema aqui é que às vezes não conhecemos g exatamente, mas apenas aproximadamente. Neste caso, as igualdades devem ser substituída por desigualdades aproximadas, por exemplo:

$$| coord(x) - coord(w) - desloca(w, x) - g | \leq \varepsilon$$

ou

$$| coord(x) - coord(w) - desloca(w, x) + g | \leq \varepsilon,$$

onde ε é o erro máximo esperado no valor de g, ou o normal, sem erro

$$| coord(x) - coord(w) - desloca(w, x) | = 0$$

Já os pequenos erros de deslocamento, relativos a erros de inserção e remoção de bases em pontas, em geral isolados, fazem com que tenhamos que substituir a condição

$$coord(x) = coord(w) + desloca(w, x)$$

por uma mais tolerante:

$$| coord(x) - coord(w) - desloca(w, x) | \leq \varepsilon_1,$$

onde ε_1 é uma estimativa do máximo erro tolerável acumulado devido ao erros de deslocamento.

As arestas falsas ocorrem quando a sobreposição entre dois fragmentos é falsa, provocada por partes repetidas do genoma, ou por fragmentos quiméricos.

Entrada: Grafo de sobreposição OSS.
Saída: tamanho do genoma, se grafo mapeável; não mapeável, se retornou algo pequeno.
MAPEÁVEL2(G)
1 $len \leftarrow 0$;
2 **for each** v **in** $V(G)$
3 **do** $coord[v] \leftarrow$ NIL;
4 Escolha $v \in V(G)$;
5 $coord[v] \leftarrow 0$;
6 DFS(v);
7 **return** len;

Entrada: vértice visitado
DFS2(w)
1 **for each** x **in** $Adj(w)$
2 **do if** $coord[x] =$ NIL
3 **then** $coord[x] \leftarrow coord[w] + desloca(w, x)$;
4 DFS2(x);
5 **else** $novacoord \leftarrow coord[w] + desloca(w, x)$;
6 $len \leftarrow$ MDC(len, ABS($coord[x] - novacoord$));

Figura 2.10: Algoritmo MAPEÁVEL2 e sua subrotina DFS2.

2.2.1 Adaptações para o caso do genoma circular

Em nosso trabalho, lidamos com genomas circulares, por isso, foram necessárias algumas modificações no algoritmo para se adequar a essa característica do genoma. Como visto anteriormente, uma das mudanças é o cálculo das coordenadas no algoritmo MAPEÁVEL, que inclui o $\pmod{g}$, sendo g o tamanho do genoma.

A questão é como descobrir o tamanho do genoma dado o grafo de sobreposição OSS. Para dados artificiais, sem erro e um genoma sem buraco, poderíamos modificar o algoritmo MAPEÁVEL para que, a medida que se atribui as coordenadas aos vértices calcule-se o tamanho do genoma. Caso este tamanho seja pequeno, menor que o tamanho do maior clone, por exemplo, então o grafo não é mapeável. Do contrário, o grafo é mapeável. A descrição do algoritmo está na figura 2.10 sob o nome MAPEÁVEL2.

Um ponto digno de nota é o uso do conceito de clones com maior e menor coordenadas para identificar os clones que estão nas duas extremidades de um mapa. Para algo circular, isso soa estranho, já que em um círculo não se distingue o começo e nem o fim, portanto, não tem extremidades.

Apesar disso, iremos empregar o termo maior e menor coordenada no caso circular, pois cada clone receberá sempre uma coordenada, independentemente do tipo de genoma.

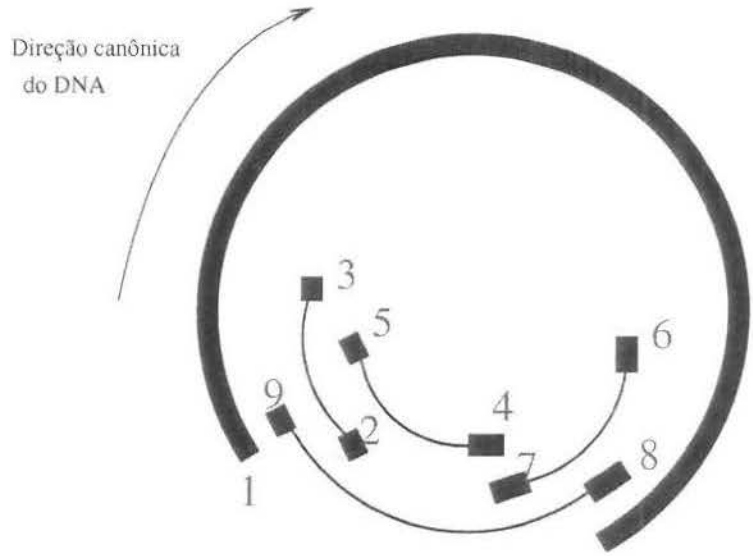

Figura 2.11: O mapa de distribuição dos fragmentos de um genoma circular.

Neste caso, nem sempre as menores e maiores coordenadas serão de clones mais extremos do mapa de distribuição de clones, mas tem uma boa chance de sê-lo.

Um outro problema que surge para o caso do genoma circular é encontrar o caminho que queremos, dado os dois vértices, s e t, representando os clones que estariam nas extremidades do mapa de distribuição. Por ser circular, pode haver mais de um caminho ligando s e t. O caminho que buscamos é aquele que cubra a maior região do genoma. Para ilustrar este problema, observe a figura 2.11, representando um genoma circular coberto por clones (pontas seqüenciadas) e um pedaço da região do genoma seqüenciado. Cada fragmento é associado a um vértice cujo rótulo é o número atribuído a cada uma das pontas de clones e à seqüência maior.

Na situação desenhada na figura 2.11, digamos que a menor coordenada identificada pelo algoritmo MAPEÁVEL é o vértice 2 e o maior é o 4. Para o algoritmo de Dijkstra, o menor caminho partindo do vértice 2 e chegando no 4, resultará em $\{2, 3, 1, 5, 4\}$. Contudo, o scaffold que nos interessaria seria aquele dado pelo caminho representado pela seqüência $\{2, 3, 1, 6, 7, 4\}$. Para tentar resolver esse problema, tiramos algumas arestas do grafo H, que será passado para o algoritmo de Dijkstra, para procurar o menor caminho entre s e t. A idéia é tirar, com base nas coordenadas dos clones, as arestas que fechariam o genoma circular, tentando deixá-lo linear. Uma aresta (u, v) é retirada do grafo H se ocorrer a condição (1) ou (2) a seguir.

$$coord[u] > coord[v] \text{ e } p(u, v) > 0 \quad (1)$$
$$coord[u] < coord[v] \text{ e } p(u, v) < 0 \quad (2)$$

Com a condição (2) pretendemos retirar as arestas simétricas às arestas que obedecem

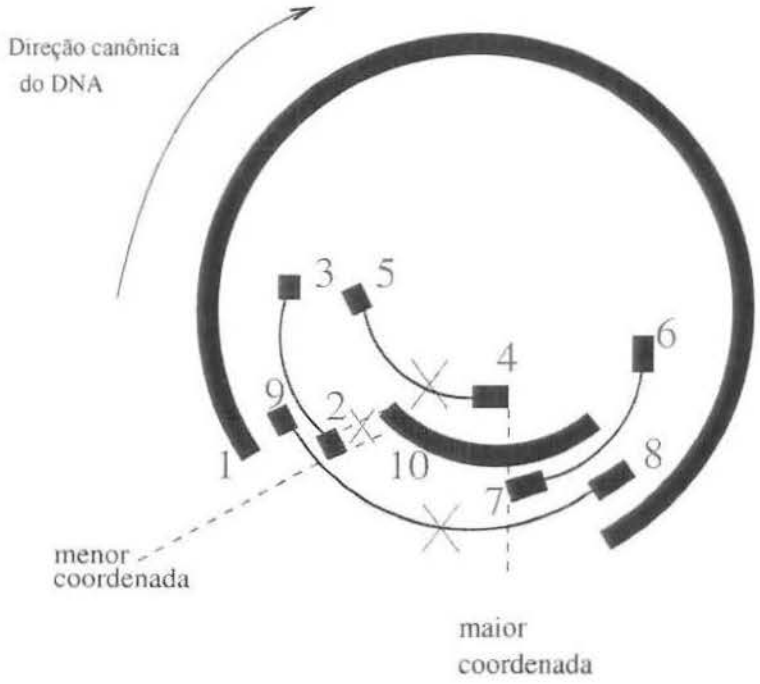

Figura 2.12: Ligações eliminadas (marcado com 'X') para tratar o caso do genoma circular.

à condição (1). Lembramos que num grafo OSS, a cada aresta (u, v) com peso $p(u, v)$, sempre existirá uma aresta simétrica (v, u) com peso $p(v, u) = -p(u, v)$, conforme definimos na seção 1.5.

Generalizando as condições anteriores, temos que uma aresta (u, v) é retirada do grafo H se

$$(coord[v] - coord[u]) * p(u, v) < 0.$$

Voltando ao exemplo anterior, as arestas eliminadas seriam as que permitiriam atravessar da região de maior coordenada, representada pelo vértice 4, para a região de menor coordenada, delimitada pelo vértice 2, conforme mostrada pela figura 2.12. No grafo OSS derivado da situação da figura, serão eliminadas as arestas de ligação $(4, 5)$ e $(8, 9)$, bem como as respectivas arestas simétricas $(5, 4)$ e $(9, 8)$ e as de sobreposição $(10, 2)$ e $(2, 10)$.

Teoricamente, esta solução evitaria o fechamento do genoma, mas na prática, na maioria das vezes, a montagem OSS atinge o fechamento completo do genoma ou, quando isso não acontece, está muito perto disso. No capítulo 4, mostraremos os resultados de vários testes feitos e analisaremos os casos em que a montagem OSS não conseguiu atingir 100% de cobertura do genoma.

A complexidade do algoritmo é $O(mn)$, sendo m o número de arestas e n a quantidade de vértices do grafo de sobreposição OSS, devido à verificação de intersecção entre dois conjuntos antes de juntá-los (figura 2.6, linha 9 do algoritmo).

Capítulo 3

O Ambiente de Simulação

Com o intuito principal de testar os algoritmos propostos para a montagem de fragmentos OSS, foi idealizado um ambiente de simulação OSS.

Os módulos do ambiente são para gerar o conjunto de dados, desenhar a distribuição dos fragmentos, encontrar a sobreposição entre eles, gerar o grafo OSS, encontrar o scaffold e os clones a serem seqüenciados, desenhar o mapa dos dados da iteração corrente e introduzir novos dados no conjunto a ser usado na próxima iteração.

Estes módulos podem ser divididos essencialmente como peças de dois grupos: o gerador de dados e o gerenciador de iterações e dados. O ambiente será descrito módulo a módulo a seguir e a visão geral do ambiente está esquematizada na figura 3.1. Foram usadas na implementação principalmente as linguagens C/C++ (compilador gcc/g++ 2.95.1) e Perl (versão 5.005_03). Os algoritmos para procura de scaffold foram implementados em C/C++ por estarmos interessados em performance da implementação, já que boa parte do processamento do ambiente estaria nesses algoritmos. Nos outros módulos e sub-módulos foi usada Perl, pois a tarefa deles era primordialmente de processamento de arquivos e esta era a melhor linguagem para isso. Entre programas, scripts do ambiente de simulação e scripts de apoio para disparar os testes e auxiliar na tabulação dos resultados, totalizamos cerca de 9500 linhas de código, contando aí linhas de comentários e de código propriamente dito.

3.1 Gerador de Dados

O gerador de dados é o módulo que cria os dados artificiais para serem usados em nossos testes controlados, isto é, os dados são livres de complicações comuns de uma montagem de fragmentos de DNA, como erros de leitura e regiões repetidas, descritos no capítulo 1, página 7. Uma visão geral do gerador de dados, com suas rotinas de processamento e os respectivos arquivos usados e/ou gerados, está esquematizada na figura 3.2, enquanto os

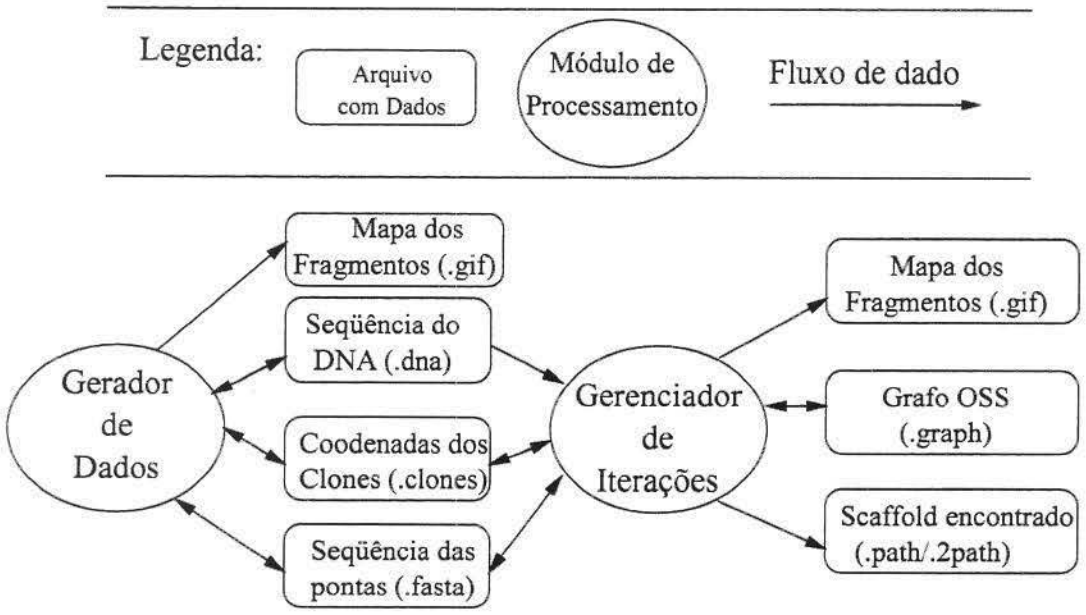

Figura 3.1: Esquema geral do ambiente de simulação OSS.

passos e os arquivos criados nesse módulo são enumerados e descritos a seguir.

1. Recebimento de parâmetros:
   - tamanho do DNA;
   - tamanho dos clones;
   - número de clones a serem usados;
   - tamanho das pontas dos clones;
   - nome a ser dado ao conjunto de dados e aos arquivos gerados.
2. Geração do DNA e seu complemento reverso, cada um em um arquivo. Um tem a extensão "*.dna*" e o outro, "*.dna-rev*".
3. Quebra aleatória para criação dos clones, resultando num arquivo de coordenadas das posições dos clones no DNA. A extensão usada é "*.clones*".
4. Batismo dos clones, incluindo seu nome no final de sua informação de tamanho e coordenadas. O nome é formado por um número sequencial, prefixado pela letra "c" de clone, por exemplo `c36`.
5. Criação das coordenadas das pontas à esquerda e à direita dos clones. O ponto de referência das coordenadas é o DNA, na direção do arquivo "*.dna*" para as pontas esquerdas, e na direção do "*.dna-rev*" para as pontas direitas.

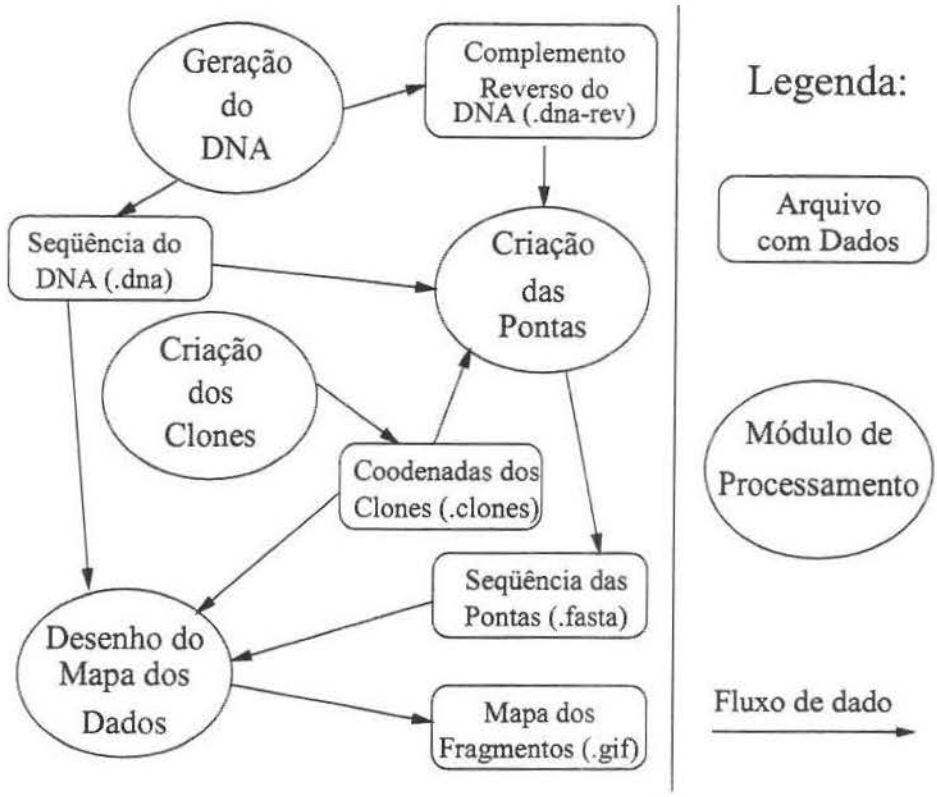

Figura 3.2: Visão geral do gerador de dados.

6. Geração e batismo da seqüência das pontas com base nas coordenadas criadas. A ponta à direita é gerada a partir do DNA, enquanto a ponta à esquerda vem do complemento reverso do DNA. Por fim, o nome da ponta de clone é formado pelo nome do clone e sufixado por "-a", se é ponta mais à esquerda, ou "-b", se é a mais à direita. Por exemplo, `c36-a` ou `c36-b`.

7. Ordenação das pontas por nome. Observemos que dadas duas pontas de um mesmo clone, a ponta mais à esquerda do clone vem imediatamente antes da ponta mais à direita.

8. Transformação do arquivo de seqüências de pontas no formato FASTA [15].

9. Geração do mapa de distribuição dos clones gerados.

Como os testes foram feitos com variação da cobertura com reads shotgun inseridos nos conjuntos de dados gerados anteriormente, também foi implementado uma versão que gera reads shotgun para o DNA e clones gerados previamente. Os dados necessários são:

- arquivo FASTA da seqüência de DNA;
- tamanho mínimo e máximo dos reads shotguns a serem gerados;
- cobertura total desejada de reads shotgun no DNA. A cobertura determinará a quantidade de reads que será gerada.

A geração dos reads shotgun passam pelos seguintes passos:

1. quebra aleatória para criação dos reads, resultando num arquivo de coordenadas de suas posições no DNA;
2. batismo dos reads. O nome é formado por um número sequencial, com o número de dígitos igual ao do tamanho do DNA, com as letras "sh" prefixando os números. Para um DNA de 100000 bp, teríamos o read sg000379, por exemplo. O seu nome será guardado depois de sua informação de coordenadas. O arquivo terá a extensão "*.shg-clones*";
3. geração da seqüência dos reads baseados nas coordenadas criadas e na seqüência de DNA. O arquivo resultante será no formato FASTA e a extensão será "*.shg-fasta*".

O conjunto de dados que conjugará os dados das pontas de clones e reads shotgun será formado pela concatenação dos arquivos "*.fasta*" com "*.shg-fasta*" e "*.clones*" e "*.shg-clones*".

3.2 Gerenciador de Iterações

O módulo "gerenciador de iterações" do ambiente de simulação OSS é o administrador dos vários submódulos executados para se fazer a montagem de fragmentos OSS, de forma que os arquivos necessários para um submódulo já tenham sido criados pela rotina anterior. A cada execução desse conjunto de programas temos uma iteração e o ciclo se repete tanto quanto for necessário.

A seguir, estaremos descrevendo, com detalhes, os parâmetros recebidos por este módulo e cada um dos arquivos gerados ou modificados e, mais resumidamente, os submódulos invocados pelo gerenciador, que serão detalhados na seção 3.3. Uma visão geral das relações entre os submódulos e os arquivos é melhor obtida pelo esquema da figura 3.3 da página 39.

3.2.1 Parâmetros

Dado o nome do arquivo no formato FASTA [15] com o conjunto de fragmentos que entraram na montagem como parâmetro do módulo gerenciador de iterações, supõe-se que os outros arquivos necessários e os gerados no meio de cada iteração têm o mesmo nome, diferindo-se apenas em suas extensões. As extensões serão listadas mais à frente.

Outros parâmetros, neste caso opcionais, são:

- a iteração i em que este módulo começará a processar, supondo-se que os resultados da iteração anterior estão corretamente disponíveis para se iniciar uma nova. Sem esse parâmetro, o módulo considera que irá começar o processamento do zero, quando os dados não sofreram nenhum processamento e $i = 0$. Caso esse parâmetro seja informado, é obrigatório informar também a extensão, em pares de base, da montagem feita até a iteração i. A extensão final é usada como um dado de parada do módulo, por isso a sua importância;

- total de clones processados até a iteração dada e a quantidade de pares bases que eles representam no esforço de seqüenciamento. Estes dados farão parte das informações comparativas entre os algoritmos para os vários conjuntos de dados, portanto recomenda-se que estes dados sejam informados quando a iteração não se inicia desde o começo;

- a iteração n em que o módulo deve parar.

3.2.2 Arquivos necessários para rodar cada iteração

- Arquivo do DNA no formato FASTA. Extensão do arquivo é "*.dna*"

- Arquivo no formato FASTA, das seqüências das pontas dos clones. A extensão dele é "*.fasta*"

- Arquivo com extensão é "*.clones*", onde são guardadas as coordenadas dos clones no DNA.

3.2.3 Arquivos modificados a cada iteração

Listaremos a seguir os arquivos que são modificados a cada iteração da montagem OSS. A cada iteração, apenas é guardada uma cópia do arquivo usado na iteração imediatamente anterior à mudança dos dados, anexando "-old" ao final da extensão. Às extensões destes arquivos não é adicionado o número i da iteração, já que não há interesse em conservá-los para futuras consultas. Os dois primeiros arquivos dessa lista são mudados pelo

submódulo de seqüenciamento, que é quem extrai a seqüência consenso da região montada pelo submódulo scaffold.

- Arquivo, no formato FASTA, das seqüências das pontas dos clones. Sua extensão é "*.fasta*".
- Arquivo "*.clones*" com as coordenadas dos clones.
- Arquivo "*.cross*" com as informações de alinhamento das seqüências contidas em "*.fasta*". Este arquivo é recriado a cada iteração pelo programa `cross_match` — um programa para comparação de seqüências, que integra o software de montagem `phrap` [10]. O interesse nesse arquivo é essencialmente para gerar o grafo de sobreposição OSS.
- Arquivo "*.ignore*" é criado a cada iteração e contém a lista dos fragmentos que foram ignorados na construção do grafo de sobreposição e os seus respectivos motivos: contido em outro, ou identificado pelo `cross_match` como duplicado.

3.2.4 Arquivos criados

Dois arquivos são criados no início do módulo gerenciador e eles vão agregando informações ao longo das iterações:

Arquivo "*.log*": este arquivo funciona como relatório de acompanhamento dos passos do gerenciador ao longo do processamento. Nele contém as chamadas de submódulos, alguns de seus resultados de controle, seus tempos de execução em CPU, a evolução da montagem em cada iteração e o resultado final ao término das iterações. Este arquivo auxilia no diagnóstico de resultados inesperados, uma vez que mostra os submódulos executados e os estados da montagem OSS.

Arquivo "*.resume*": neste arquivo são registrados, no final de cada iteração, os principais resultados da montagem OSS, que são:

- o número de iterações da montagem;
- o tempo total de CPU usado até então;
- a percentagem do genoma coberto pela montagem na última e penúltima iteração;
- a quantidade de clones usados pelos scaffolds e que foram inteiramente seqüenciados para agregar nova informação na próxima iteração;
- o total de pares de bases que estes clones representaram.

Arquivo "*.timecpu*": tempo de execução em CPU de cada submódulo do gerenciador de iterações. Este arquivo é usado apenas para analisarmos os percentuais de tempo de cada submódulo.

Arquivo "*.timesys*": tempo de sistema que foi usado para executar cada submódulo do gerenciador de iterações.

Em cada iteração, outros arquivos serão criados. Os nomes destes arquivos são formados através da concatenação do nome do conjunto de dados, a extensão específica ao tipo de arquivo gerados e o número i da iteração em que eles foram criados. Ou seja:

$$nome\ do\ arquivo = nome\ do\ conjunto\ de\ dados + extensão + i.$$

Como a iteração normal começa do zero, se $i = 0$ (zero), o i é omitido do nome dos arquivos.

As extensões associadas aos arquivos criados são:

- "*.graph*": O arquivo abriga o grafo de sobreposição OSS, gerado a partir do resultado da análise do `cross_match` escrito em "*.cross*";
- "*.path*" ou "*.2path*": arquivo contendo o caminho encontrado no grafo de sobreposição, que representa um scaffold da montagem OSS. A diferença da extensão é apenas para indicar o resultado de diferentes algoritmos usados no submódulo de procura do scaffold. Neste caso, essas extensões dão a dica de que são resultados dos algoritmos 1 e 2, respectivamente;
- "*.gif*": Desenho da distribuição dos clones e reads shotgun e suas características na iteração corrente, além da marcar os clones que entraram no caminho encontrado pelos algoritmos de procura de scaffold.

Para simplificar, vamos convencionar que uma referência no texto, a qualquer um desses três arquivos mencionados anteriormente, será feita só pela extensão. Subentendendo-se que há um i acoplado ao final deles, dependendo da iteração em que eles são gerados ou usados no gerenciador de iterações.

3.2.5 O processamento de cada iteração por submódulos

Aqui listaremos os vários módulos que compõem uma iteração da montagem OSS do nosso ambiente de simulação, bem como os arquivos gerados e usados. As relações entre eles estão esquematizadas na figura 3.3.

1. execução do `cross_match` para encontrar as sobreposições entre os fragmentos do arquivo "*.fasta*", gerando o arquivo com extensão "*.cross*";

2. montagem do grafo de sobreposição OSS, a partir dos dados em "*.cross*", gerando resultados em "*.graph*";

3. procura do scaffold, caminho no grafo "potencialmente" de maior extensão, e dos clones a seqüenciar e armazenagem dos dados no arquivo "*.path*" ou "*.2path*", dependendo do algoritmo usado nesse submódulo;

4. criação do mapa de distribuição dos clones na iteração corrente no arquivo "*.gif*";

5. teste de fim da iteração. Uma iteração chega ao fim se a montagem atingiu 100% do tamanho previsto, ou se, em relação à montagem da iteração anterior, a extensão não mudou;

6. se não é fim da iteração, extração da seqüência "consenso" da região coberta pelo *scaffold*;

7. compressão do arquivo "*.graph*" criado e usado na iteração corrente. A finalidade desse passo é de poupar o espaço ocupado pelos dados;

8. impressão dos dados de tempo e clones seqüenciados até a iteração corrente.

3.3 Os submódulos do gerenciador

3.3.1 Comparação de seqüências

Este módulo compara todas as seqüências do conjunto de dados para identificar os fragmentos que se sobrepõem. O programa usado é o `cross_match` — um dos programas integrantes do software de montagem `phrap` [10] e que tem a função de fazer a comparação de seqüências.

O gerenciador de iterações executa o programa da seguinte forma:

```
cross_match -tags -alignments -masklevel 101
            -retain_duplicates arq.fasta
```

A sobreposição mínima para o `cross_match` considerar uma sobreposição como válida entre duas seqüências é de 14 bp e uma pontuação maior que 30 no alinhamento. A opção `-tags` ativa a colocação de palavras chaves para identificar os resultados do arquivo de saída, a opção `-alignments` mostra o alinhamento encontrado entre duas seqüências, já a opção `-masklevel 101` faz com que o programa relate todos os alinhamentos encontrados e, por fim, a opção `-retain_duplicates` inclui as seqüências repetidas de entrada no processo de comparação [10].

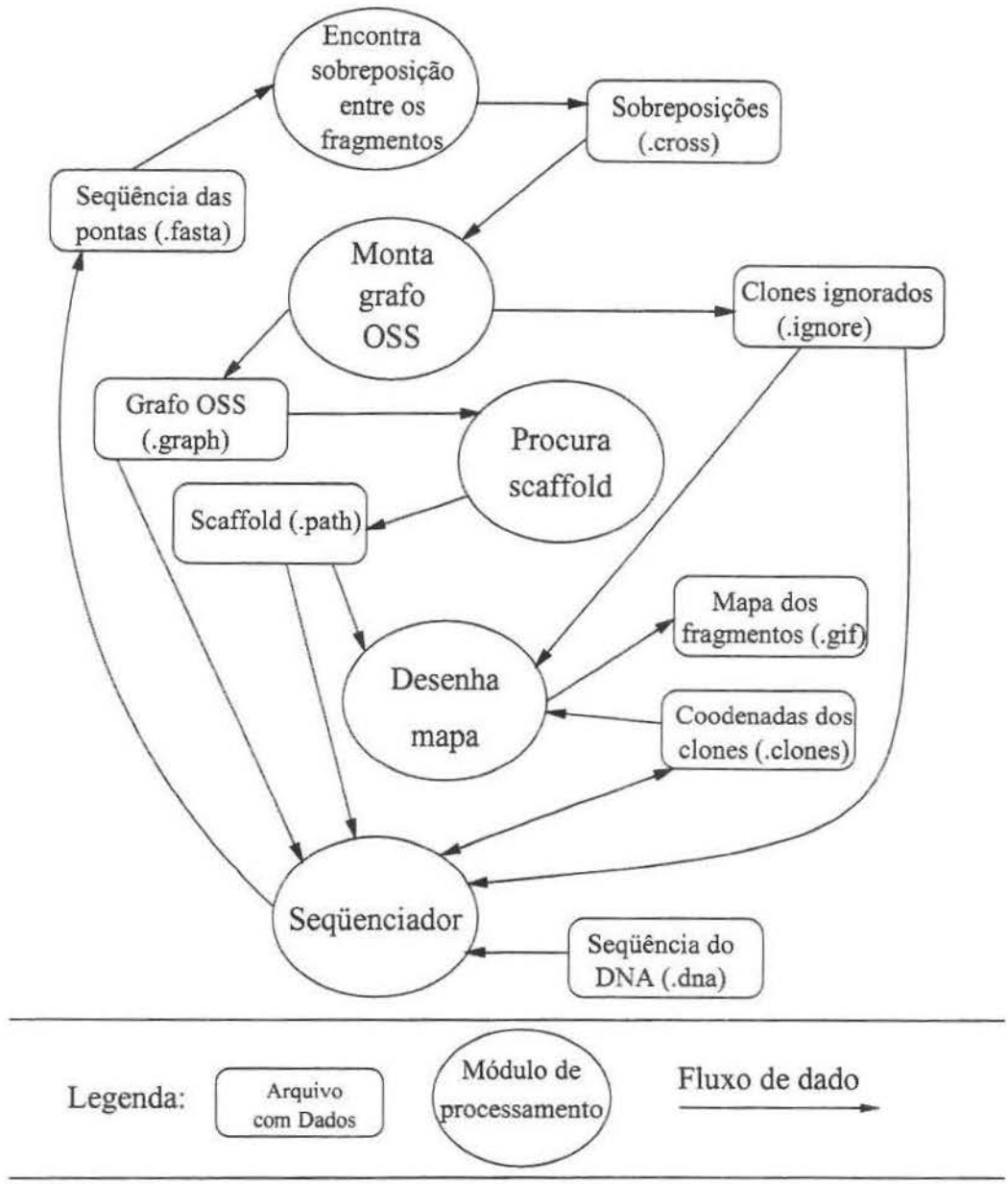

Figura 3.3: Visão geral do gerenciador de iterações.

3.3.2 Construção do grafo de sobreposição OSS

Esse programa é baseado no montador do grafo de sobreposição implementado no trabalho de mestrado de Fábio Cerqueira [3]. Como o nosso grafo possui características próprias, houve modificações consideráveis.

O grafo de sobreposição OSS é um grafo orientado, com peso nas arestas. Ele introduz os conceitos de arestas de sobreposição e as de ligação, conforme explicado no capítulo 1 desse trabalho.

As primeiras são arestas indicadoras de alinhamentos entre a extremidade à direita de uma seqüência com a extremidade à esquerda de outra seqüência.

As arestas de ligação representam a conexão natural entre duas pontas do mesmo clone, pois elas não aparecem sozinhas.

O peso nas arestas indica a quantidade de pares de bases que separam o início da primeira seqüência até o início da segunda seqüência. Se existe uma aresta de peso p, saindo de um vértice a para o vértice b, então existe outra aresta, a complementar ou simétrica, de b para a com peso $-p$.

Parâmetros

Os parâmetros que este módulo espera receber são: nome do arquivo "*.cross*", gerado pelo `cross_match`, e o arquivo "*.fasta*" processado pelo programa.

Processamento

Leitura dos fragmentos Este módulo basicamente é dedicado à leitura do arquivo "*.fasta*". Cada seqüência lida resulta em um vértice e mais outro vértice representando seu complemento reverso.

Os dados associados aos vértices são:

- seu rótulo;
- tamanho da seqüência que representa;
- em quem ele está contido;
- tamanho do clone ao qual a seqüência pertence.

Processamento dos resultados do arquivo "*.cross*" Serão verificados os alinhamentos que aparecem no arquivo "*.cross*", a fim de decidir quais arestas de sobreposição entre os fragmentos serão criadas, bem como quais são os fragmentos que estão contidos em outros.

Vale explicar aqui que o `cross_match`, na versão que utilizamos (0.990319), quando há sobreposição entre seqüências com a mesma direção (do jeito que veio do arquivo ".fasta"), não relatará a sobreposição de seus complementos reversos, cabendo ao módulo incluir as arestas devidas.

Filtragem e exclusão dos contidos O resultado deste filtro é uma lista de vértices que devem ser eliminados do grafo, pois eles representam clones que estão contidos em outros.

Se a seqüência for ponta de um clone, ela só será eliminada se a outra ponta, do mesmo clone também estiver contida no mesmo fragmento e as posições das pontas deverão ser tais que a ponta esquerda do clone esteja antes da ponta direita e na mesma orientação.

Caso a seqüência não seja clone, mas um fragmento inteiriço, um *read shotgun*, um clone totalmente seqüenciado, ou algum fragmento inserido no conjunto de dados na iteração anterior, basta estar contida a própria seqüência.

Exclusão das arestas que levam a fragmentos eliminados Varre a lista de arestas removendo aquelas cujo vértice, tanto de entrada como de saída, estejam na lista de vértices eliminados.

Criação das arestas de ligação Para cada clone, separam-se os vértices que representam as suas duas pontas (nomes terminados em -a e -b no arquivo "*.fasta*" original). Sejam s e t estas pontas. Os seus complementos reversos (nomeados internamente como -ac e -bc) serão $\overline{s}$ e $\overline{t}$. Então as arestas de ligação serão $(s, \overline{t})$ e $(t, \overline{s})$. Uma revisão das figuras 1.6 e 1.7 facilitará a compreensão desse parágrafo.

Os pesos da aresta de ligação serão:

$$p(s, \overline{t}) = \textit{tamanho do clone} - |\overline{t}|$$

e

$$p(t, \overline{s}) = \textit{tamanho do clone} - |\overline{s}|.$$

As arestas complementares são $(\overline{t}, s)$ e $(\overline{s}, t)$, cujos pesos são

$$p(\overline{t}, s) = -p(s, \overline{t})$$

e

$$p(\overline{s}, t) = -p(t, \overline{s}).$$

Caso o clone separado já esteja todo seqüenciado, a definição de pontas não se aplica a ele. Então, há apenas um único vértice que o representa, digamos que este seja x e z seja o vértice do seu complemento reverso. Neste caso, as arestas de ligação são (x, x) e

(z, z) com pesos iguais a zero. Só serão criadas as arestas de ligação dos clones se nenhum vértice que os representa estiver fora.

Impressão do grafo O grafo é a saída deste módulo. Ele é despejado na própria saída padrão, por isso a saída é redirecionada, pelo gerenciador de iterações, para o arquivo "*.graph*" apropriado.

O seu layout de saída tem o seguinte formato:

CONTEÚDO	BREVE EXPLICAÇÃO
c comentário	Comentário qualquer
p alguma_coisa Num_aresta Num_vértices	Total de arestas e vértices
a vértice vértice peso_da_aresta tipo_da_aresta	Descreve uma aresta
a vértice vértice peso_da_aresta tipo_da_aresta	Descreve outra aresta
...	Outras arestas...
v vértice nome_clone direção tamanho_clone	Descreve um vértice
v vértice nome_clone direção tamanho_clone	Descreve outro vértice
...	Outros vértices

O item *vértice* costuma ser número inteiro.

O item *tipo_da_aresta* pode ser "*l*" de ligação ou "*s*" de sobreposição.

O item *direção* pode ser 1 ou -1 para indicar se a seqüência representada pelo vértice vai para a direção do arquivo "*.dna*" ou do arquivo "*.dna-rev*".

Um pequeno exemplo do arquivo seria:

```
c OSS Overlap graph for DNA3M_375c40K_p450
p * 6 10
a 1 30 36859 s
a 1 1 0 l
a 2 29 2950292 s
a 2 2 0 l
a 27 30 39206 l
a 28 29 -39206 l
a 29 28 39206 l
a 29 2 -2950292 s
a 30 27 -39206 l
a 30 1 -36859 s
v 1 s1649504 1 2987601
v 2 s1649504 -1 2987601
v 27 c1647157-a 1 450
v 28 c1647157-a -1 450
```

```
v 29 c1647157-b -1 450
v 30 c1647157-b 1 450
```

Melhorando o desempenho

O montador do grafo OSS passou por uma avaliação após análise dos tempos que cada módulo do gerenciador de iterações precisou para rodar os testes com os conjuntos de dados apresentados na capítulo 4. Os testes com os conjuntos de dados com tamanhos de DNA de 100, 200, ..., 1000 kbp indicaram que o módulo "Gerador do grafo OSS" — chamaremos também de Monta Grafo — era o responsável por 95% do tempo total de execução do gerenciador de iterações de todos os testes envolvendo os 10 conjuntos de dados citados. A preocupação em diminuir o tempo total de execução do gerador de iterações surgiu porque estava se tornando inviável esperar quase uma semana pelos resultados de testes toda vez que se fazia uma nova rodada de testes para os mesmos dados.

Constatado o culpado, o próximo passo foi separar um subconjunto desses dados e usá-los para medir os tempos de execução de cada uma das grandes tarefas do montador do grafo OSS. As tarefas identificadas foram aquelas descritas anteriormente:

1. leitura e análise dos resultados do arquivo "*.cross*";
2. filtragem e exclusão dos fragmentos/vértices contidos;
3. exclusão das arestas que levam a ou saem de vértices excluídos;
4. criação das arestas de ligação;
5. impressão do grafo OSS.

Como este módulo foi escrito em Perl, não foi possível traçar o perfil do programa (*profile*) como se faz com os compilados. Desta maneira, registramos o tempo de execução entre cada tarefa, coletando os tempos em pontos estratégicos dentro do próprio código.

Os dados escolhidos para a próxima medição foram os com DNA de tamanho 100, 200,...,1000 kbp e cobertura de reads shotgun 10 X, descritos no capítulo 4.

Os resultados mostraram que só os itens 2, 3 e 4 somavam mais de 70% do tempo de execução do programa. Por isso, estes módulos mereceram um olhar mais cuidadoso à procura de pontos ajustáveis. A figura 3.4 mostra a curva de tempo versus conjuntos de dados crescentes para montar o grafo OSS de cada um deles.

A conclusão foi que poderíamos juntar as tarefas desses três módulos em um, eliminando a lista de vértices excluídos — consultada na hora de excluir as arestas de sobreposição, na criação das arestas de ligação e impressão do grafo.

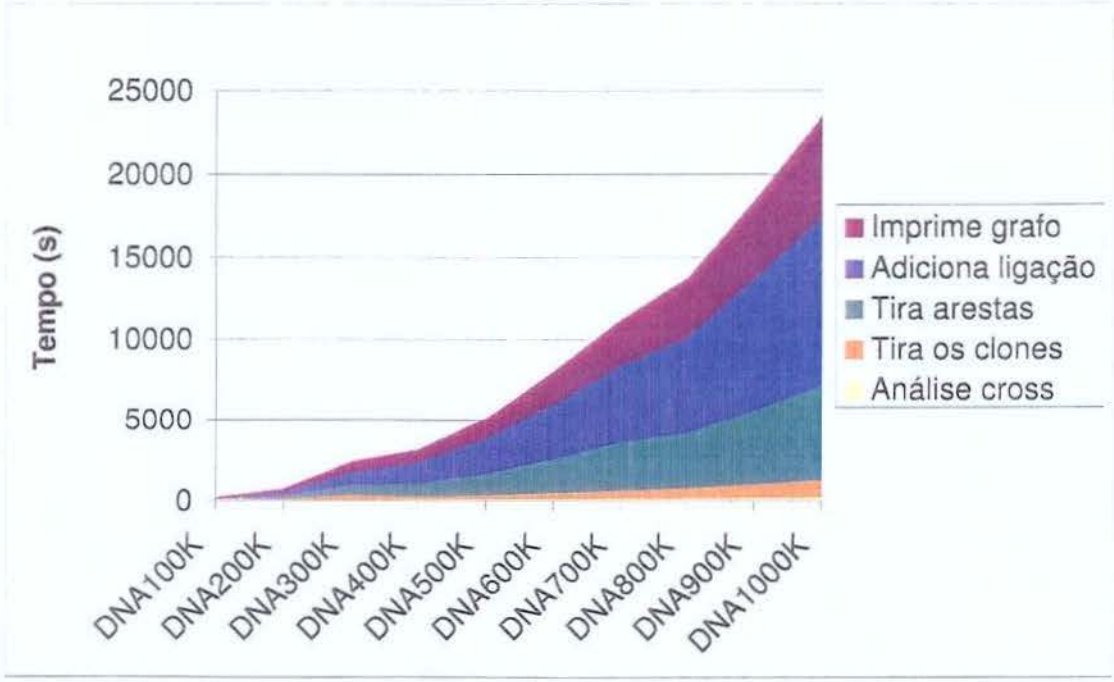

Figura 3.4: Gráfico de evolução do tempo dos módulos no Monta Grafo antigo.

Em Perl, verificar, repetidamente, se um elemento está numa lista é uma operação cara. No módulo antigo, as seguintes operações faziam isso:

1. remoção de arestas de todos os vértices considerados excluídos;

2. inclusão das arestas de ligação dos vértices não excluídos.

Por isso, o tempo do Monta Grafo novo melhorou ao fazermos as operações citadas já na identificação e remoção dos vértices eliminados (operação 1) ou dos que não tem mais chances de serem excluídos (operação 2). Neste caso, sacrificamos a modularidade do programa pela melhora significativa do tempo de execução, conforme mostra o gráfico comparando os tempos totais das duas versões do programa de criação do grafo OSS da figura 3.5. Observe que a melhora foi de uma a duas ordens de magnitude. A tabela de dados das duas versões do Monta Grafo está na figura 3.6 e a atual participação do módulo na montagem OSS pode ser conferida nas seções B.5 e B.6, apêndice B.

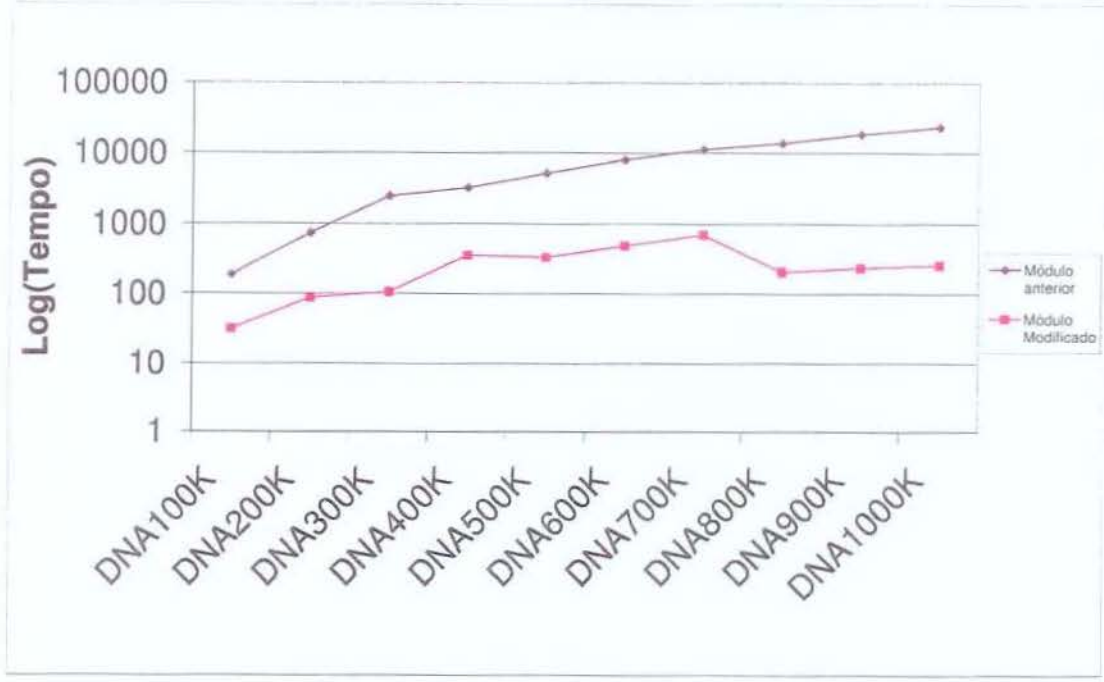

Figura 3.5: Gráfico de comparação de tempo das duas versões do Monta Grafo.

Resumos de tempo (s) de CPU dos submódulos do MontaGrafo OSS

	Módulo anterior						Módulo Modificado				Ganho de Performace		
Submódulos	Análise cross	Tira os clones	Tira arestas	Adiciona ligação	Imprime grafo	Totais	Análise cross	Tira os clones	Imprime grafo	Totais	Economia	% tempo do novo p/ velho	Qtas. vezes Modificado é mais rápido que Anterior
DNA100K	20,78	7,07	43,67	70,22	42,68	184,42	20,63	8,77	1,30	30,70	83,35%	16,65%	6,0
DNA200K	40,47	28,30	185,62	302,75	179,18	736,32	42,12	40,98	2,73	85,83	88,34%	11,66%	8,6
DNA300K	31,53	293,22	637,12	821,32	636,28	2419,47	32,48	72,22	0,02	104,72	95,67%	4,33%	23,1
DNA400K	82,13	130,12	824,72	1360,87	784,75	3182,58	85,58	258,75	5,22	349,55	89,02%	10,98%	9,1
DNA500K	102,95	204,13	1344,62	2218,30	1280,53	5150,53	105,12	216,88	6,32	328,32	93,63%	6,37%	15,7
DNA600K	126,00	328,62	2097,23	3492,98	2035,15	8079,98	128,43	347,28	7,18	482,90	94,02%	5,98%	16,7
DNA700K	149,58	472,23	2950,53	4865,90	2818,50	11256,75	153,63	538,78	8,55	700,97	93,77%	6,23%	16,1
DNA800K	169,32	594,37	3387,60	6035,87	3607,55	13794,70	176,47	15,97	10,03	202,47	98,53%	1,47%	68,1
DNA900K	189,93	807,78	4603,47	8141,63	4759,45	18502,27	202,78	18,13	11,03	231,95	98,75%	1,25%	79,8
DNA1000K	214,60	1042,48	5945,70	10273,78	6043,63	23520,20	222,90	19,50	11,82	254,22	98,92%	1,08%	92,5
Total	**1127,30**	**3908,32**	**22020,27**	**37583,62**	**22187,72**	**86827,22**	**1170,15**	**1537,27**	**64,20**	**2771,62**			
Fatia do tempo	**1,30%**	**4,50%**	**25,36%**	**43,29%**	**25,55%**	**100%**	**42,22%**	**55,46%**	**2,32%**	**100%**			

Figura 3.6: Tabela de dados das duas versões do Monta Grafo.

3.3.3 À procura do *Scaffold*

É possível usar um dos dois algoritmos propostos no capítulo 2 — algoritmo guloso e algoritmo maior componente conexa OSS — para tentar procurar o melhor *scaffold*, a cada iteração, num grafo de sobreposição OSS.

Todos os dois algoritmos consideram que, se há uma ponta de clone representada no grafo, a outra ponta também está, bem como os seus respectivos complementos reversos.

A entrada para os algoritmos é o grafo de sobreposição OSS, sendo que para o segundo algoritmo passamos também o tamanho do DNA para que ele conseguisse tratar o caso circular, conforme explicação no capítulo 2, pois na versão de nosso programa não foi implementada a parte de descobrir o tamanho do DNA.

A saída dos programas é constituída pelas arestas que formam o caminho, a sua extensão e os clones marcados para serem seqüenciados e inseridos na próxima iteração. O formato da saída é o seguinte:

Caminho encontrado pelo algoritmo
$(v_1, v_2, peso)$
$(v_2, v_3, peso)$
...
$(v_{n-1}, v_n, peso)$
Extensao: extensao_do_caminho
Clones marcados
v_1 *nome_clone direção*
v_2 *nome_clone direção*
...
v_n *nome_clone direção*

Agora, vejamos o formato descrito com um exemplo:

```
Caminho encontrado pelo Algoritmo Conjuntos Disjuntos...
(1383,1382,34728)
(1382,1355,-112)
(1355,1354,34546)
(1354,1331,343)
(1331,1330,34945)
(1330,1303,416)
(1303,1302,34531)
Extensao:  139847
Vertices/Clones marcados para sequenciar:
1383 c2160840-b -1
1355 c2126404-b -1
```

```
1331 c2091114-b -1
1303 c2056165-b -1
```

3.3.4 O mapa dos dados

O mapa é uma representação da distribuição dos clones e reads shotgun que entraram na montagem OSS. Nele estão apresentadas as porções conhecidas dos clones e reads (área pintada em roxo) e a porção desconhecida do clone (em branco). Os reads e pontas que formam o *scaffold* encontrado pelo submódulo anterior são pintados em vermelho. Se o clone todo está presente no *scaffold*, o seu corpo é todo contornado em vermelho.

Por considerar a circularidade do genoma, se a coordenada inicial do clone for maior que a final, considera-se que o clone deve ser desenhado em dois pedaços. O primeiro pedaço vai da coordenada inicial até o final do mapa e o segundo vai do início do mapa até a coordenada final. O primeiro pedaço será identificado pelo nome do clone e com "..." como sufixo e o segundo também pelo nome do clone, mas com "..." prefixado.

São usados nesse submódulo os arquivos:

- "*.clones*": para saber as coordenadas no DNA dos clones a serem desenhados;
- "*.fasta*" e" *.dna*" : para obter o tamanho das seqüências conhecidas e do tamanho do DNA;
- "*.path*" ou "*.2path*" e "*.graph*" : para identificar os clones que entraram no caminho que forma o "*scaffold*" e marcá-los de vermelho;
- "*.ignore*" : para ignorar o desenho dos clones que não entraram no grafo.

A resultado do programa é um arquivo no formato GIF com o desenho do mapa. O seu nome é dado como parâmetro de entrada do programa. Um exemplo do que pode aparecer num mapa de distribuição dos dados está apresentado na figura 3.7.

Figura 3.7: Exemplo do mapa de distribuição dos dados (apenas de 143333–200000).

3.3.5 Seqüenciador

Este submódulo altera os dados dos arquivos "*.fasta*" e "*.clones*" com base nos arquivos "*.path*"/"*.2path*", "*.dna*", "*.ignore*" e no próprio "*.clones*".

Resumidamente, o programa pega a coordenada inicial do primeiro clone do scaffold e a coordenada final do último clone como as coordenadas da nova seqüência representando a região coberta pelo caminho no grafo. A nova coordenada é incluída no novo arquivo "*.clones*" e ela é usada para extrair a nova seqüência, a partir de "*.dna*", que será incluída no arquivo "*.fasta*". A nova seqüência adotará o nome do read shotgun que contribuiu com a coordenada inicial ou o nome do clone, trocando a inicial "c" pela letra "s". Assim, se foi a ponta do clone `c235` que contribuiu com a coordenada inicial da nova seqüência, então esta se chamará `s235`.

Os clones que eram representados pelos vértices ignorados, listados em "*.ignore*" são eliminados tanto do "*.fasta*", que guarda suas seqüências, como do "*.clones*", que guarda as coordenadas deles no DNA.

Também são eliminados do arquivo FASTA, os clones inteiros e reads shotguns que fizeram parte do scaffold encontrado, pois elas deram origem à seqüência da região coberta pelo scaffold e podem ser tiradas para diminuir a carga de trabalho da próxima iteração. Isso também acontece com as informações relacionadas às coordenadas desses clone, armazenadas no arquivo "*.clones*".

Ao final da execução deste programa, os arquivos originais "*.fasta*" e "*.clones*" são copiados em arquivos adicionado com "-old" às suas extensões originais, enquanto as novas versões desses arquivos, produzidos por esse seqüenciamento simulado do scaffold, tomam seus lugares para a próxima iteração da montagem OSS.

Capítulo 4

Testes e Resultados

Apresentaremos aqui os testes com os dados artificiais, gerados pelo módulo gerador de dados, sem introdução de erros encontrados normalmente em dados vindo de laboratórios de seqüenciamento. As tabelas de resultados e os gráficos citados nessa seção encontram-se todos no apêndice A (tabelas) e apêndice B (gráficos).

Os conjuntos de dados gerados tem a seguinte convenção de nome:

`DNA`[D]`_`[N]`c`[C]`K_p`[P],

onde [D] é o tamanho DNA em Mbp ou kbp (seguido da letra M ou K, respectivamente), [N] é o número de clones, [C] é o tamanho médio do clone em kbp e [P] é o tamanho das pontas em bp. Por exemplo, o nome do conjunto `DNA3.2M_450c40K_p500` nos indica que temos um DNA de 3200 kbp (3,2 Mbp), 450 clones de aproximadamente 40 kbp com pontas seqüenciadas de 500 pares de bases cada.

Os testes finais apresentados foram feitos numa máquina Digital AlphaServer GS140, com 10 processadores Alpha 21264 EV6 de 524 MHz, palavra de 64 bits, 8 GB de memória física, arquitetura de memória compartilhada e sistema operacional OSF1 versão 4.0. Apesar da máquina possuir vários processadores, cada teste usou apenas um deles.

Os módulos implementados em C/C++ foram compilados com opção *-O3* do compilador gcc/g++ para otimizar as duas versões do programa scaffold, conforme descritos em capítulos anteriores.

4.1 Geração e Teste com Dados Artificiais

Foram gerados 10 conjuntos básicos de dados, contendo apenas pontas de clones extraídas de um DNA circular com tamanho variando de 100 kbp a 1000 kbp, crescendo em múltiplos de 100 kbp, mantendo-se as seguintes características fixas: tamanho médio dos clones de 40 kbp (mínimo de 39,5 kbp e máximo de 40,5 kbp), tamanho das pontas de 500 bp,

cobertura dos clones no DNA de 5 vezes (5 X). Com isso, o que varia é o número de clones, decorrente da necessidade de manter uma cobertura 5 X só de clones nos vários tamanhos do DNA. Além disso, os dados foram gerados de modo a garantir que os clones cubram toda a região do DNA, portanto não teremos buracos. Em outras palavras, se tivéssemos a seqüência inteira dos clones, isso já seria suficiente para obter o consenso.

A partir de cada conjunto básico gerado, mais 10 conjuntos de dados foram criados, adicionando-lhe reads shotgun com cobertura que vai de 1 X até 10 X e tamanho variando de 400 a 500 bp. Logo, obtemos 11 conjuntos de dados a serem testados no gerenciador de iterações para cada conjunto básico só com clones.

O objetivo destes testes é obter um panorama do comportamento dos dois algoritmos descritos anteriormente, bem como do ambiente de simulação para volume de dados crescentes. O genoma gerado é circular, pois pretende-se tratar a montagem de DNA de uma forma geral, sendo que o genoma linear é um caso específico e com menos problemas que o circular. Testes com DNA circular nos revelam as questões que ainda devem ser tratadas.

Além de tudo, a fim de avaliar o desempenho do ambiente de simulação para volume de dados maior, também criamos o conjunto de dados com tamanho de DNA circular de 3000 kbp (próximo ao tamanho do DNA da bactéria *Xylella fastidiosa*) e os respectivos conjuntos com reads shotgun de 1 X a 10 X.

Os testes finais foram rodados apenas uma vez para cada conjunto de dados, totalizando 121 execuções do gerenciador de iterações para cada algoritmo, que resultaram em amostras suficientes para nos dar um panorama geral do comportamento do gerenciador de iterações e de seus submódulos, incluindo aí as implementações dos algoritmos propostos no capítulo 2.

4.2 Resultados dos testes

As tabelas de resultados do apêndice A são compostas por colunas com os seguintes dados:

1. a cobertura dos reads shotgun. Nossos testes foram feitos com cobertura de 0 X até 10 X. 0 X indica que não houve adição de reads shotgun;

2. total de reads shotgun adicionados ao conjunto composto, inicialmente, só de clones;

3. número de iterações executadas pela montagem OSS antes da parada;

4. tempo total de CPU do gerenciador de iterações para aquela instância de dados, dado em segundos;

5. dados específicos da penúltima e última iteração: número de fragmentos no scaffold encontrado naquela iteração, número de fragmentos no scaffold que eram pontas

de clones (os restantes são shotgun ou seqüência inteira de clone) e o percentual do genoma coberto pelo scaffold encontrado. O cálculo do percentual é feito pelo extensão do scaffold encontrado (ext) sobre o tamanho do DNA previsto ($|DNA|$), portanto $ext/|DNA|$. Para a penúltima iteração, pode aparecer um traço nos campos de dados, significando que não houve resultado, pois para aquele conjunto de dados, ocorreu uma única iteração na montagem OSS;

6. esforço total de seqüenciamento, medido em número de clones que seriam seqüenciados, olhando para o scaffold encontrado pelo programa, e total de bases que eles representam. Este valor é calculado pelo gerenciador de iterações e ele computa como um clone a ser seqüenciado numa iteração apenas quando as duas pontas do clone aparecem uma seguida da outra no caminho. Os reads shotgun e as pontas de clone inicialmente seqüenciados não são computados neste total.

4.2.1 Resultados com o Algoritmo 1

Notemos que nas tabelas da seção A.1, página 69, como era esperado, os genomas gerados são sempre fechados pela montagem OSS usando o algoritmo 1. Conforme apontado na seção 2.1, página 21, com o genoma circular podem ocorrer casos em que o algoritmo guloso encontre um scaffold que cubra o genoma mais de uma vez, por isso aparecem percentuais do genoma coberto pelo scaffold ultrapassando o 100% e chegando a mais de 400% (tabela A.1) nas 11 tabelas de resultados do gerenciador de iterações apresentados.

Esta é uma questão importante a ser tratada, e foi incluída na lista de desenvolvimentos futuros no capítulo 5.

4.2.2 Resultados com o Algoritmo 2

Ao contrário do algoritmo 1, nem sempre o algoritmo 2 conseguiu fechar o genoma. Nessa seção, analisaremos, caso a caso, o motivo do genoma não ser fechado pela montagem OSS usando o algoritmo 2.

Pelos resultados das tabelas apresentadas na seção A.2, notamos que o percentual do genoma coberto pelo scaffold encontrado pelo algoritmo 2 pode passar de 100% também, mas nunca em mais de 190%, diferentemente do algoritmo guloso, cujos resultados foram apresentados anteriormente.

Na grande maioria dos casos o algoritmo comportou-se como esperado, com o genoma sendo fechado com uma cobertura ligeiramente acima de 100%. Este excesso é normal, em se tratando de genoma circular, e acontece porque o scaffold é composto no final por uma última seqüência cujo comprimento final ultrapassa o ponto em que o primeiro fragmento

começa. A seguir, enumeramos os problemas que apareceram na montagem OSS com o algoritmo 2. As figuras das páginas seguintes servirão de apoio para as explicações.

1. **Clone circular**. Nos testes com o conjunto básico `DNA100K_13c40K_p500` (tabela A.12), os conjuntos com 1 X, 2 X e 6 X shotgun deixaram de cobrir um buraco de até 13% do tamanho total do genoma, porém os mapas de distribuição (todos com o padrão da figura 4.1) mostram que se incluíssemos somente um dos clones remanescentes no mapa, o genoma já estaria fechado. O algoritmo não conseguiu identificar isso, pois as pontas possuem coordenadas dentro da região coberta pelo fragmento selecionado, que representa a maior e menor coordenada no mapa. Esse é um caso que num genoma linear não aconteceria, e mostra uma situação na qual genomas circulares precisam de cuidados especiais.

2. **Mais de uma volta**. Para os conjuntos com shotgun 7 X e 8 X da tabela A.12, a montagem fechou o genoma em torno de 180%, o que é um exagero. O mapa do dado com shogun 7 X está na figura 4.2 e podemos ver que o scaffold encontrado (composto por fragmentos marcados em vermelho) cobre certas regiões em mais de uma vez com os clones e fragmentos. Esse é um exemplo que evidencia um dos efeitos da alteração proposta no capítulo 2, seção 2.2.1, para conseguir tratar o caso do genoma circular. Por causa das arestas tiradas, o scaffold que começa no vértice de menor coordenada (ponta inicial de `c12`) precisou usar outras arestas do grafo para se chegar ao vértice de maior coordenada (`sg000740`). Estas arestas acabaram resultando em um caminho que utilizava fragmentos que cobriam a mesma região. As arestas retiradas são as de sobreposição ou ligação conectando fragmentos que se situam (pelas coordenadas iniciais) à direita do fragmento `s2` (coordenada zero) com os que estão à esquerda dele. Para facilitar a visualização do problema, a linha vertical verde na figura 4.2 indica o ponto zero do mapa, atribuído pelo algoritmo 2. Veja na figura que existe um scaffold muito melhor que o escolhido.

3. **Componente curta**. Analisando os resultados de cada iteração para o conjunto `DNA200K_25c40K_p500` com cobertura shotgun 4 X (tabela A.13), por exemplo, percebemos que o algoritmo consegue um scaffold que cobre o genoma em 98,87% já na primeira iteração (não mostrado na tabela), mas na segunda iteração (a penúltima), o algoritmo escolheu os fragmentos que ficavam no meio do genoma e que não pertenciam e nem se conectavam à região coberta pelo scaffold da iteração anterior para formar o scaffold da iteração seguinte, que cobre 1,08% do genoma. A figura 4.3 mostra o mapa da penúltima iteração desse conjunto para entendermos o problema. Observamos que, caso o algoritmo tivesse percebido que o clone `c24` ou o clone `c25` fechavam o genoma, o processamento acabaria uma iteração antes. A escolha feita

na segunda iteração deveu-se ao critério de preferir procurar o scaffold em componentes conexas com mais membros (descrito na seção 2.2, página 22). Na última iteração, o programa pára, porque o scaffold é composto por apenas um fragmento, o `sg000141`, originado após primeira iteração.

Para ver o mesmo problema com outra nuance, vale um comentário sobre o conjunto `DNA700K_88c40K_p500` com 3 X (tabela A.18). Este é um caso em que já na quarta iteração, o genoma atingiu 98,75% de cobertura (não mostrado), mas ainda havia 4 componentes conexas OSS formadas por fragmentos de read shotgun, o que fez com que a montagem continuasse por mais 4 iterações procurando scaffold nessas componentes, para chegar na décima iteração e parar, quando o algoritmo indica um scaffold apenas com fragmento `sg001643`, que é resultante do scaffold encontrado na quinta iteração. Na figura 4.4, temos o mapa com a configuração dos dados após a última iteração, onde os 5 fragmentos do meio do genoma são formados após as iterações 5–9. Observemos que o processamento teria acabado na quinta iteração se o algoritmo tivesse percebido que só com clone `c23` ou `c24` o genoma fecharia.

Na tabela 4.1 está tabulado o diagnóstico dado para cada resultado das tabelas relativas ao algoritmo 2 (tabelas A.12 a A.22). O identificador do diagnóstico é o número dado para cada item descrito acima. Quando o resultado foi normal (sem problema), isto é indicado por um hífen ("-"). Consideramos "normal" uma cobertura entre 100% e 130%.

Tabela	Conjunto	Cobertura Shotgun										
		0	1	2	3	4	5	6	7	8	9	10
A.12	DNA100K_13c40K_p500	-	1	1	-	-	-	1	2	2	2	-
A.13	DNA200K_25c40K_p500	-	-	-	-	3	2	-	-	-	-	-
A.14	DNA300K_38c40K_p500	-	1	-	-	1	-	-	-	-	-	-
A.15	DNA400K_50c40K_p500	-	1	3	-	-	-	-	-	-	-	-
A.16	DNA500K_63c40K_p500	1	-	-	-	-	-	-	-	-	-	-
A.17	DNA600K_75c40K_p500	1	-	-	-	-	-	-	-	-	-	-
A.18	DNA700K_88c40K_p500	-	-	-	3	1	-	-	-	-	-	-
A.19	DNA800K_100c40K_p500	-	-	-	-	-	-	1	-	-	-	-
A.20	DNA900K_113c40K_p500	-	-	-	-	1	-	-	-	-	-	-
A.21	DNA1000K_125c40K_p500	-	-	-	-	3	-	-	-	-	-	-
A.22	DNA3M_375c40K_p450	1	-	-	-	1	-	-	-	-	-	-

Os números dessa tabela se referem à explicação iniciada na página 54 para as tabelas de resultados do apêndice A:
1= Clone circular; 2= Mais de uma volta; 3= Componente curta.

Tabela 4.1: Diagnóstico dos resultados das tabelas para o algoritmo 2.

Figura 4.1: Mapa da montagem final DNA100K_13c40K_p500 com 1 X (editado). O algoritmo não viu que o clone c11 (ou c12) fecharia o genoma.

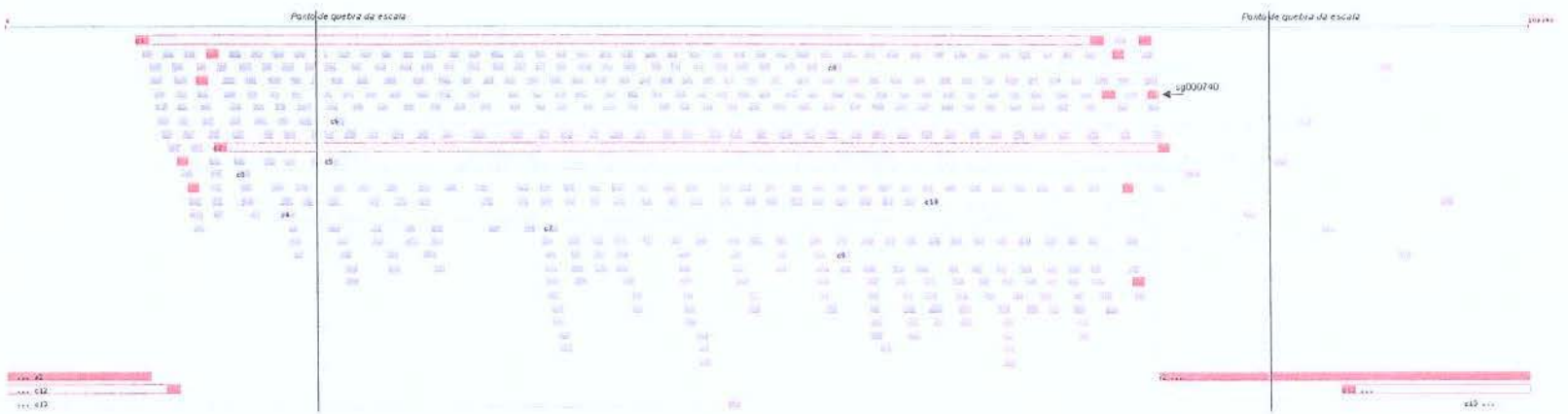

Figura 4.2: Mapa da montagem do DNA100K_13c40K_p500 com 7 X (editado). O scaffold escolhido foi: c12, fragmentos shotgun ligando a c2, s2, c1, seguido de mais fragmentos shotgun até chegar ao sg000740, totalizando cerca de 180% do genoma. Um scaffold melhor seria s2, c1 e fragmentos shotgun ligando a s2.

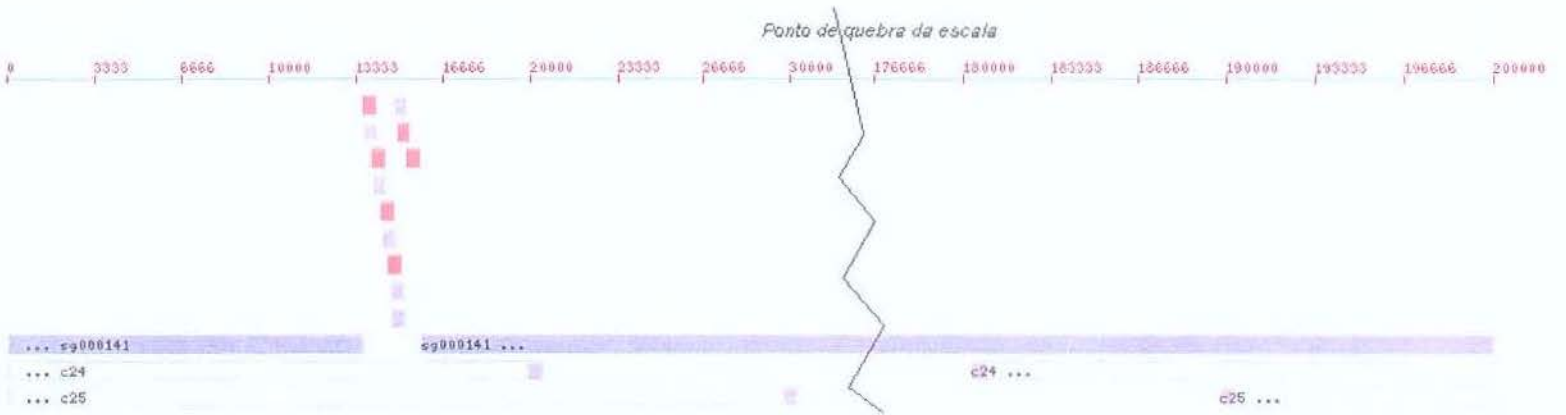

Figura 4.3: Mapa da montagem do DNA200K_25c40K_p500 com 4 X (editado). Foi escolhido indevidamente um scaffold com os fragmentos pequenos por ser de uma componente com mais membros.

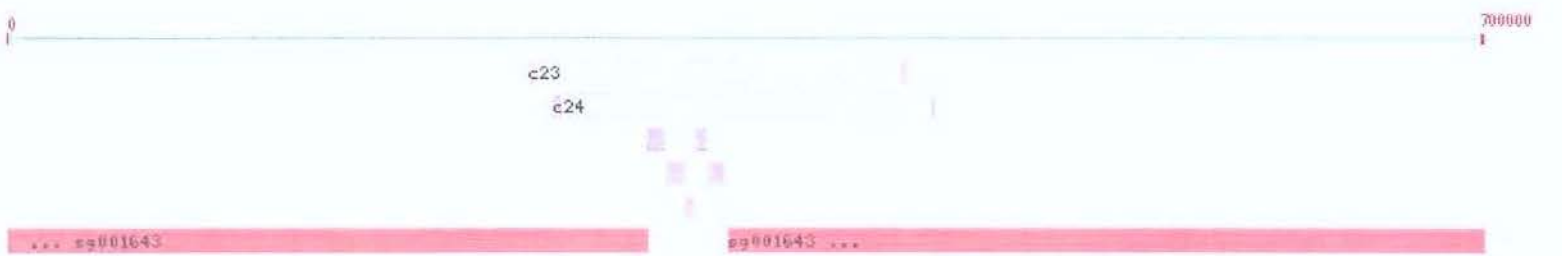

Figura 4.4: Mapa da montagem do DNA700K_88c40K_p500 com 3 X (editado). Mesmo problema da figura anterior, só que dentro de cada fragmento do buraco deixado por sg001643.

4.3 Avaliação dos dois algoritmos Scaffold

Analisamos o desempenho dos dois algoritmos por vários critérios e estes dados serão apresentados em forma de gráficos comparando:

- total dos tempos de CPU empregados pelos algoritmos para encontrar o scaffold versus a cobertura de reads shotgun. O tempo total corresponde à soma dos tempos de execução do submódulo scaffold, em cada iteração do gerenciador de iterações para fazer a montagem OSS de um conjunto de dados. Por causa da grande diferença de tempo entre os dois algoritmos, os gráficos para essa análise estão em escala logarítmica e são mostrados nas figuras B.1 a B.11 (apêndice B). O gráfico resumo está na figura B.12.

- número total de iterações executadas até a sinalização de fim da montagem OSS pelo gerenciador de iterações versus cobertura shotgun. Esses totais estão tabelados nos resultados apresentados nas seções anteriores e nos gráficos do apêndice B, seção B.2.

- esforço adicional de seqüenciamento ou total de pares bases necessário para desvendar o genoma a partir do conjunto de dados inicial. Esse total é obtido pela soma dos tamanhos dos clones que efetivamente compuseram os scaffolds encontrados a cada iteração. Este dado também está nas tabelas de resultados apresentados anteriormente. Os gráficos para esta análise encontram-se na seção B.3.

Os gráficos de comparação do tempo de execução do módulo scaffold (figuras B.1 a B.11 na seção B.1 do apêndice) indicam que o algoritmo 2 é mais lento, chegando a abrir uma diferença de duas ordens de grandeza em relação ao tempo do algoritmo 1. Tipicamente, o algoritmo 1 leva por volta de 10 segundos e o algoritmo 2 leva cerca de 1000 segundos, para DNA de até 100 kbp e cobertura 10 X. No teste com DNA de 3 Mbp (tabela B.11), o algoritmo 1 alcançou 100 segundos e o algoritmo 2, 10000 segundos aproximadamente.

O último gráfico desta série de comparação de tempo do scaffold mostra, na figura B.12, a curva de tempo em relação ao crescimento do DNA. O valor do tempo para cada DNA é obtido pela soma dos seus respectivos tempos de execução com coberturas de 0 X a 10 X.

Observamos pelos gráficos de iterações (figuras B.13 a B.23, seção B.2) que, em geral, usar o algoritmo 1 na montagem OSS resulta em maior número de iterações para fazer a montagem quando a cobertura shotgun é menor que 5 X, comparado ao algoritmo 2; depois disso a situação se inverte.

É bom frisar que quanto mais reads shotgun inseridos, mais informações temos para a montagem, portanto menos iterações são necessárias. Por exemplo, para o conjunto básico `DNA900K_113c40K_p500` (figura B.21), o algoritmo 1 executou 14 iterações em conjunto

sem reads shotguns, enquanto com cobertura 7 X uma iteração apenas foi suficiente. Já para o algoritmo 2, os números foram 10 e 3 iterações, respectivamente. Se somarmos quantas vezes o algoritmo de scaffold foi executado para cada um dos grandes conjuntos, o algoritmo 1 foi quem mais iterações fez, conforme mostra o gráfico da figura B.24.

Por outro lado, vale lembrar que uma iterações a mais em dados menores provoca bem menos impacto no tempo de execução de uma montagem OSS do que em conjunto com volume de dados maiores. Por isso, uma iteração a mais que o algoritmo 2 precise para montar um genoma maior, com grande volume de dados (mais de 6 X de read shotgun), seu tempo acaba destoando do tempo total da montagem usando o algoritmo 1, como veremos na seção 4.4.1.

Ainda comparando o número de iterações dos dois algoritmos, veremos algumas evidências de que o sistema usado pelo algoritmo 2 para encontrar os vértices mais extremos de um mapa, e depois encontrar o caminho entre eles, ainda não conseguiu tratar o caso do genoma circular. Por exemplo, o algoritmo 2, às vezes, faz mais iterações que o necessário, afinal o algoritmo 1 (guloso) conseguiu fazer a montagem em menos iterações, por exemplo para conjuntos com cobertura shotgun de 9 X.

Já no quesito esforço de seqüenciamento necessário para cobrir o genoma, o algoritmo 2 é o que menos seqüencia e mais aproveita as seqüencias já existentes para formar o scaffold, ou melhor, quanto mais reads shotgun, menor o esforço de seqüenciamento após início da montagem OSS com os dados existentes. Essa característica do algoritmo 2 é nitidamente visível para conjunto de dados maiores. Percebemos também pelos gráficos das figuras B.25 a B.35, que o algoritmo 1 não tira muita vantagem do aumento de informações com maior cobertura shotgun, chegando até a apresentar o aumento do seqüenciamento quando a cobertura com shotgun era maior. Por exemplo, para o conjunto básico `DNA1000K_125c40K_p500` (tabela B.34), enquanto sem reads shotgun o algoritmo 1 seqüencia 1,2 Mbp e o algoritmo 2 seqüencia 1,3 Mbp, com 10 X de reads shotgun esse números vão para 1,2 Mbp e 0,08 Mbp, respectivamente.

O gráfico da figura B.36 resume o esforço de seqüenciamento total em cada conjunto.

4.4 Avaliação do gerenciador de iterações por algoritmo

Nessa seção, compararemos o desempenho do gerenciador de iterações usando os algoritmos propostos para o módulo scaffold e o seu impacto no tempo de execução da montagem OSS. Também analisaremos o peso percentual de cada módulo no tempo de execução total do gerenciador de iterações.

4.4.1 Tempo total do gerenciador

Os gráficos de tempo total do gerenciador nos mostram que apesar das grandes diferenças do tempo de execução dos algoritmos scaffold, essa diferença não tem o impacto da mesma proporção no gerenciador de iterações. A explicação é que o tempo do módulo do scaffold representa apenas uma pequena fatia do gerenciador, quando comparado ao tempo de execução do `cross_match` (abreviado como *cross* na legenda dos gráficos) – o comparador de seqüências para procurar sobreposições – e do Monta Grafo, como mostraremos na análise do gerenciador de iterações por módulos.

Para tamanhos de DNA até 1000 kbp, em geral, o tempo da montagem OSS usando o algoritmo 1 é bem próximo do usando o algoritmo 2 (com o tempo menor) até chegar nos conjuntos com coberturas shotgun maiores que 7 X, quando o tempo com algoritmo 2 cresce mais rápido que o do algoritmo 1 e acaba ficando mais lento que este. Vejamos figuras B.37 a B.46.

Um ponto interessante a se notar é que para o conjunto maior, `DNA3M_375c40K_p450` (figura B.47), em oposição aos conjuntos menores, o tempo em CPU do gerenciador de iterações tem uma tendência de queda à medida que se aumenta a cobertura shotgun. Esse comportamento é observado para qualquer um dos algoritmos empregado na montagem OSS até a cobertura 6 X, quando o tempo do algoritmo 2 ascende e ultrapassa o do algoritmo 1.

Esse comportamento reflete o fato de que é para DNA maiores que se percebe realmente como é o impacto de se inserir mais reads shotgun nos conjuntos. Além de significar mais dados a serem comparados e analisados, aumentar a cobertura shotgun pode diminuir notavelmente o número de iterações necessárias para a montagem desvendar o genoma, portanto menos tempo de processamento. O ponto de ultrapassagem do algoritmo 1 pela curva do algoritmo 2 marca o ponto (cobertura 7 X) em que o total de iterações executadas pelos dois se aproximam, até que na cobertura 9 X, o algoritmo 2 é quem precisa de mais iterações para terminar a montagem OSS.

4.4.2 Gerenciador de iterações módulo a módulo

Podemos observar pelos gráficos apresentados na seção B.5 que, com o algoritmo 1, os módulos de maior peso do gerenciador de iterações são o `cross_match` e o Monta Grafo. O `cross_match` tende a diminuir sua participação no tempo, com coberturas shotgun crescentes, mas não nos enganemos com a aparente relação direta tempo vs. quantidade de reads shotgun. Na verdade, o tempo do `cross_match` é proporcional ao número de iterações totais executadas e ao volume de dados. Como comparar seqüências em larga escala é custoso, executar essa operação repetidas vezes faz com que seu tempo acumulado supere bem o de outros módulos, com tempos bem menores e menos sensíveis ao volume

de dados.

Esse raciocínio também se aplica para analisarmos os tempos do gerenciador de iterações incorporando o algoritmo 2 (gráficos da seção B.6). Nesse caso, ressalvamos que o próprio algoritmo também é bastante sensível ao aumento de fragmentos, e portanto, ao número de reads shotgun adicionados na montagem. Conseqüentemente, mesmo com menos iterações realizadas pelo `cross_match` por causa do algoritmo 2 nos vários conjuntos de dados, o grande volume de dados faz com que esse mesmo algoritmo tome para si os níveis de tempo "economizado" pelas iterações (figura B.69). Isso explica um certo equilíbrio na comparação do tempo final da montagem OSS entre as versões com algoritmo 1 e algoritmo 2, considerando-se que os tempos entre um algoritmo e o outro vão de uma a quase três ordens de magnitude de diferença.

Algo a se chamar a atenção é que para o conjunto `DNA3M_375c40K_p450`, cujo tamanho do DNA é de 3 Mbp, o algoritmo 2 chega a montar o genoma, em tempo sensivelmente menor para cobertura shotguns menores que 7 X, conforme mostra figura B.47 na página 107, embora o tempo isolado do algoritmo 2 seja maior que o do algoritmo 1 em duas ordens de magnitude.

4.5 Considerações finais

Os testes mostraram que o algoritmo 1 tem menor tempo de execução que o algoritmo 2, o que já era esperado pela nossa análise de complexidade no capítulo 2.

Ainda que o algoritmo 2 seja mais lento para encontrar os scaffolds da montagem OSS, em geral ele demanda menos esforço de seqüenciamento e menos iterações na montagem. Em termos práticos, para os laboratórios de seqüenciamentos, seqüenciar menos significa economizar reagentes e tempo hora-homem nos processos laboratoriais.

Já para o algoritmo 1, vimos que o seu tempo é menor e também menos sensível ao tamanho do grafo, mas geralmente precisa de mais iterações e seqüenciamento. Todavia, por causa da redundância de seqüenciamento, há maior confiança nas bases do genoma decifrado, pois a quantidade de bases que cobrem a mesma região faz com que as bases acabem se confirmando. Ressaltamos que há outras maneira de se incrementar a qualidade da montagem sem aumentar a redundância de dados, por exemplo garantindo a qualidade das bases dos fragmentos seqüenciados em laboratório e aumentando a exigência para se considerar uma sobreposição entre duas seqüências de DNA.

Apesar de na maioria das vezes o algoritmo 2 montar os fragmentos e chegar a 100% do genoma, ele merece uma atenção especial para se tratar os casos em que o genoma fecharia se apenas um clone a mais fosse usado no scaffold e melhorar a contabilização do percentual do genoma desvendado em caso de formação de vários contigs, ao invés de um consenso.

Ter como resultado um consenso que cobre o genoma em várias vezes também não se pode dizer que foi uma montagem bem sucedida, portanto o algoritmo 1 também deve ser ajustado para reconhecer que o scaffold fechou um genoma circular.

Analisando o processo da montagem de fragmentos OSS como sendo algo composto por vários módulos e iterações, concluímos que um algoritmo scaffold bom é aquele que resulta numa montagem OSS com:

- menor esforço de seqüenciamento;
- menor número de iterações;
- menor tempo de execução em CPU.

Menor esforço de seqüenciamento e menor número de iterações significam exigir menos trabalho e recorrer menos vezes aos laboratórios de seqüenciamento, que agrega os custos de tempo e materiais de laboratório.

Tempo de execução em CPU é o tempo total necessário para se conseguir montar um genoma com o método OSS usando o recurso computacional. Este item é influenciado pelo número de iterações da montagem, portanto diminuir o número de iterações pode ajudar a diminuir o tempo de execução em CPU.

Usando os critérios expostos, a conclusão é que nenhum dos dois algoritmos propostos é absolutamente melhor, pois enquanto um ganha no quesito menor tempo, o outro ganha em menor esforço de seqüenciamento. As diferenças são bastante grandes principalmente para volume de dados maiores.

Isso tudo nos leva a refletir sobre como seria o algoritmo que integrasse todas as qualidades identificadas aqui, mas esse já seria um capítulo de outra dissertação.

Capítulo 5

Conclusões

O objeto principal desta dissertação, que é a montagem de fragmentos pelo método *Ordered Shotgun Sequencing* (OSS), não tenta montar o genoma em apenas um passo. Ela tem uma abordagem iterativa. Este método obtém um rápido panorama do genoma com poucas informações iniciais, só com pontas de clones e alguns reads shotgun. Assim, no início do projeto já se pode ter uma idéia de como grupos de reads se ligam e se posicionam um em relação ao outro através do *scaffold*. É como uma montagem virtual que vai se tornando real a cada iteração, por causa das novas informações produzidas e agregadas, passo a passo, à montagem.

Pelo fato do método OSS ser iterativo, construímos um ambiente de simulação que gerenciasse o processo e substituísse a intervenção dos laboratórios de seqüenciamento na obtenção de novos dados para o passo seguinte da montagem OSS por um módulo computacional. Com isso foi possível testar os algoritmos estudados, medir a eficiência e comparar as montagens utilizando as soluções projetadas.

Nossa idéia inicial era testar os algoritmos propostos para encontrar scaffold usando o ambiente simulação OSS com três tipos de dados:

Artificiais Gerados artificialmente pelo módulo gerador de dados, sem introdução de erros encontrados normalmente em dados vindo de laboratórios de seqüenciamento.

Semi-reais Dados preparados a partir de dados reais do genoma da *Xylella fastidiosa* [18]. Com o genoma completo, temos o consenso do DNA e as coordenadas dos clones e reads shotguns usados para se obtê-lo. O que se pretende é que a partir desses dados se gerem as seqüências de pontas de clones e reads shotgun, como fazemos com os dados artificiais, no gerador de dados do ambiente de simulação OSS descrito no capítulo 3. Com isso retiramos os erros de leitura dos dados e podemos usar o gerenciador de iterações para executar o teste sobre esses dados.

Reais Dados totalmente provenientes do Projeto Genoma *Xylella fastidiosa*, reads com erros comuns em montagem de fragmentos de DNA como erro das bases, repetições em partes do genoma, etc.

Esperávamos, primeiramente, fazer os testes com dados artificiais para testar a robustez dos algoritmos e refinar os códigos, para depois alterá-los de forma que lidassem com problemas que aparecem em genomas reais como repetições, erros de leitura e falta de cobertura.

No final, por questão de tempo, apenas foi possível concretizar os testes com dados artificiais produzidos pelo gerador de dados do ambiente de simulação. Uma característica destes dados é que os dados gerados eram de genomas circulares e foram produzidos de forma a garantir que toda a extensão do genoma fosse coberta. A opção por um DNA circular é explicada pelo fato dele ser um caso geral e com mais problemas que o linear que, por sua vez, pode ser considerado como um caso particular do circular.

Por causa dessa característica uniformizada de nossos dados, além de avaliar o desempenho de nossos algoritmos, nossos testes serviram também para detectar os problemas oriundos de genomas circulares para os dois algoritmos implementados, mesmo para dados artificiais. Esses problemas devem ser tratados antes de se pensar em testar os algoritmos com dados provenientes de um genoma real.

5.1 Contribuições

Foram propostos neste trabalho, dois algoritmos para encontrar scaffold a cada iteração da montagem OSS. Um é o algoritmo guloso também denominado algoritmo 1 e o outro é o algoritmo de maior componente conexa OSS, também referenciado como algoritmo 2. Ambos foram modelados como um problema de grafo e as complexidades deles são, respectivamente, $O(n+m)$ e $O(nm)$, sendo n o número de vértices e m o número de arestas.

Os testes aplicados apontaram alguns problemas que os algoritmos enfrentaram e estes são pontos bastante relevantes para um genoma circular. Para resolver este problemas é necessário que os algoritmos consigam :

- determinar o tamanho do genoma;
- evitar que o scaffold expanda mais que o necessário, isto é, muito mais que o tamanho do genoma;
- lidar com coordenadas num mapeamento circular, levando em conta o fato de que duas posições que diferem por um múltiplo inteiro do tamanho do genoma são na verdade equivalentes.

Identificamos ainda características importantes que influenciam a montagem OSS e que um bom algoritmo que procura por scaffold deve proporcionar:

- menor esforço de seqüenciamento;
- menor número de iterações;
- menor tempo de execução em CPU.

Nenhum de nossos algoritmos contempla todos os itens acima. O algoritmo 1 tem consideravelmente o menor tempo de execução em CPU, que é o tempo total de máquina necessário para se conseguir montar um genoma com o método OSS. Em contrapartida, o algoritmo 2 demanda menor esforço de seqüenciamento e número de iterações, que significaria economia de tempo e custos nos processos laboratoriais.

Analisando, isoladamente, a soma do tempo de CPU do módulo scaffold, que é a implementação dos algoritmos propostos, a diferença de tempo do algoritmo 1 para o algoritmo 2 é em torno de duas ordens de magnitude. No entanto, pelo algoritmo 2 usar menos iterações na montagem OSS, o tempo total em CPU de todo o processo de montar os fragmentos não refletiu a mesma discrepância dos tempos isolados. Isso nos leva a intuir que diminuir o número de iterações pode colaborar para o decréscimo do tempo de execução em CPU. Da mesma forma, diminuir o impacto de tempo de outros módulos envolvidos no processo de montagem também pode ajudar. Algo nesse sentido foi feito com o módulo que monta o grafo OSS, descrito na seção 3.3.2. Por fim, vale mencionar que, em geral, o número de iterações diminui com o aumento da cobertura com reads shotgun inseridos nos dados iniciais para a montagem OSS.

Para que o ambiente de simulação funcione com dados reais, além dos algoritmos de scaffold tratarem dos problemas característicos dos dados, os seguintes módulos devem se adaptar:

- Monta Grafo OSS. Este deve ter um parâmetro de tolerância de erros nas sobreposições encontradas. Como este foi baseado num programa que já fazia isso, no fundo este módulo já está praticamente preparado para lidar com erros. No entanto, como o trabalho original [3] que contribui para este módulo era baseado no método Shotgun de montagem de fragmentos de DNA, algumas adaptações para clones ainda seriam necessárias;
- construtor do mapa dos dados. Como para dados reais não há um arquivo de coordenadas, apenas será possível desenhar o scaffold encontrado na iteração, baseado na informação do caminho encontrado no grafo OSS, além de posicionar os clones no mapa apenas aproximadamente, pois os seus tamanhos serão apenas estimados;

- o seqüenciador deve ser capaz de tomar como base o scaffold encontrado e simular o seqüenciamento, retornando o consenso do caminho.

O `cross_match` é um programa membro do software de montagem `phrap` [10] que já lida com sobreposições imperfeitas entre fragmentos ao comparar as seqüências, portanto ficaria a cargo de um outro programa analisar e filtrar estes resultados e montar o grafo OSS apropriado, que no nosso caso seria o Monta Grafo.

Vale a pena mencionar aqui uma formulação teórica que tenciona modelar melhor o que se busca na montagem de um genoma circular. Esta formulação foi elaborada com base na experiência obtida com os dois algoritmos estudados e se parece com o algoritmo 2, já que também procura uma solução em uma componente conexa:

> Dado um mapeamento de pontas de clones e reads shotgun, determinar um subconjunto B de clones a seqüenciar, tal que B e certos grupos de reads shotgun e pontas cubram a mesma região do mapa, mas a soma dos tamanhos dos clones em B seja mínima.

Na situação apresentada pela figura 5.1, com a formulação acima, gostaríamos que a solução fosse um conjunto B composto pelos clones (R, T, V).

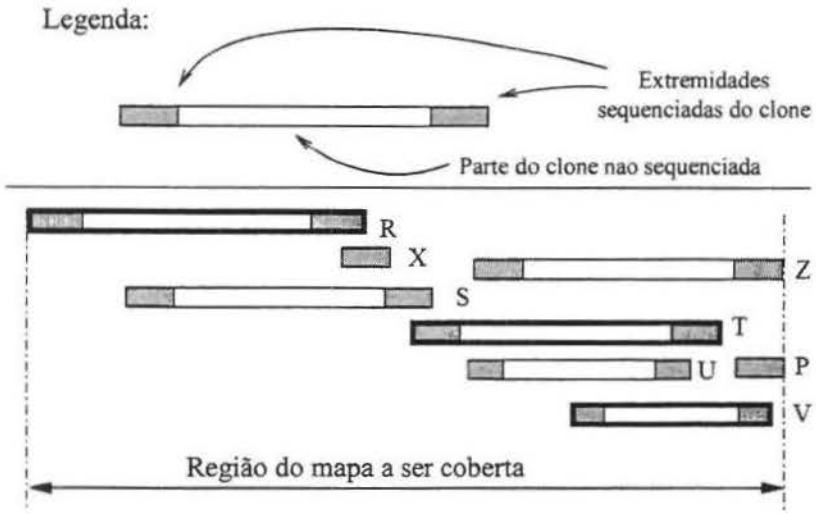

Figura 5.1: Mapa de distribuição dos fragmentos mostra os clones (contorno em negrito) que precisariam ser seqüenciados pela nova formulação teórica do problema.

Por último, destacamos que os algoritmos são sensíveis aos dados, já que se tratam de heurísticas. Os dados gerados apenas nos deram um panorama geral e inicial do comportamento dos dois algoritmos e revelam uma certa sensibilidade aos dados de entrada, pois vários gráficos apresentam picos destoantes em suas curvas. Eles nos indicam que seria

interessante testar a sensibilidade gerando mais instâncias para os mesmos parâmetros de geração de dados.

5.2 Trabalhos futuros

Além dos problemas encontrados exclusivamente por causa da característica circular do genoma (5.1), consideramos que algumas questões relacionadas aos algoritmos propostos neste trabalho também mereceriam um estudo mais aprofundado no futuro.

- Contabilização da extensão do genoma quando a montagem resultar em vários contigs. No algoritmo 2 estes seriam as várias componentes conexas OSS e os contigs seriam desvendados no final da montagem OSS. Já o algoritmo 1 apenas trabalharia em um dos contigs e pararia.
- Estudo da situação em que um fragmento e seu complemento reverso encontram-se num mesmo caminho. Atualmente isto é evitado, mas não está claro que a solução adotada seja a melhor alternativa.
- Proposta para solução de um grafo não mapeável.
- Identificação de grafos mapeáveis para dados reais e com tamanhos de clones aproximados. Uma idéia inicial foi esboçada na página 27, capítulo 2.
- Outros critérios que poderiam ser usados para escolher a componente conexa OSS na qual se deve procurar o scaffold. Atualmente o algoritmo 2 escolhe a componente com mais membros.
- Elaboração de critérios gulosos que pudessem resultar num scaffold que demandasse menos esforço de seqüenciamento no algoritmo 1.
- Exploração da nova formulação para o algoritmo 2.

Considerando que vários problemas enfrentados por Cerqueira [3] com dados reais nós também teríamos que encarar, seria interessante incorporar suas implementações em trabalhos futuros. Da mesma forma, idéias para outras formulações de algoritmos scaffold poderiam se somar às apresentadas em trabalhos onde a montagem OSS foi empregada com sucesso para projetos genoma reais como o da *Drosophila* [12] e da *Xanthomonas axonopodis* pv. *citri* [17].

Esperamos assim, com as questões abordadas sobre a montagem de fragmentos de DNA pelo método OSS, ter lançado uma luz sobre futuras pesquisas neste tópico, bem como ter contribuído para a pesquisa em Biologia Computacional.

Apêndice A

Tabelas

Nesse apêndice listamos as tabelas de resultados explicadas e referenciadas na seção 4.2.

A.1 Resultados com Algoritmo 1

Os resultados das tabelas contidas aqui são analisados na seção 4.2.1.

				DNA100K_13c40K_p500							
				Penúltima iteração			**Última** iteração			**Total Seqüenciado**	
cobertura	**# reads shotgun**	**iterações**	**Tempo (s)**	#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	2	1,07	3	3	40,82%	6	4	101,60%	3	120041
1	222	2	3,40	4	3	41,12%	9	4	102,35%	3	120041
2	444	2	6,38	14	2	43,67%	7	4	101,90%	3	120041
3	667	2	11,03	7	3	41,91%	14	4	103,97%	3	120041
4	889	2	16,37	7	3	42,24%	25	4	107,97%	3	120041
5	1111	2	21,65	30	3	49,87%	48	8	194,09%	5	199251
6	1333	1	16,37	-	-	-	187	14	261,81%	5	199947
7	1556	1	21,48	-	-	-	113	6	158,73%	3	120446
8	1778	1	28,17	-	-	-	108	6	156,28%	3	120446
9	2000	1	37,27	-	-	-	396	19	454,96%	8	319103
10	2222	1	46,67	-	-	-	326	15	395,75%	7	279598

Tabela A.1: Resultados com algoritmo 1 para conjunto DNA100K_13c40K_p500.

DNA200K_25c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	4	4,72	6	4	90,46%	6	4	125,21%	7	280542
1	444	4	11,93	10	4	91,04%	6	4	125,21%	7	280542
2	3111	4	23,13	13	4	91,28%	6	4	125,21%	7	280542
3	18667	3	29,37	7	4	76,52%	23	4	116,74%	6	240091
4	57778	2	32,37	9	5	40,87%	48	10	137,04%	7	280004
5	131111	1	24,78	-	-	-	63	12	128,05%	6	240275
6	249333	1	34,92	-	-	-	56	11	107,46%	5	200080
7	423111	1	45,52	-	-	-	154	19	184,39%	8	320662
8	663111	1	60,13	-	-	-	432	33	353,42%	14	560373
9	980000	1	86,12	-	-	-	533	31	352,54%	13	520860
10	1384444	1	112,50	-	-	-	358	28	322,16%	13	520790

Tabela A.2: Resultados com algoritmo 1 para conjunto DNA200K_25c40K_p500.

DNA300K_38c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	5	14,82	6	4	96,17%	6	4	121,97%	10	400261
1	667	5	27,93	6	4	96,17%	6	4	121,97%	10	400261
2	4000	4	35,93	8	4	72,95%	16	6	110,43%	9	360750
3	22000	4	56,28	8	4	73,06%	14	6	110,13%	9	360750
4	66667	4	87,62	43	6	90,15%	4	2	100,92%	8	320407
5	150000	3	90,78	32	7	94,03%	15	4	109,07%	9	361078
6	284000	1	54,13	-	-	-	180	16	100,01%	6	241442
7	480667	3	163,22	121	13	99,96%	4	2	101,09%	8	320990
8	752000	1	99,50	-	-	-	174	16	100,01%	6	241256
9	1110000	1	153,93	-	-	-	194	16	115,67%	7	281221
10	1566667	2	362,40	170	14	100,00%	4	2	106,66%	7	281508

Tabela A.3: Resultados com algoritmo 1 para conjunto DNA300K_38c40K_p500.

DNA400K_50c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	7	43,67	6	4	97,30%	6	4	115,38%	14	559972
1	889	7	66,17	9	4	97,54%	6	4	115,38%	14	559972
2	4889	7	98,95	16	4	97,93%	6	4	115,38%	14	559972
3	13333	4	79,67	27	8	59,23%	41	10	106,88%	12	479990
4	27556	4	114,03	32	8	68,68%	24	8	104,36%	12	480132
5	48889	3	123,42	29	6	61,00%	69	15	130,42%	13	520556
6	78667	2	157,62	153	19	92,12%	36	4	107,87%	10	400030
7	118222	1	110,22	-	-	-	253	37	168,90%	15	599533
8	168889	1	146,05	-	-	-	407	42	232,70%	20	799968
9	232000	2	355,98	51	8	44,07%	110	24	159,31%	15	599915
10	308889	1	381,80	-	-	-	848	53	304,45%	23	919166

Tabela A.4: Resultados com algoritmo 1 para conjunto DNA400K_50c40K_p500.

DNA500K_63c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	7	66,75	6	4	98,24%	6	4	113,04%	17	680075
1	1111	6	68,35	16	12	92,97%	8	4	106,45%	16	639728
2	5778	8	159,67	6	4	99,95%	6	4	109,57%	17	680895
3	14000	3	103,87	37	12	95,22%	15	4	102,11%	14	559932
4	25778	3	142,25	21	12	94,15%	21	4	102,57%	14	559932
5	41111	3	189,85	21	6	97,89%	4	2	104,06%	13	521393
6	60000	2	180,58	85	14	60,98%	49	18	130,41%	16	640730
7	82444	2	239,23	41	10	42,12%	108	19	116,20%	14	561307
8	108444	1	189,25	-	-	-	234	38	158,01%	18	720751
9	138000	1	271,33	-	-	-	612	53	225,37%	23	920337
10	171111	1	542,40	-	-	-	566	50	222,74%	23	920337

Tabela A.5: Resultados com algoritmo 1 para conjunto DNA500K_63c40K_p500.

DNA600K_75c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	8	127,43	6	4	94,25%	6	4	103,95%	19	760929
1	1333	8	165,68	6	4	94,25%	6	4	103,95%	19	760929
2	6667	7	211,37	15	4	98,07%	4	2	103,96%	18	718839
3	16000	8	305,03	18	4	98,26%	4	2	104,12%	18	719418
4	29333	6	266,32	38	6	80,91%	50	11	113,34%	19	757546
5	46667	5	309,85	69	4	70,74%	46	12	107,28%	17	679401
6	68000	3	314,03	104	15	92,51%	72	6	110,98%	16	638025
7	93333	1	203,47	-	-	-	335	33	111,40%	14	559064
8	122667	1	251,68	-	-	-	438	39	131,09%	16	639675
9	156000	1	384,17	-	-	-	732	53	188,63%	22	878086
10	193333	1	841,85	-	-	-	900	61	225,51%	26	1037041

Tabela A.6: Resultados com algoritmo 1 para conjunto DNA600K_75c40K_p500.

DNA700K_88c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	12	264,47	6	4	95,21%	6	4	101,34%	23	922595
1	1556	11	291,02	9	4	90,31%	7	4	100,40%	22	881565
2	3556	11	359,20	10	4	91,03%	8	4	100,56%	22	882537
3	6000	10	452,57	21	4	90,88%	22	4	101,02%	22	881706
4	8889	8	478,20	21	4	94,26%	30	6	109,23%	21	840116
5	12222	5	392,73	30	9	87,44%	32	6	102,10%	19	759702
6	16000	3	429,30	117	11	97,10%	29	4	102,88%	17	680333
7	20222	3	537,47	56	10	99,99%	6	4	109,00%	18	721209
8	24889	1	331,35	-	-	-	729	67	199,98%	29	1159709
9	30000	1	515,00	-	-	-	718	66	211,38%	31	1240726
10	35556	1	1306,60	-	-	-	914	77	239,11%	34	1360945

Tabela A.7: Resultados com algoritmo 1 para conjunto DNA700K_88c40K_p500.

DNA800K_100c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	11	406,17	6	4	96,16%	6	4	102,28%	26	1038626
1	1778	10	391,68	8	4	92,61%	9	4	100,84%	25	999414
2	3556	10	484,37	21	4	93,20%	10	4	100,84%	25	999766
3	5333	8	489,30	17	6	92,01%	21	6	104,11%	24	958958
4	7111	5	423,03	51	14	96,17%	12	6	107,43%	23	919244
5	8889	4	402,98	66	12	78,72%	118	18	118,84%	24	957534
6	10667	1	186,60	-	-	-	324	41	102,27%	18	720201
7	12444	2	403,92	14	4	10,36%	254	42	101,55%	19	760126
8	14222	1	452,60	-	-	-	653	73	195,76%	34	1358082
9	16000	1	696,78	-	-	-	844	95	238,98%	41	1637224
10	17778	1	377,43	-	-	-	1244	108	297,55%	49	1957132

Tabela A.8: Resultados com algoritmo 1 para conjunto DNA800K_100c40K_p500.

DNA900K_113c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	14	571,57	8	6	97,49%	6	4	105,66%	29	1162052
1	2000	13	553,43	7	4	89,25%	10	6	101,90%	28	1121736
2	4000	11	573,67	16	9	90,35%	15	6	102,10%	28	1121625
3	6000	11	773,25	35	14	99,70%	6	4	106,36%	29	1160799
4	8000	8	628,42	82	18	85,50%	52	9	103,49%	26	1042287
5	10000	6	685,83	21	8	91,33%	24	8	107,00%	25	1000772
6	12000	4	600,13	137	16	69,89%	157	19	111,25%	24	961517
7	14000	1	351,43	-	-	-	556	65	143,94%	28	1121165
8	16000	1	579,88	-	-	-	496	53	124,77%	24	961300
9	18000	2	879,53	324	38	83,00%	124	17	120,49%	24	958898
10	20000	1	439,07	-	-	-	923	96	220,93%	42	1679900

Tabela A.9: Resultados com algoritmo 1 para conjunto DNA900K_113c40K_p500.

DNA1000K_125c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração #reads	Penúltima iteração #pontas	Penúltima iteração % coberto	Última iteração #reads	Última iteração #pontas	Última iteração % coberto	Total Seqüenciado #clones	Total Seqüenciado #bases
0	0	14	933,80	10	8	99,39%	4	2	100,72%	31	1243075
1	2222	13	858,75	11	8	97,72%	13	6	104,76%	32	1283006
2	4444	14	1199,05	14	8	99,52%	4	2	100,72%	31	1243075
3	6667	13	1348,57	16	8	99,57%	4	2	100,90%	31	1243174
4	8889	10	1226,93	10	6	95,97%	10	4	102,45%	30	1203446
5	11111	7	1054,90	59	12	96,01%	37	6	107,30%	28	1121490
6	13333	4	915,00	61	10	89,44%	64	10	108,52%	26	1039912
7	15556	2	853,62	273	29	64,78%	182	21	105,69%	23	920632
8	17778	2	1584,63	264	27	60,65%	298	41	143,32%	32	1282472
9	20000	1	412,10	-	-	-	587	54	119,91%	25	1002527
10	22222	1	495,47	-	-	-	727	70	144,84%	30	1203135

Tabela A.10: Resultados com algoritmo 1 para conjunto DNA1000K_125c40K_p500.

DNA3M_375c40K_p450

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração #reads	Penúltima iteração #pontas	Penúltima iteração % coberto	Última iteração #reads	Última iteração #pontas	Última iteração % coberto	Total Seqüenciado #clones	Total Seqüenciado #bases
0	0	41	73650,27	8	6	97,76%	6	4	100,00%	92	3678660
1	6667	39	61694,62	9	6	95,84%	13	8	100,47%	91	3639429
2	13333	38	67506,47	10	6	97,78%	9	6	100,82%	91	3638717
3	20000	31	53810,63	25	6	97,81%	13	6	101,55%	87	3480162
4	26667	26	55354,70	46	6	99,80%	6	4	101,77%	86	3441531
5	33333	11	24434,15	64	6	91,50%	127	16	102,13%	76	3038777
6	40000	11	24233,65	133	25	100,00%	6	4	102,06%	75	2997349
7	46667	4	8113,90	553	56	86,03%	172	30	106,45%	72	2877643
8	53333	3	3133,52	244	24	43,26%	948	109	122,26%	82	3280626
9	60000	1	1697,45	-	-	-	2027	213	159,05%	103	4122501
10	66667	1	2021,87	-	-	-	1597	181	133,19%	87	3482117

Tabela A.11: Resultados com algoritmo 1 para conjunto DNA3M_375c40K_p450.

A.2 Resultados com Algoritmo 2

Os resultados dessas tabelas são analisadas na seção 4.2.2.

				DNA100K_13c40K_p500							
				Penúltima iteração			**Última** iteração			**Total Seqüenciado**	
cobertura	**# reads shotgun**	**iterações**	**Tempo (s)**	#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	2	1,10	3	3	40,49%	5	4	101,62%	3	119679
1	222	3	5,00	6	4	41,49%	6	4	87,27%	3	119836
2	2222	3	8,67	6	2	80,24%	6	2	98,63%	3	119527
3	15333	2	12,13	19	4	81,29%	8	4	127,20%	4	158773
4	54222	1	9,23	-	-	-	21	6	122,14%	3	119527
5	138889	2	23,45	46	5	76,63%	28	4	123,21%	4	159489
6	294667	3	38,33	49	8	43,34%	49	8	99,43%	2	79892
7	552222	2	38,03	166	12	57,54%	17	6	180,04%	3	119135
8	947556	2	48,28	165	13	57,49%	19	6	180,12%	3	119135
9	1522000	3	69,32	113	12	79,99%	4	2	139,12%	2	79109
10	2322222	1	50,77	-	-	-	272	25	100,42%	0	0

Tabela A.12: Resultados com algoritmo 2 para conjunto DNA100K_13c40K_p500.

DNA200K_25c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	3	3,17	7	6	77,74%	7	6	115,31%	8	320119
1	444	3	10,32	10	8	94,76%	6	4	103,27%	8	320463
2	3111	2	15,03	21	8	77,71%	6	4	113,08%	6	238434
3	18667	1	12,38	-	-	-	55	12	116,74%	6	239195
4	57778	3	43,83	56	10	1,08%	6	0	98,87%	5	199170
5	131111	1	30,03	-	-	-	166	19	136,30%	6	239834
6	249333	2	68,92	58	5	37,31%	30	9	116,46%	6	239583
7	423111	1	55,75	-	-	-	402	37	110,74%	2	79927
8	663111	2	115,45	110	8	20,32%	435	38	118,40%	1	39847
9	980000	3	167,07	187	19	53,81%	44	5	110,92%	2	80803
10	1384444	2	181,05	527	39	98,49%	12	2	119,80%	1	40064

Tabela A.13: Resultados com algoritmo 2 para conjunto DNA200K_25c40K_p500.

DNA300K_38c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	4	8,70	5	4	78,19%	5	4	101,15%	9	361044
1	667	4	28,15	5	4	89,29%	5	4	95,85%	9	360768
2	4000	3	33,55	16	6	85,02%	6	4	102,81%	9	360505
3	22000	3	53,42	35	8	86,96%	16	4	103,95%	9	360298
4	66667	5	90,53	4	2	15,84%	4	2	99,96%	8	321119
5	150000	1	49,75	-	-	-	163	23	100,00%	7	281716
6	284000	2	117,50	112	6	13,28%	285	33	111,60%	5	200966
7	480667	2	158,77	4	2	13,23%	185	21	100,11%	6	240279
8	752000	2	190,23	231	18	40,81%	394	37	111,17%	3	120382
9	1110000	3	354,13	21	3	11,49%	435	44	105,65%	4	160652
10	1566667	2	385,27	20	4	15,72%	610	63	106,76%	3	120174

Tabela A.14: Resultados com algoritmo 2 para conjunto DNA300K_38c40K_p500.

DNA400K_50c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	6	36,82	5	4	89,33%	5	4	106,09%	14	559386
1	889	7	62,42	11	6	57,96%	11	6	96,68%	14	559618
2	4889	7	112,83	11	6	0,22%	3	0	97,32%	13	519099
3	13333	3	76,90	33	7	74,97%	45	9	107,38%	12	479699
4	27556	2	83,97	71	12	42,94%	52	15	105,62%	11	439497
5	48889	1	71,68	-	-	-	183	27	108,05%	10	399484
6	78667	2	165,97	106	10	38,69%	127	19	109,44%	11	438642
7	118222	2	241,40	99	7	18,52%	373	43	105,10%	7	280335
8	168889	2	307,32	213	10	19,41%	622	59	105,10%	3	119267
9	232000	2	453,65	209	12	19,43%	718	67	105,24%	2	80146
10	308889	2	626,30	126	16	11,38%	944	74	108,55%	1	39859

Tabela A.15: Resultados com algoritmo 2 para conjunto DNA400K_50c40K_p500.

DNA500K_63c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	6	65,85	7	6	83,38%	7	6	98,05%	16	639513
1	1111	4	52,13	14	8	84,74%	17	8	106,96%	16	639821
2	5778	4	87,77	15	12	87,90%	10	6	101,75%	16	639683
3	14000	2	96,27	50	24	88,50%	16	6	102,11%	14	560496
4	25778	2	119,93	81	22	60,08%	20	12	104,83%	15	599885
5	41111	2	166,88	66	16	30,92%	281	35	101,77%	13	520935
6	60000	2	258,93	46	2	5,32%	310	34	104,73%	11	439942
7	82444	3	455,82	110	12	23,79%	544	46	106,18%	7	280379
8	108444	3	651,23	34	6	2,51%	610	60	102,14%	9	361620
9	138000	3	570,92	160	17	19,56%	81	4	106,60%	2	79483
10	171111	3	902,00	52	9	19,56%	79	4	106,60%	3	119526

Tabela A.16: Resultados com algoritmo 2 para conjunto DNA500K_63c40K_p500.

DNA600K_75c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	8	183,37	3	2	94,25%	3	2	97,65%	18	718905
1	1333	6	100,97	7	4	77,54%	11	8	101,73%	18	718903
2	6667	5	156,10	22	8	90,08%	12	4	100,88%	17	679560
3	16000	4	168,07	12	4	71,61%	29	12	108,17%	18	719471
4	29333	3	191,12	29	7	21,29%	60	12	104,23%	19	758728
5	46667	1	135,32	-	-	-	361	47	104,34%	15	598282
6	68000	2	314,48	90	13	30,84%	272	27	103,38%	13	519106
7	93333	2	494,93	70	10	10,37%	827	72	105,25%	8	319006
8	122667	2	597,65	303	14	18,42%	704	58	105,61%	7	278917
9	156000	2	832,75	140	8	8,73%	1266	102	106,57%	3	120484
10	193333	3	1699,30	229	26	13,51%	1259	106	104,57%	2	80303

Tabela A.17: Resultados com algoritmo 2 para conjunto DNA600K_75c40K_p500.

DNA700K_88c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	9	201,27	5	4	90,90%	5	4	101,39%	21	840562
1	1556	8	172,90	8	4	61,55%	12	8	101,74%	22	882265
2	3556	7	249,00	22	8	82,67%	21	8	102,79%	22	880557
3	6000	10	809,52	4	0	0,14%	3	0	98,75%	20	800710
4	8889	5	410,50	19	8	54,22%	19	8	99,93%	22	880957
5	12222	2	284,08	255	29	51,03%	213	31	108,26%	20	801033
6	16000	1	279,52	-	-	-	516	50	102,88%	14	559267
7	20222	2	612,90	170	7	8,61%	771	73	102,60%	11	440655
8	24889	3	1123,62	2	1	0,11%	1384	124	100,06%	4	160794
9	30000	2	1147,05	168	8	8,61%	1360	116	102,60%	4	159671
10	35556	2	2053,08	161	5	8,64%	1350	121	102,60%	4	158874

Tabela A.18: Resultados com algoritmo 2 para conjunto DNA700K_88c40K_p500.

DNA800K_100c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	10	305,28	7	6	92,41%	7	6	102,56%	26	1036570
1	1778	9	319,47	11	8	89,66%	8	6	101,54%	27	1078646
2	3556	8	371,47	17	8	89,31%	32	8	104,04%	26	1039119
3	5333	7	435,67	13	6	57,51%	19	10	106,65%	26	1038470
4	7111	3	394,13	138	25	96,78%	42	2	100,00%	20	799844
5	8889	2	393,90	54	3	3,17%	298	48	101,15%	20	799041
6	10667	5	1155,83	264	44	3,95%	264	44	94,00%	23	917725
7	12444	2	765,20	172	11	12,45%	990	97	103,87%	11	439497
8	14222	3	1387,37	356	38	20,51%	1220	99	104,63%	6	239959
9	16000	2	1475,15	339	29	15,32%	1506	128	104,33%	4	160106
10	17778	2	1276,40	200	24	13,86%	1729	148	103,97%	3	120205

Tabela A.19: Resultados com algoritmo 2 para conjunto DNA800K_100c40K_p500.

DNA900K_113c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	10	539,20	7	6	93,58%	5	4	101,83%	28	1119310
1	2000	9	500,52	7	6	90,51%	9	6	100,00%	29	1158741
2	4000	7	563,58	13	8	98,10%	3	2	101,53%	27	1079888
3	6000	6	648,62	16	4	91,45%	14	6	100,77%	27	1081251
4	8000	5	769,20	26	5	91,67%	26	5	99,98%	24	960976
5	10000	1	272,67	-	-	-	506	68	107,13%	23	920329
6	12000	2	669,38	151	13	14,69%	519	58	104,39%	18	719635
7	14000	3	1355,67	145	14	9,99%	1093	101	100,00%	11	440089
8	16000	2	1269,33	535	47	25,69%	1378	121	103,01%	6	239181
9	18000	2	1209,32	207	21	25,71%	1357	109	102,99%	9	359133
10	20000	2	1717,83	142	9	5,85%	2135	186	103,38%	2	80591

Tabela A.20: Resultados com algoritmo 2 para conjunto DNA900K_113c40K_p500.

DNA1000K_125c40K_p500

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	12	895,28	9	8	97,82%	5	4	101,30%	33	1321773
1	2222	11	764,17	7	4	30,85%	10	6	100,42%	31	1240114
2	4444	11	1013,52	23	8	98,82%	6	4	100,91%	33	1321269
3	6667	6	809,15	33	8	93,16%	25	7	101,95%	30	1201653
4	8889	7	1447,62	15	6	0,18%	5	0	99,79%	29	1161246
5	11111	2	605,60	354	52	72,90%	139	22	107,39%	25	1001918
6	13333	2	893,35	19	4	8,34%	720	75	103,47%	21	841145
7	15556	3	1788,52	111	12	7,85%	1087	100	103,10%	16	640694
8	17778	2	2251,90	155	11	5,61%	1829	154	103,30%	8	319932
9	20000	2	1697,10	102	5	3,81%	2368	205	103,18%	3	120294
10	22222	3	2884,68	132	7	4,89%	2242	191	100,05%	2	80235

Tabela A.21: Resultados com algoritmo 2 para conjunto DNA1000K_125c40K_p500.

DNA3M_375c40K_p450

cobertura	# reads shotgun	iterações	Tempo (s)	Penúltima iteração			Última iteração			Total Seqüenciado	
				#reads	#pontas	% coberto	#reads	#pontas	% coberto	#clones	#bases
0	0	37	72171,10	5	4	99,59%	3	2	99,66%	95	3798521
1	6667	29	54279,13	10	4	98,45%	5	4	100,44%	92	3681285
2	13333	23	53431,72	10	5	99,62%	3	2	100,74%	90	3596752
3	20000	16	26370,90	21	6	45,00%	23	10	100,25%	87	3477794
4	26667	10	28622,93	102	22	99,81%	3	2	99,99%	84	3354928
5	33333	1	2471,12	-	-	-	1241	182	100,94%	70	2799681
6	40000	1	3243,78	-	-	-	1982	215	100,63%	62	2479048
7	46667	2	7054,15	89	7	2,36%	4088	340	101,14%	40	1601305
8	53333	2	8354,28	202	12	2,40%	5330	455	101,14%	27	1077746
9	60000	4	17748,73	279	29	3,29%	7051	551	100,93%	8	318726
10	66667	2	10758,50	106	3	1,32%	7332	578	101,31%	8	321310

Tabela A.22: Resultados com algoritmo 2 para conjunto DNA3M_375c40K_p450.

Apêndice B

Gráficos

Esse apêndice abriga os gráficos de resultados analisados e referenciados nas seções 4.3 e 4.4.

B.1 Gráficos: tempo dos algoritmos scaffold

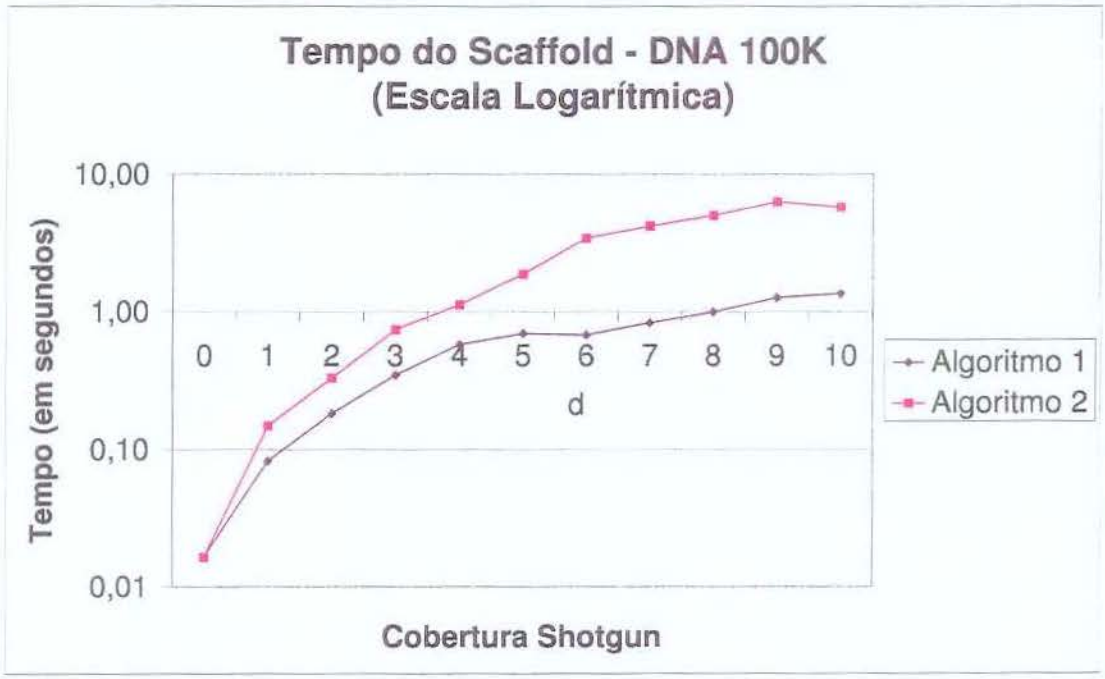

Figura B.1: Tempo dos algoritmos scaffold para conjunto `DNA100K_13c40K_p500`.

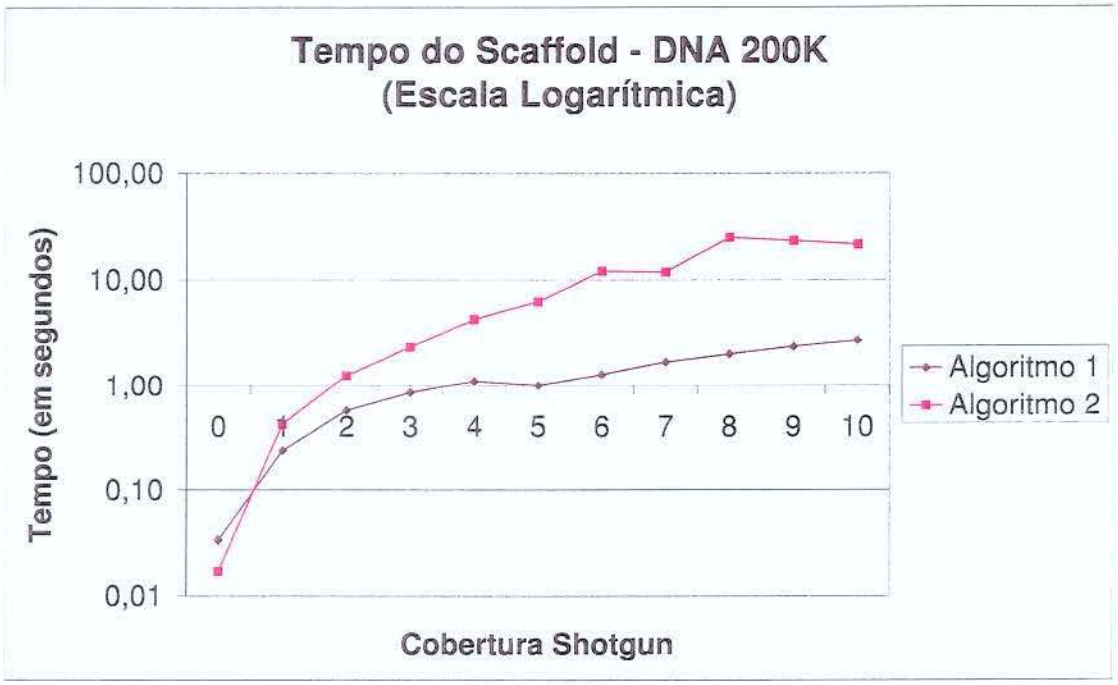

Figura B.2: Tempo dos algoritmos scaffold para conjunto `DNA200K_25c40K_p500`.

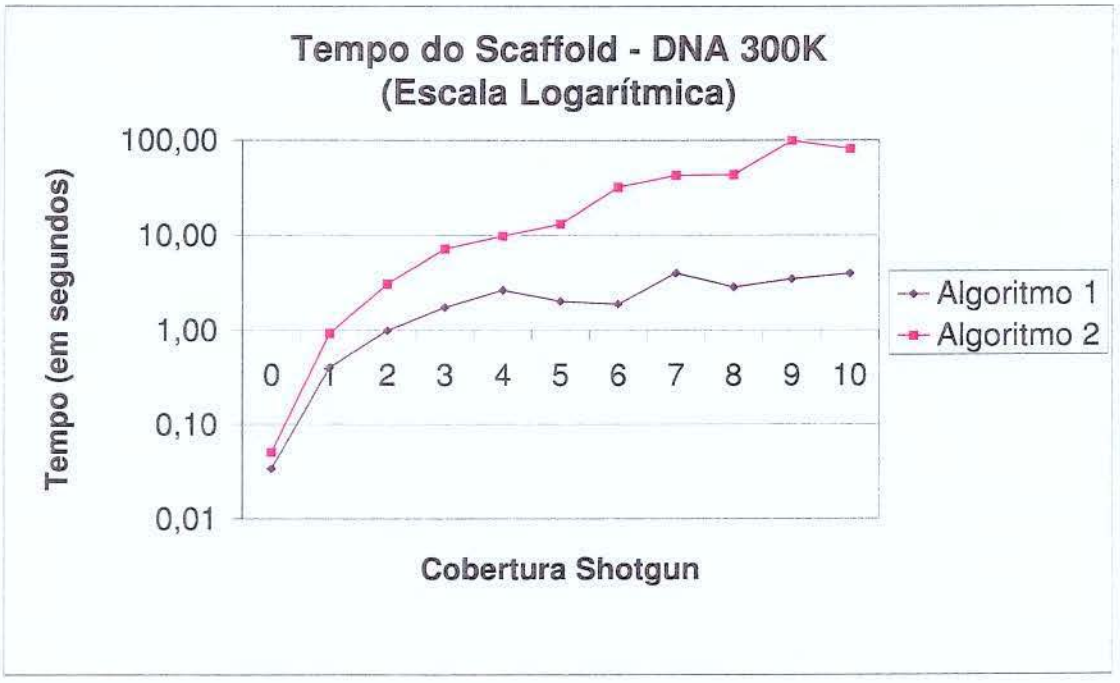

Figura B.3: Tempo dos algoritmos scaffold para conjunto `DNA300K_38c40K_p500`.

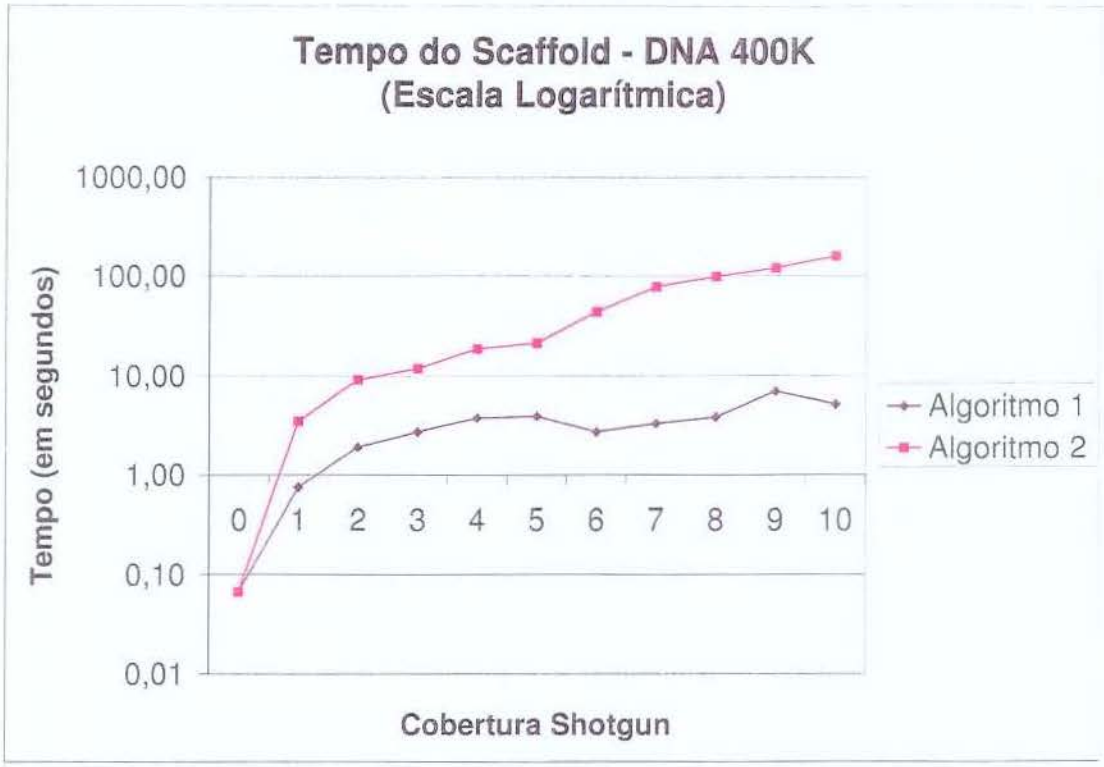

Figura B.4: Tempo dos algoritmos scaffold para conjunto DNA400K_50c40K_p500.

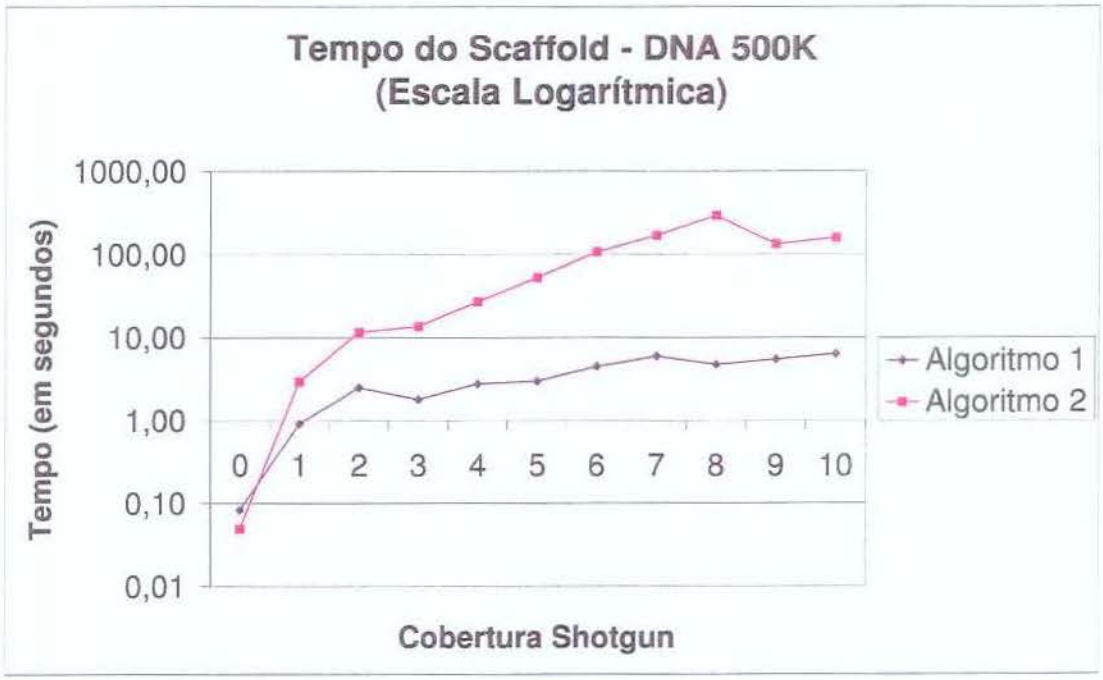

Figura B.5: Tempo dos algoritmos scaffold para conjunto DNA500K_63c40K_p500.

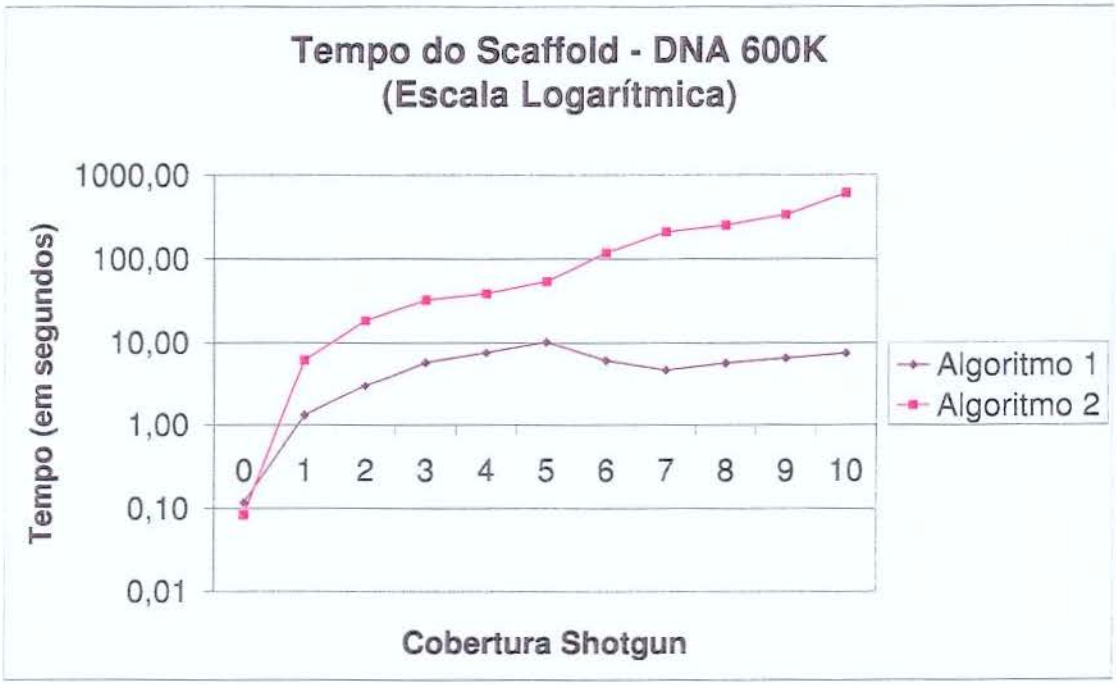

Figura B.6: Tempo dos algoritmos scaffold para conjunto `DNA600K_75c40K_p500`.

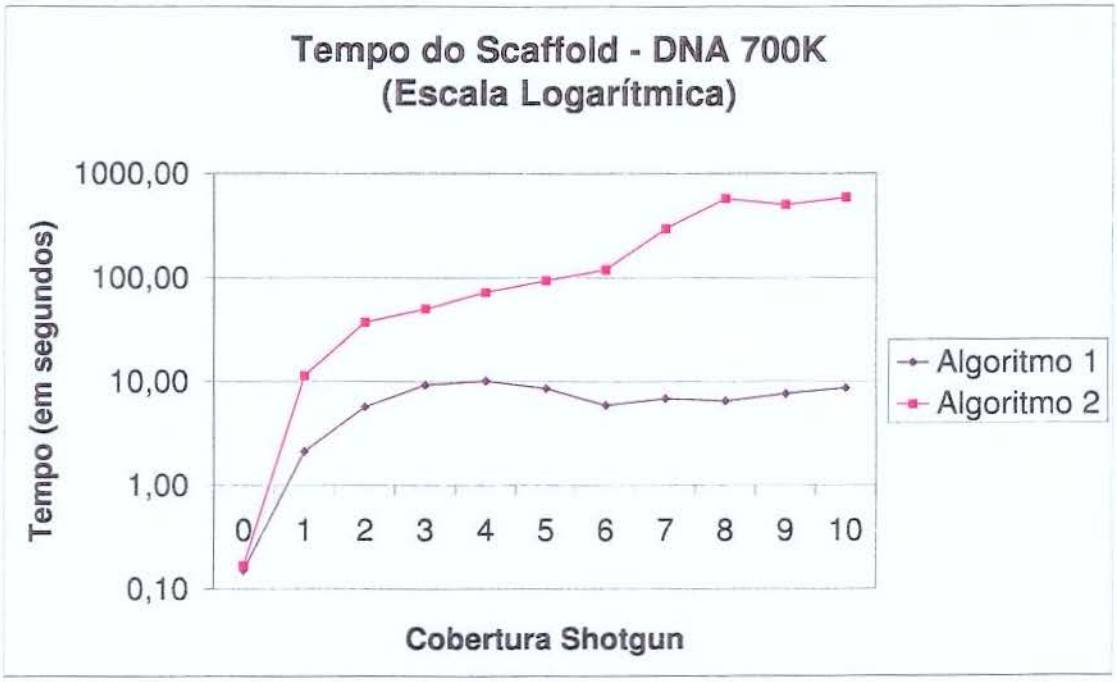

Figura B.7: Tempo dos algoritmos scaffold para conjunto `DNA700K_88c40K_p500`.

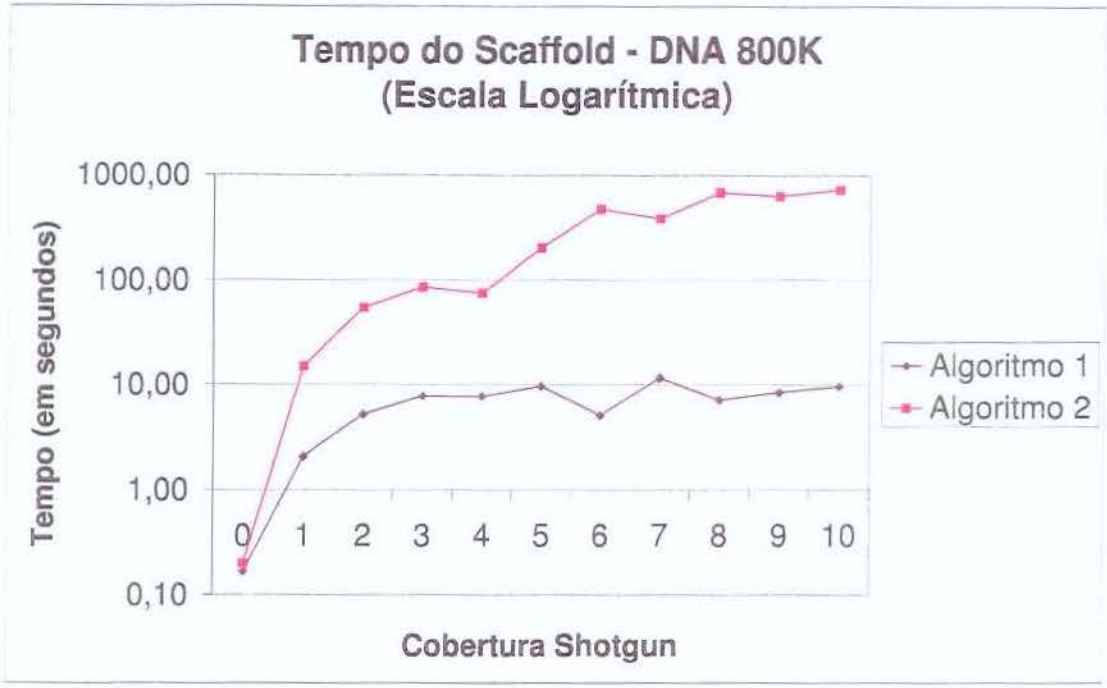

Figura B.8: Tempo dos algoritmos scaffold para conjunto DNA800K_100c40K_p500.

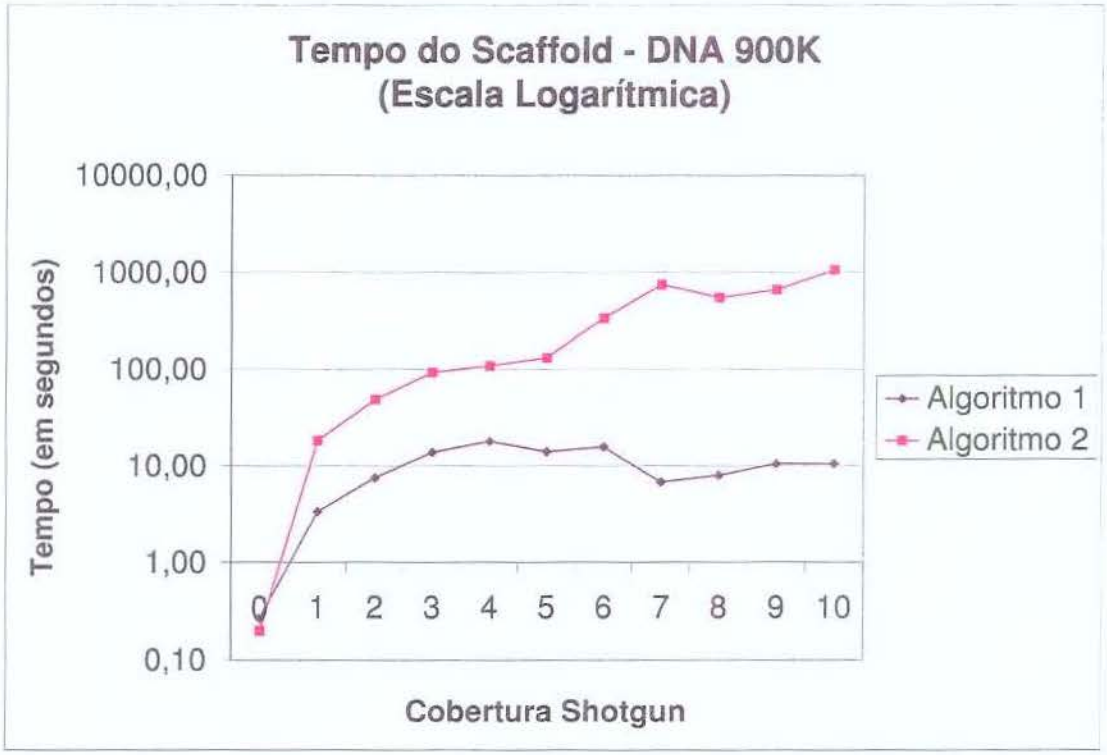

Figura B.9: Tempo dos algoritmos scaffold para conjunto DNA900K_113c40K_p500.

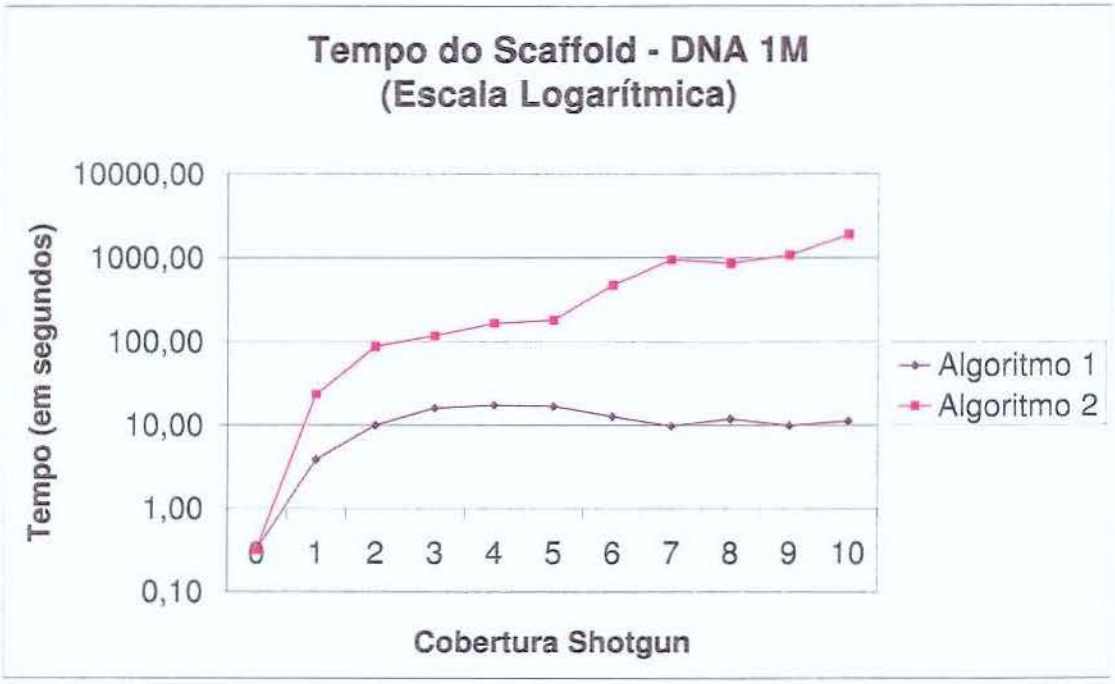

Figura B.10: Tempo dos algoritmos scaffold para conjunto `DNA1000K_125c40K_p500`.

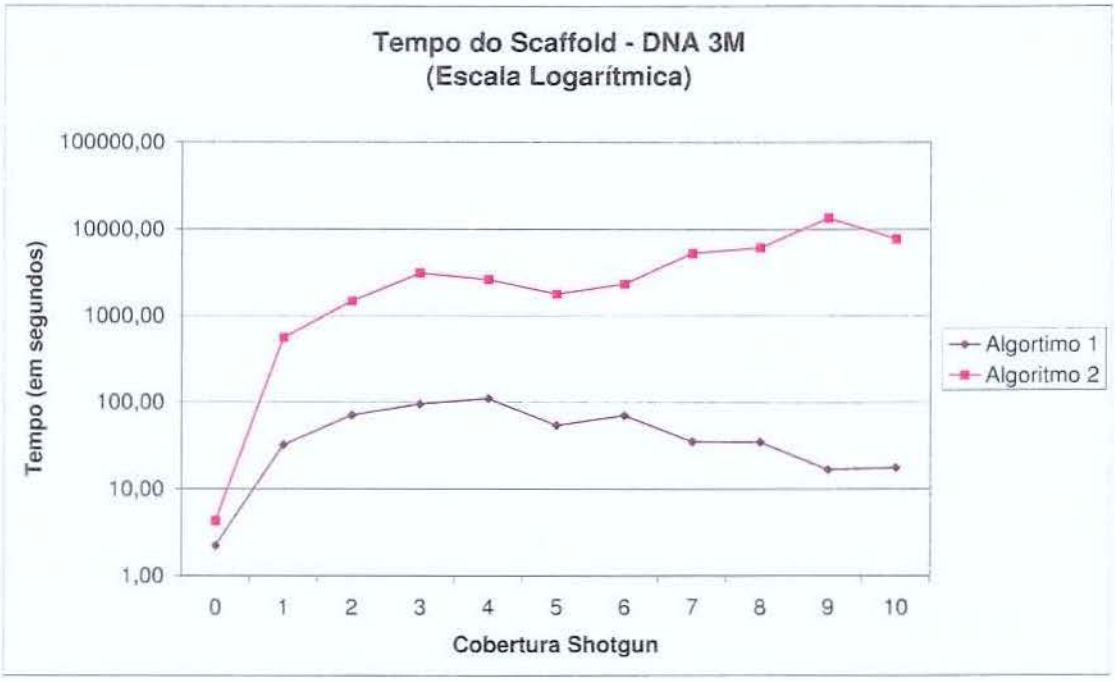

Figura B.11: Tempo dos algoritmos scaffold para conjunto `DNA3M_375c40K_p450`.

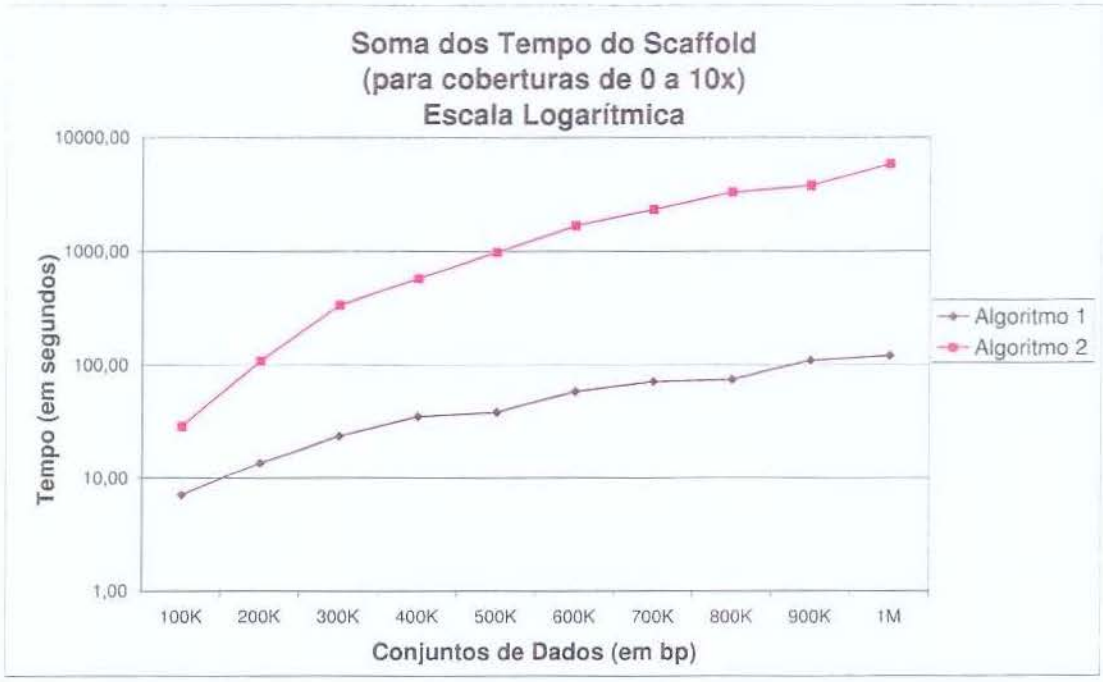

Figura B.12: Tempo total dos algoritmos scaffold para os conjuntos.

B.2 Gráficos: total de iterações executadas

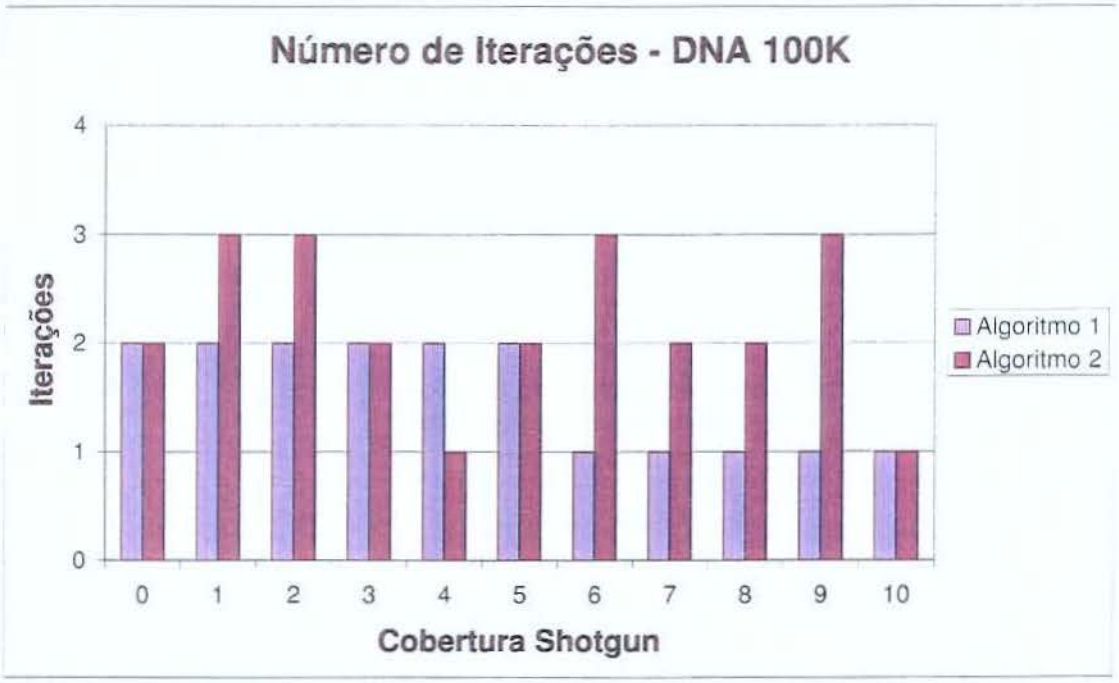

Figura B.13: Total de iterações para DNA100K_13c40K_p500.

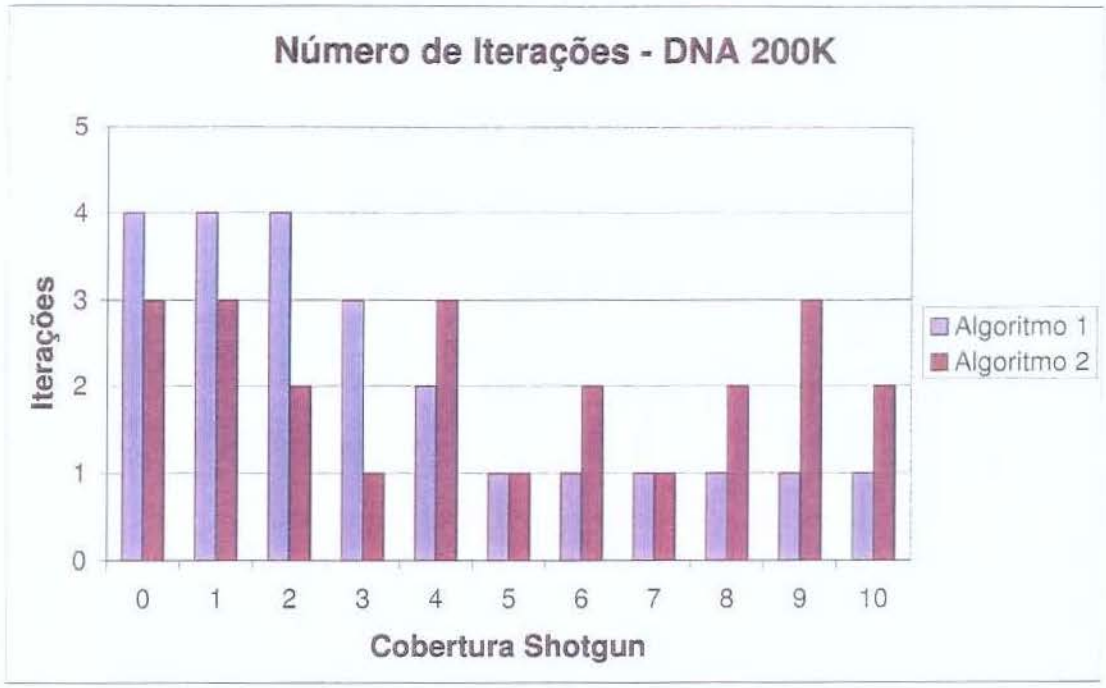

Figura B.14: Total de iterações para DNA200K_25c40K_p500.

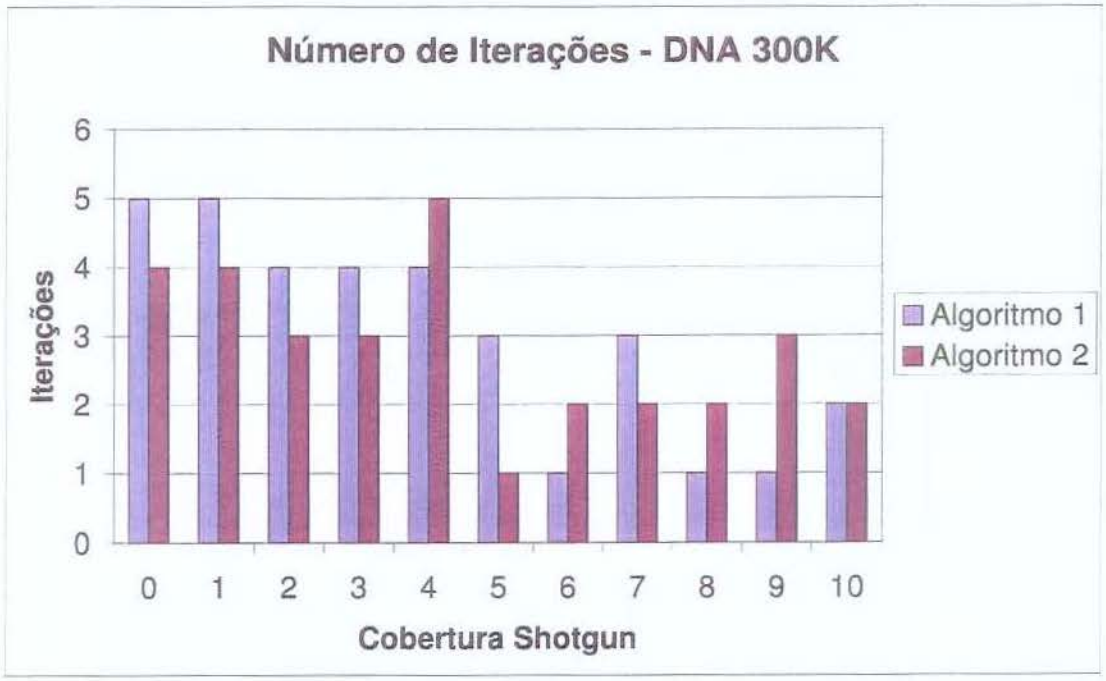

Figura B.15: Total de iterações para DNA300K_38c40K_p500.

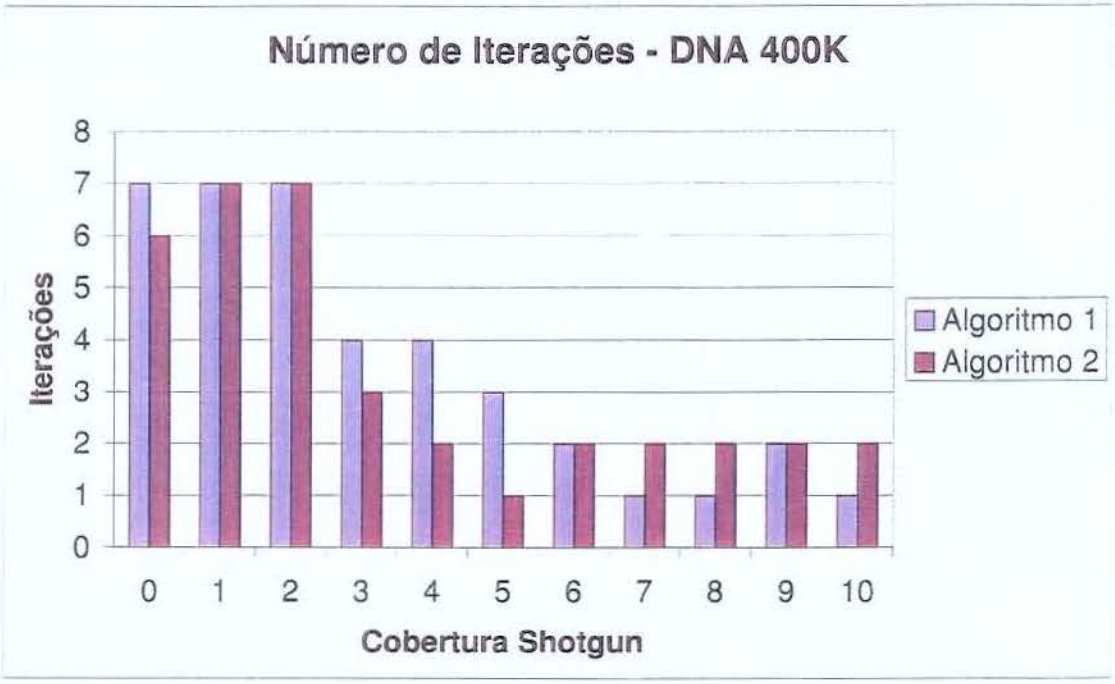

Figura B.16: Total de iterações para DNA400K_50c40K_p500.

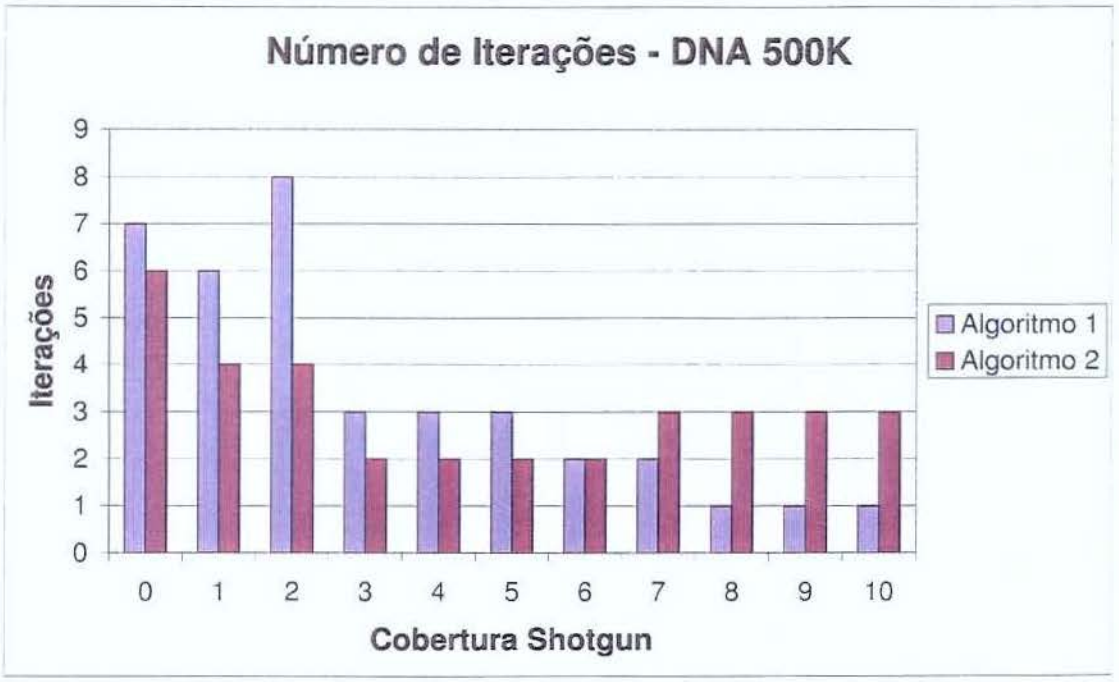

Figura B.17: Total de iterações para DNA500K_63c40K_p500.

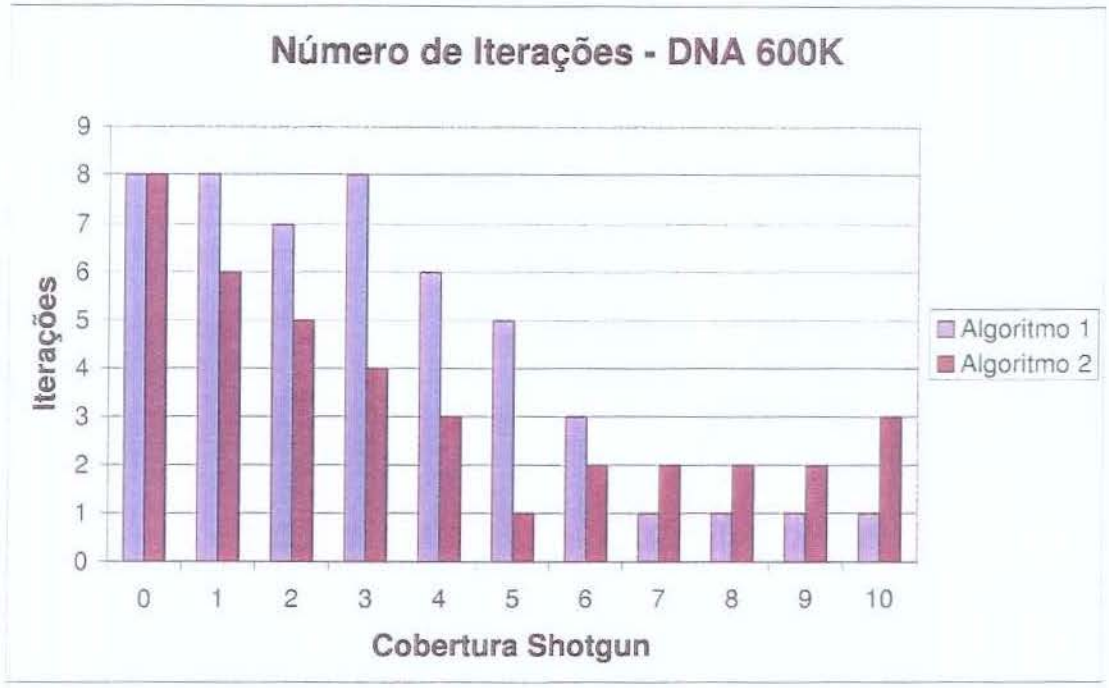

Figura B.18: Total de iterações para DNA600K_75c40K_p500.

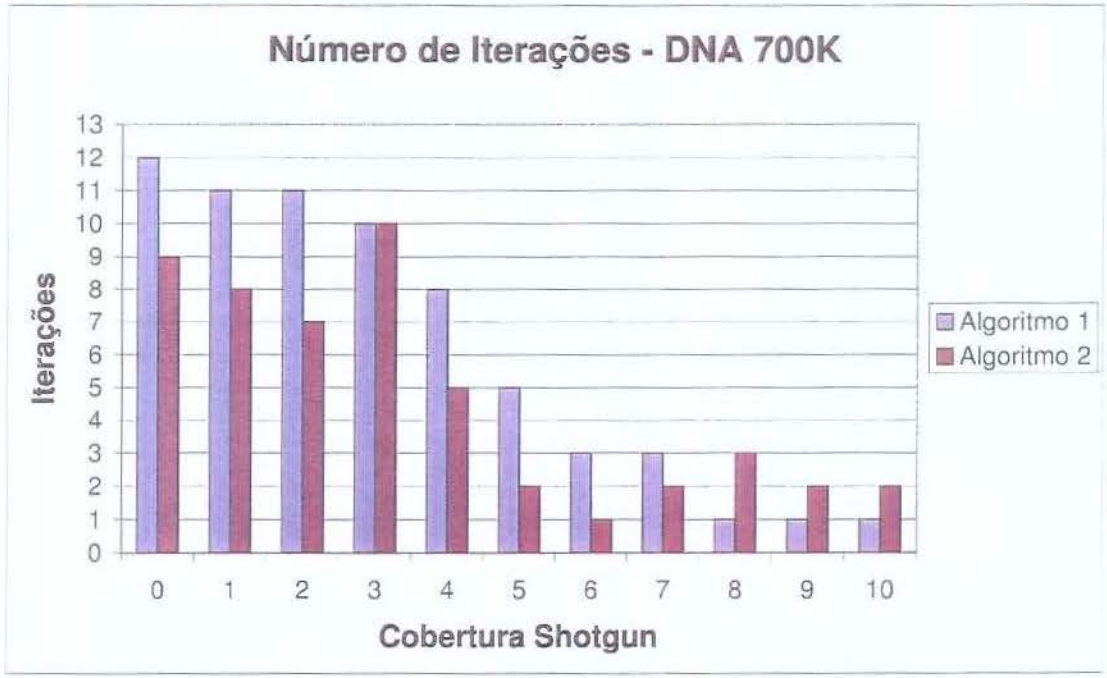

Figura B.19: Total de iterações para DNA700K_88c40K_p500.

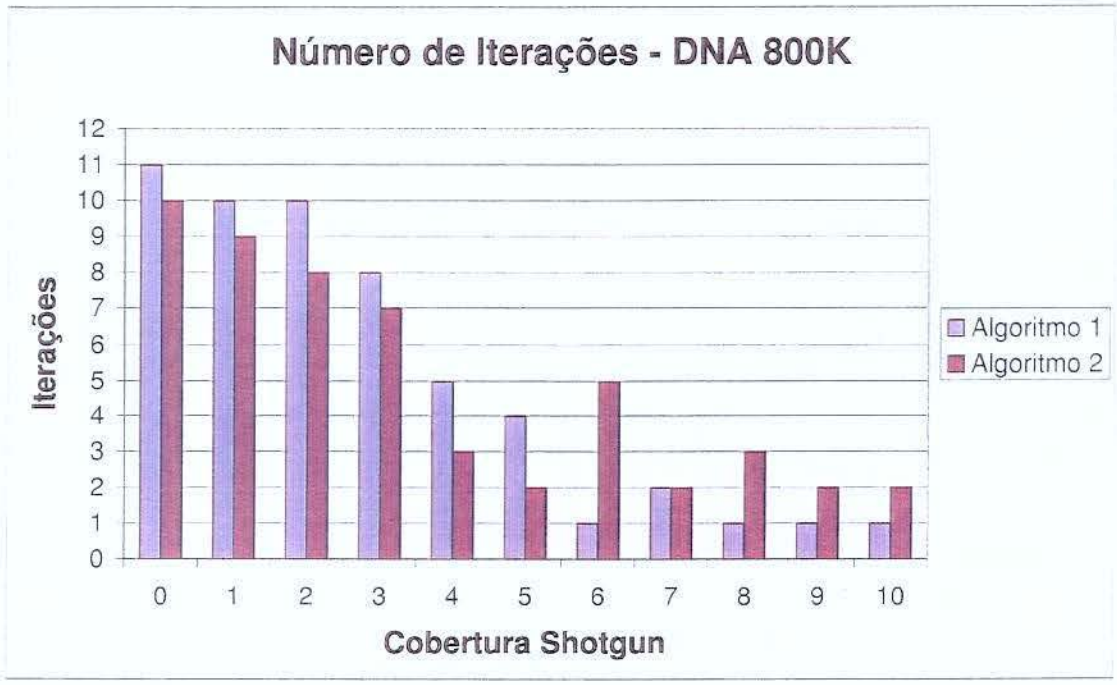

Figura B.20: Total de iterações para `DNA800K_100c40K_p500`.

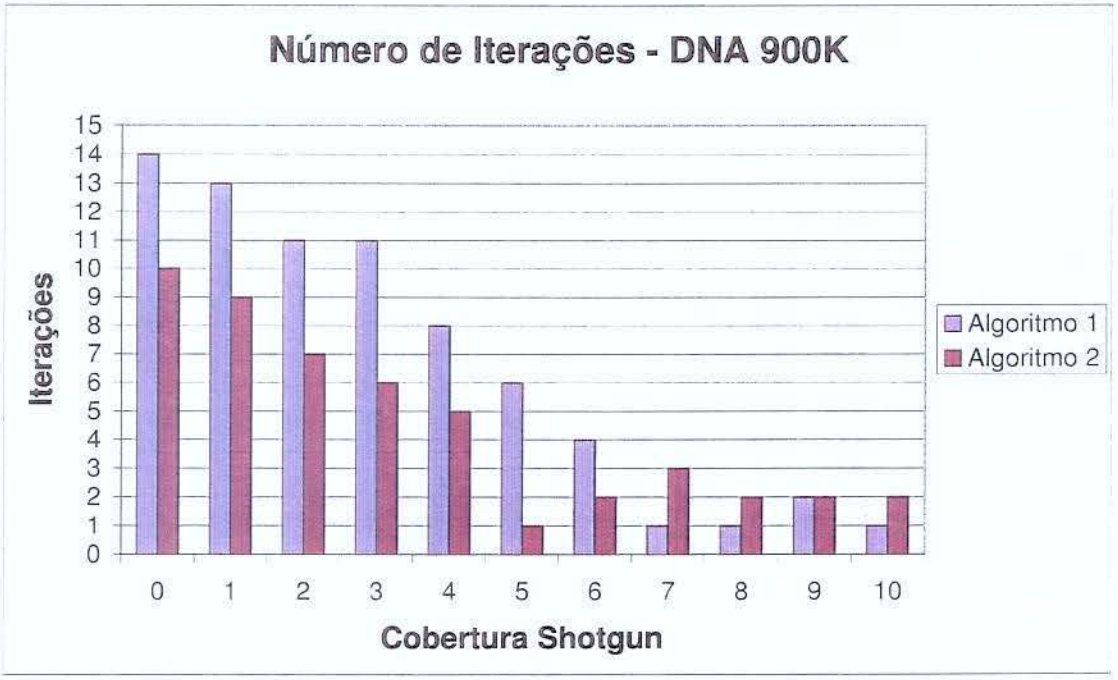

Figura B.21: Total de iterações para `DNA900K_113c40K_p500`.

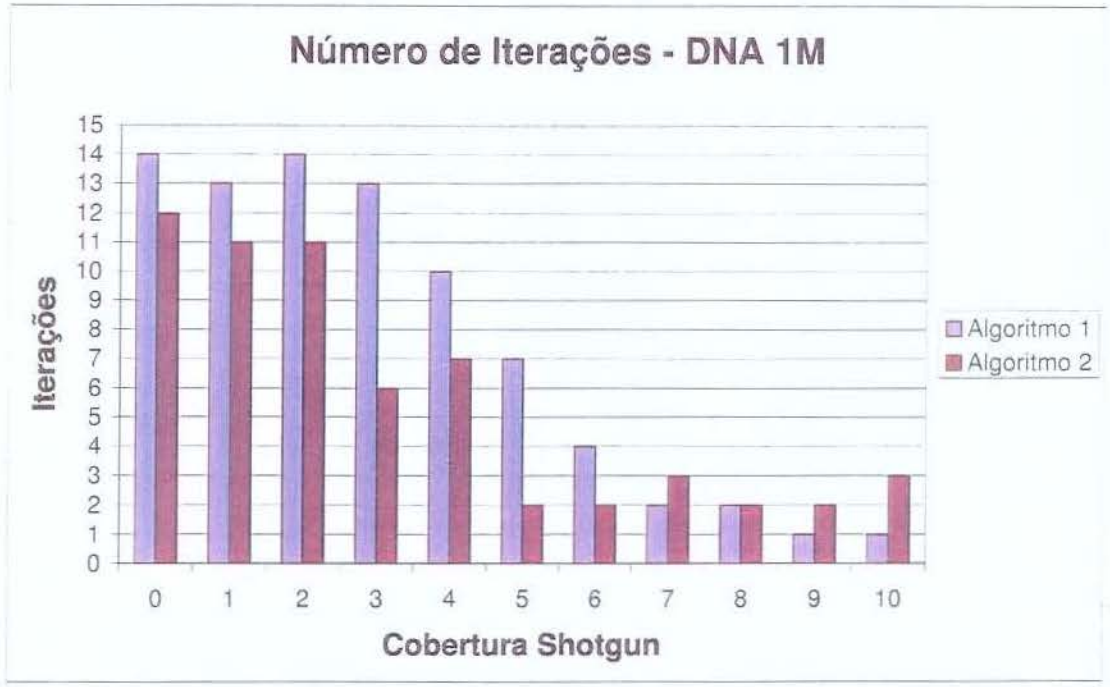

Figura B.22: Total de iterações para DNA1000K_125c40K_p500.

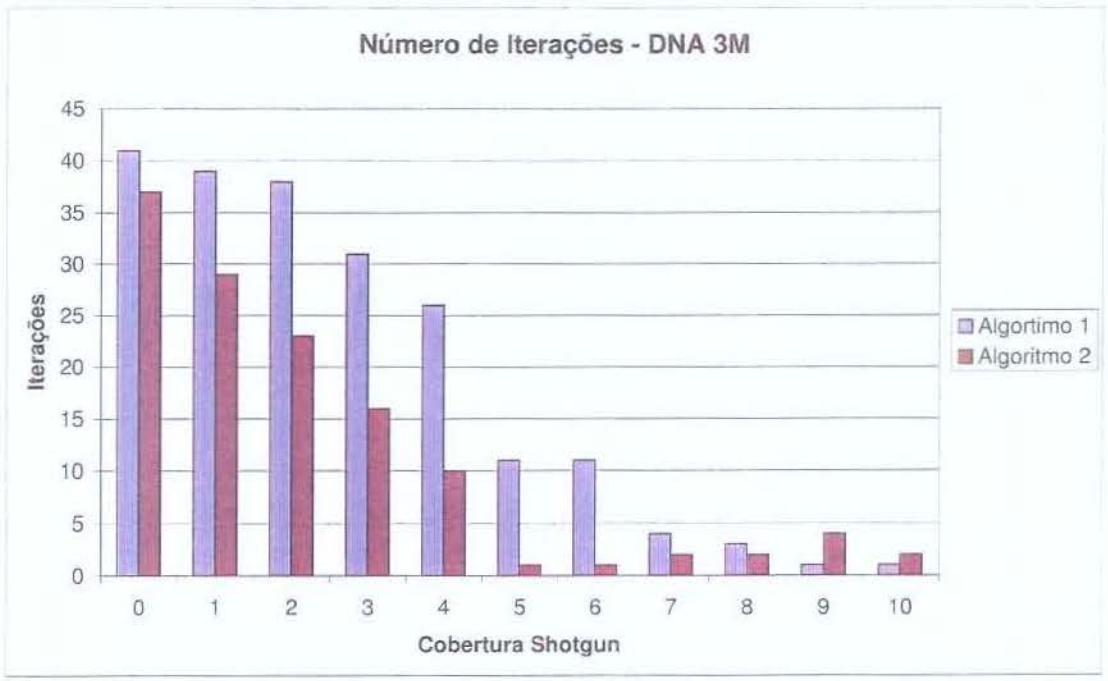

Figura B.23: Total de iterações para DNA3M_375c40K_p450.

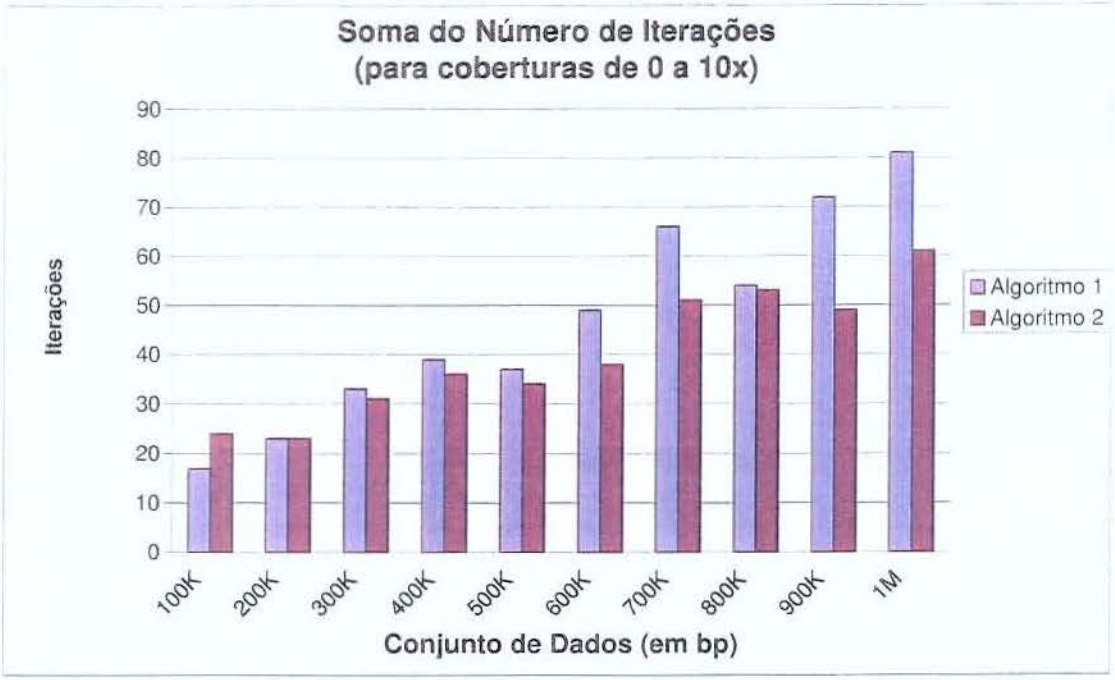

Figura B.24: Total geral de iterações para DNA de até 1Mbp.

B.3 Gráficos: esforço de seqüenciamento

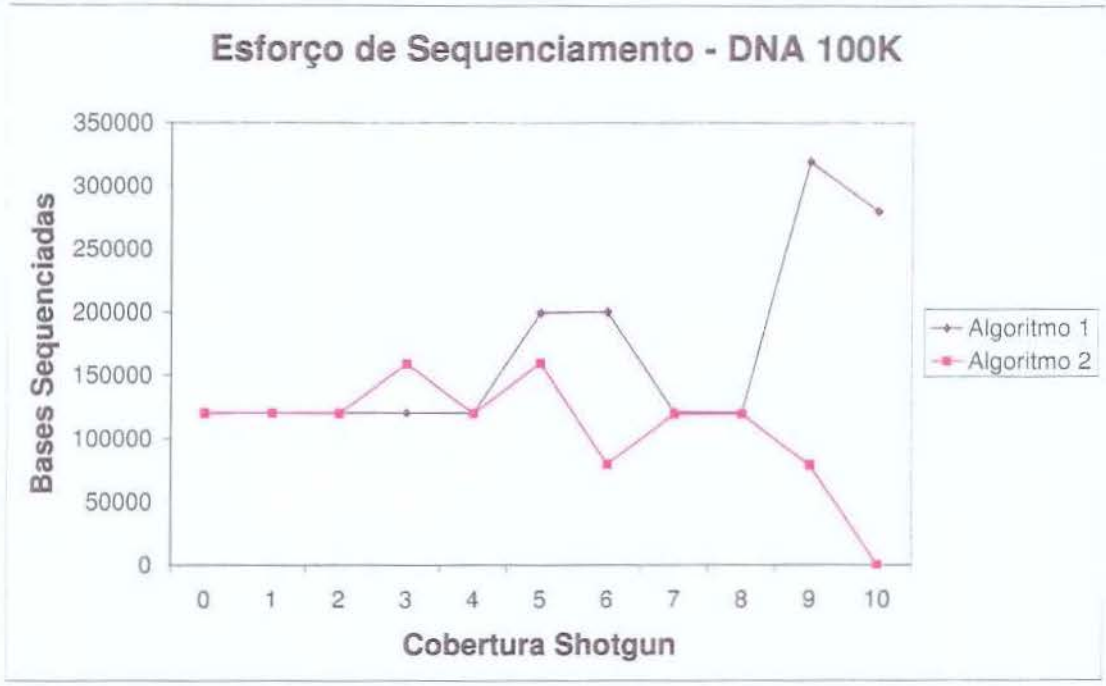

Figura B.25: Esforço de seqüenciamento para DNA100K_13c40K_p500.

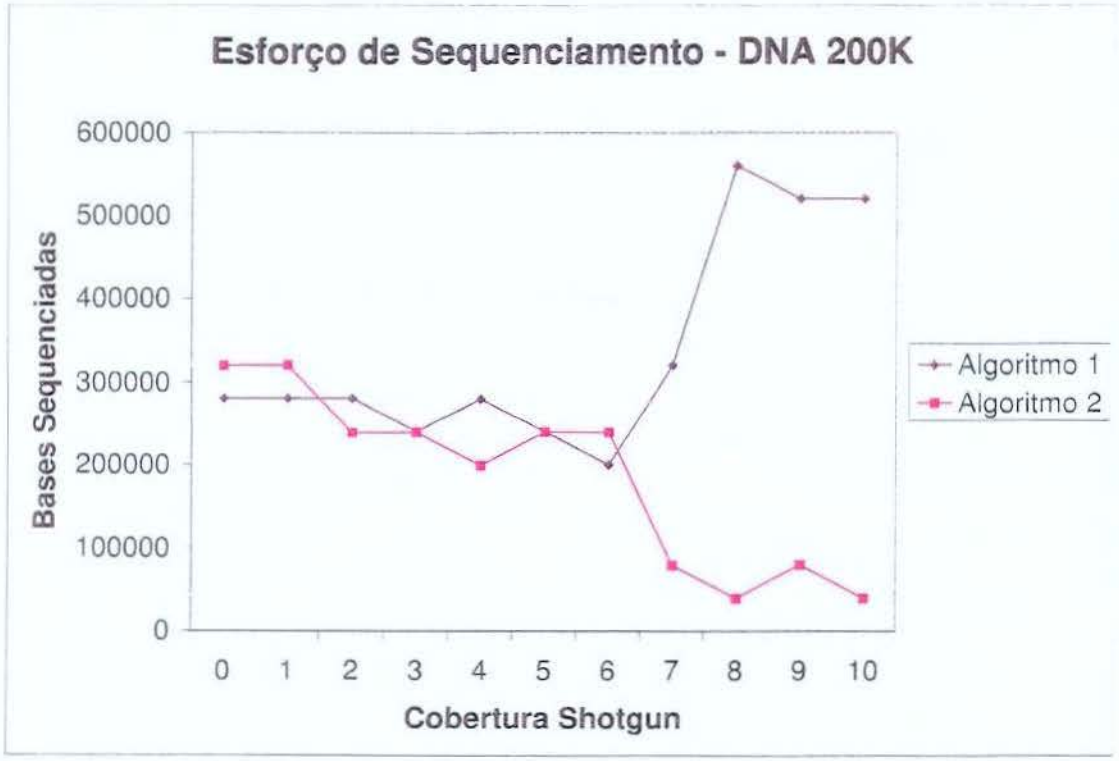

Figura B.26: Esforço de seqüenciamento para DNA200K_25c40K_p500.

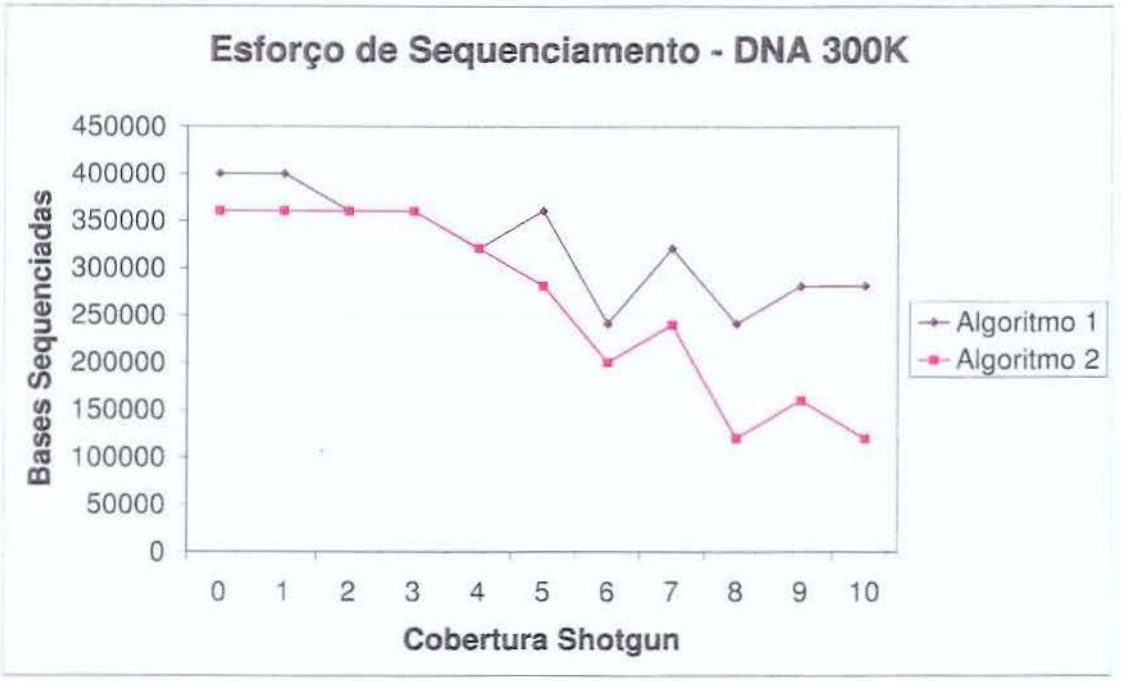

Figura B.27: Esforço de seqüenciamento para DNA300K_38c40K_p500.

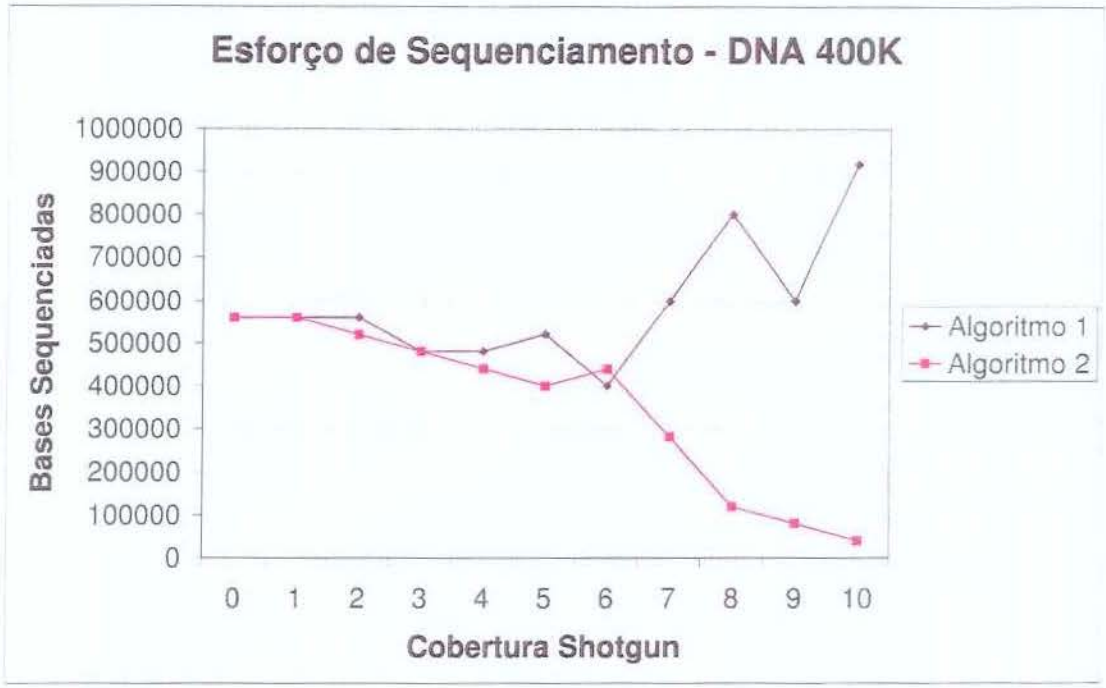

Figura B.28: Esforço de seqüenciamento para DNA400K_50c40K_p500.

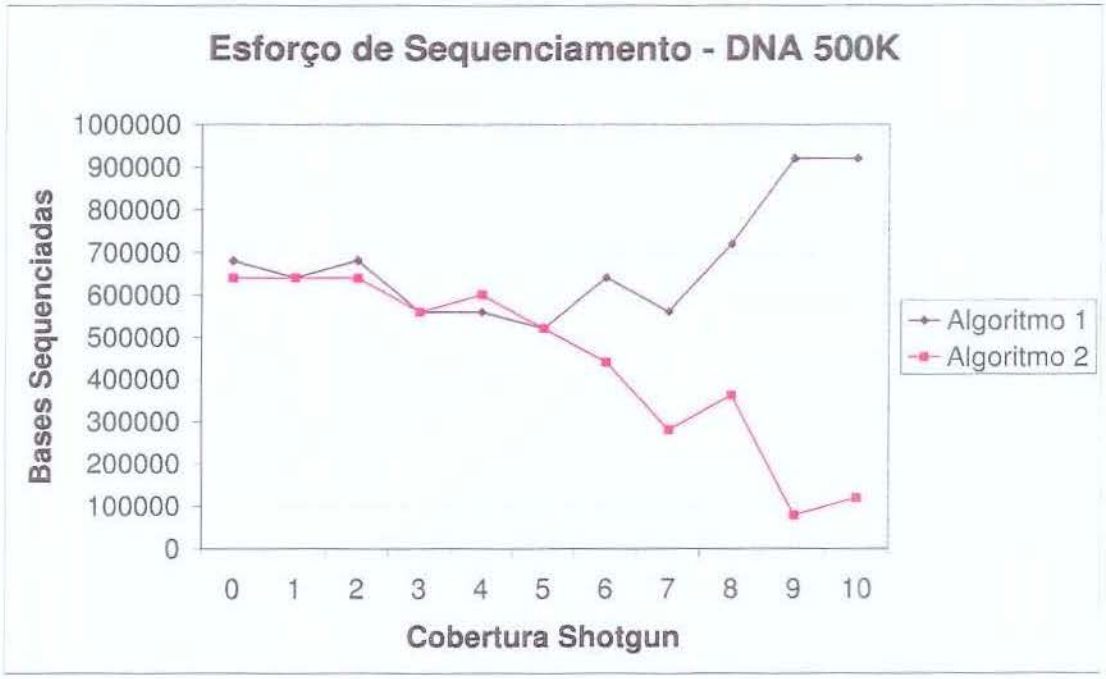

Figura B.29: Esforço de seqüenciamento para DNA500K_63c40K_p500.

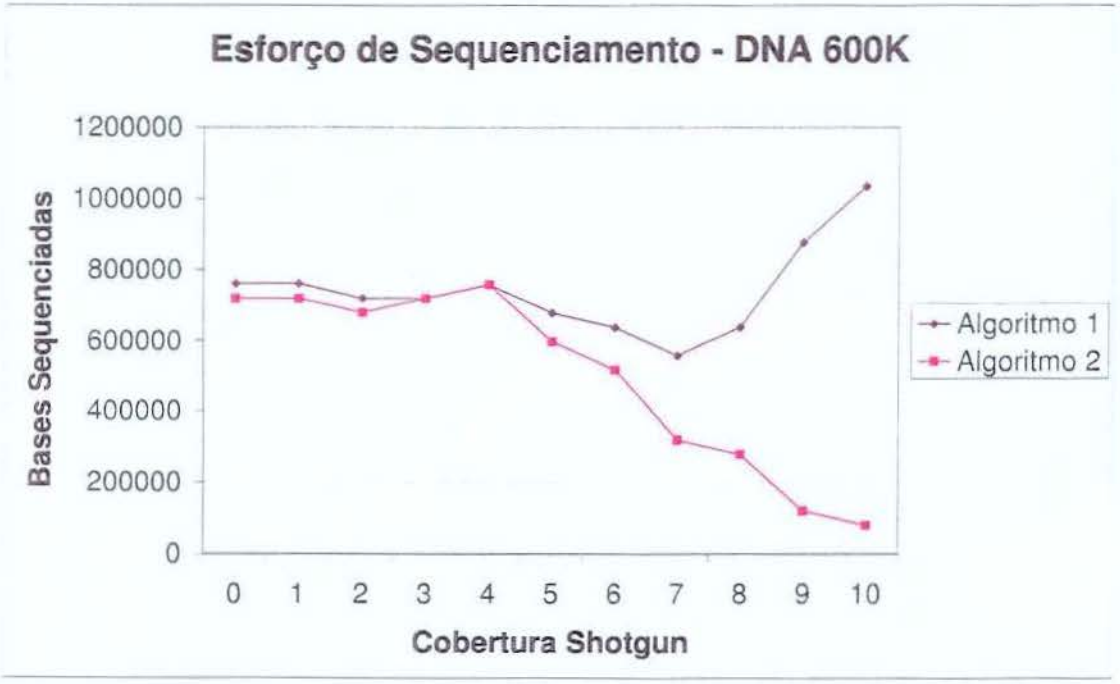

Figura B.30: Esforço de seqüenciamento para `DNA600K_75c40K_p500`.

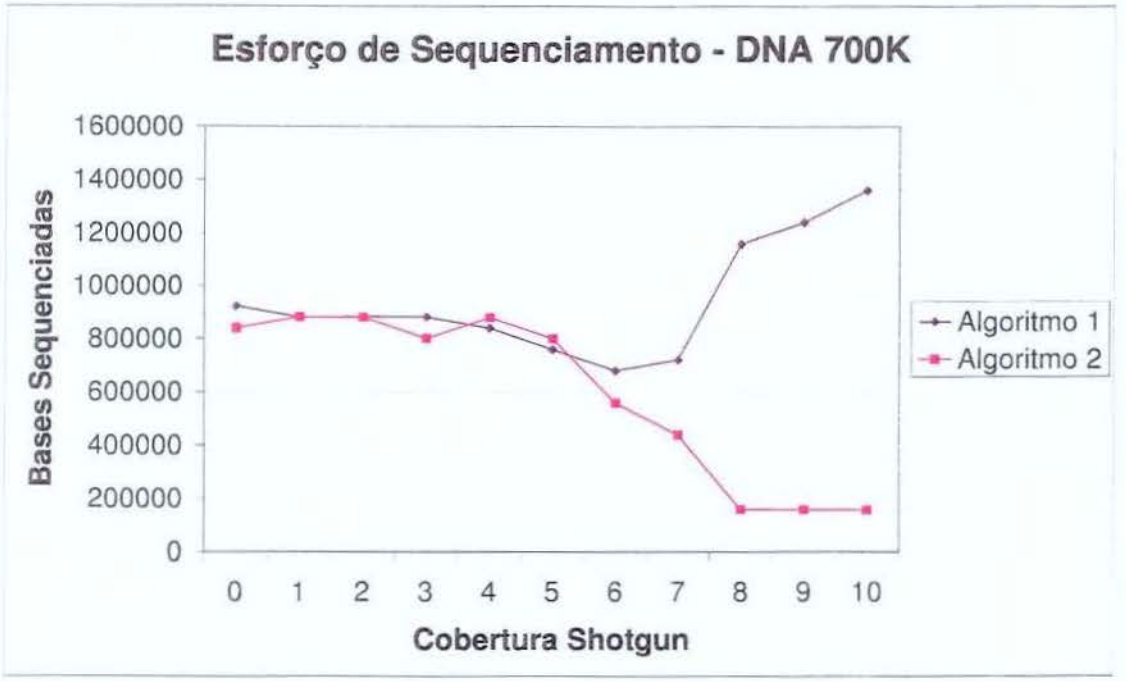

Figura B.31: Esforço de seqüenciamento para `DNA700K_88c40K_p500`.

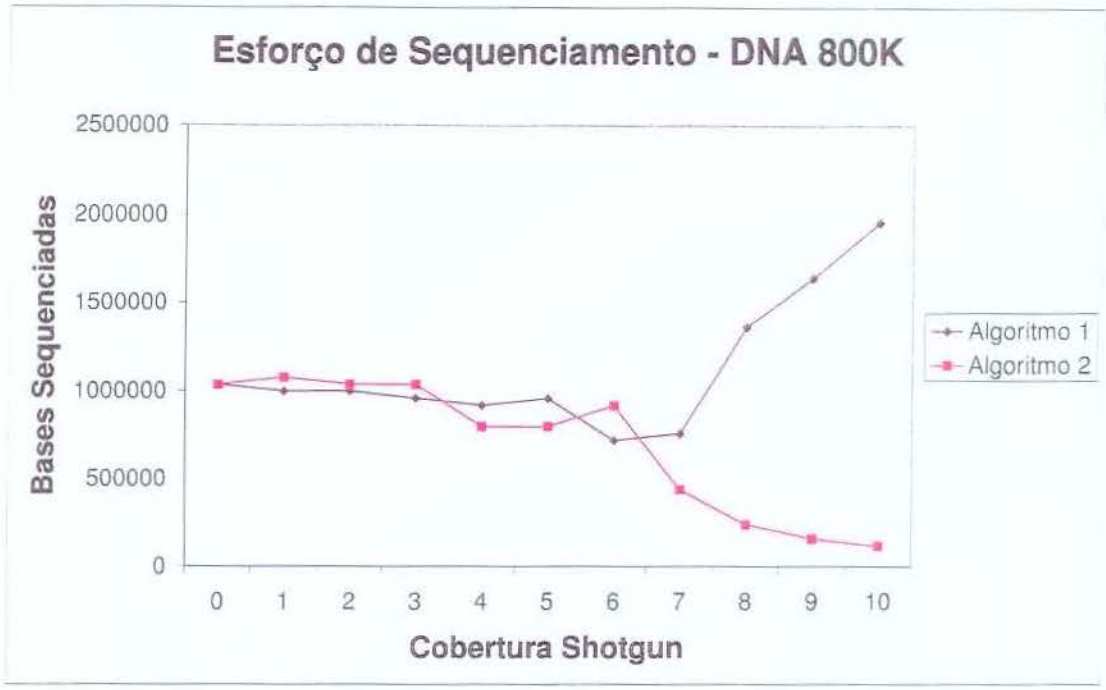

Figura B.32: Esforço de seqüenciamento para DNA800K_100c40K_p500.

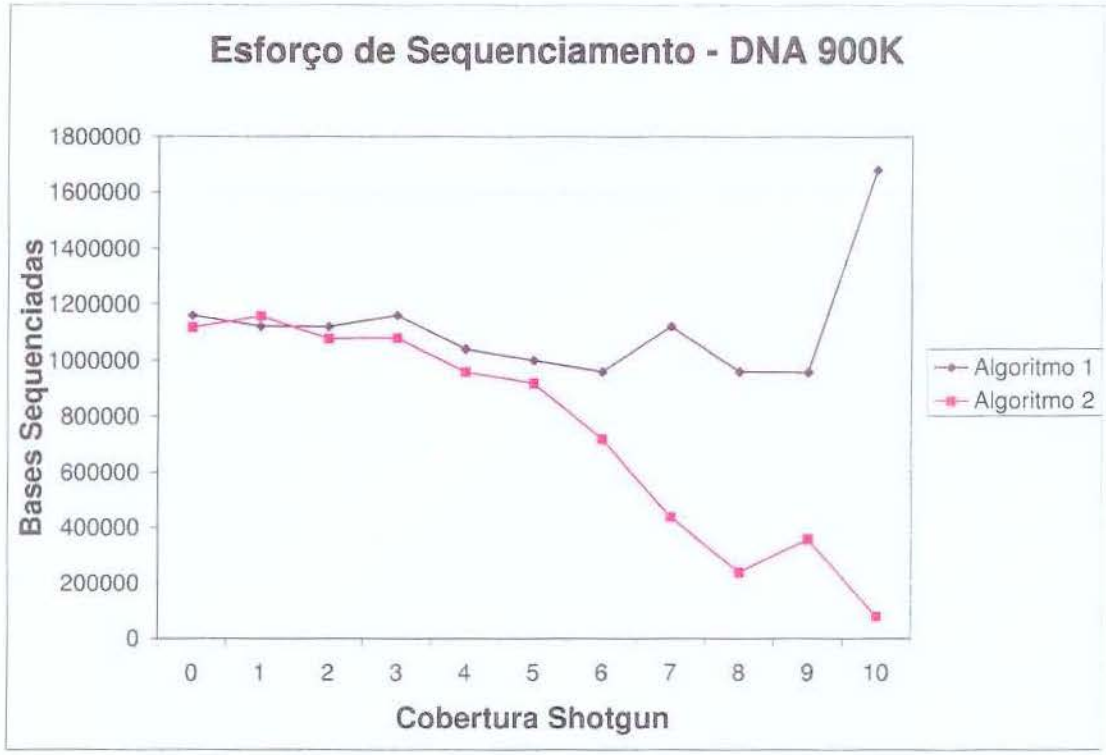

Figura B.33: Esforço de seqüenciamento para DNA900K_113c40K_p500.

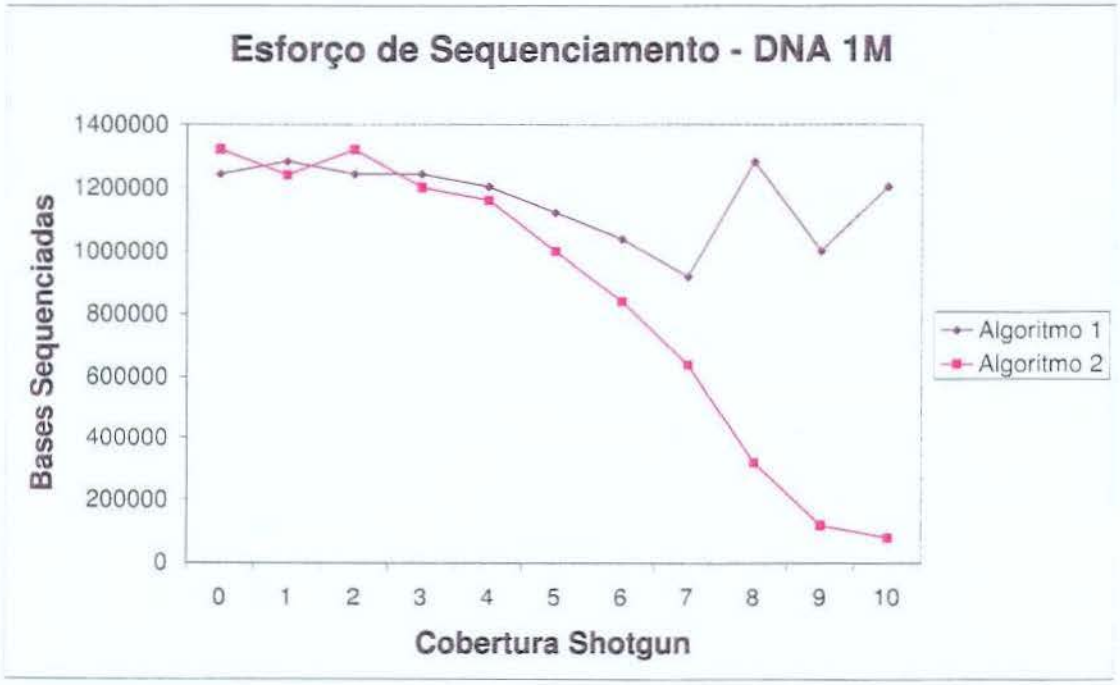

Figura B.34: Esforço de seqüenciamento para DNA1000K_125c40K_p500.

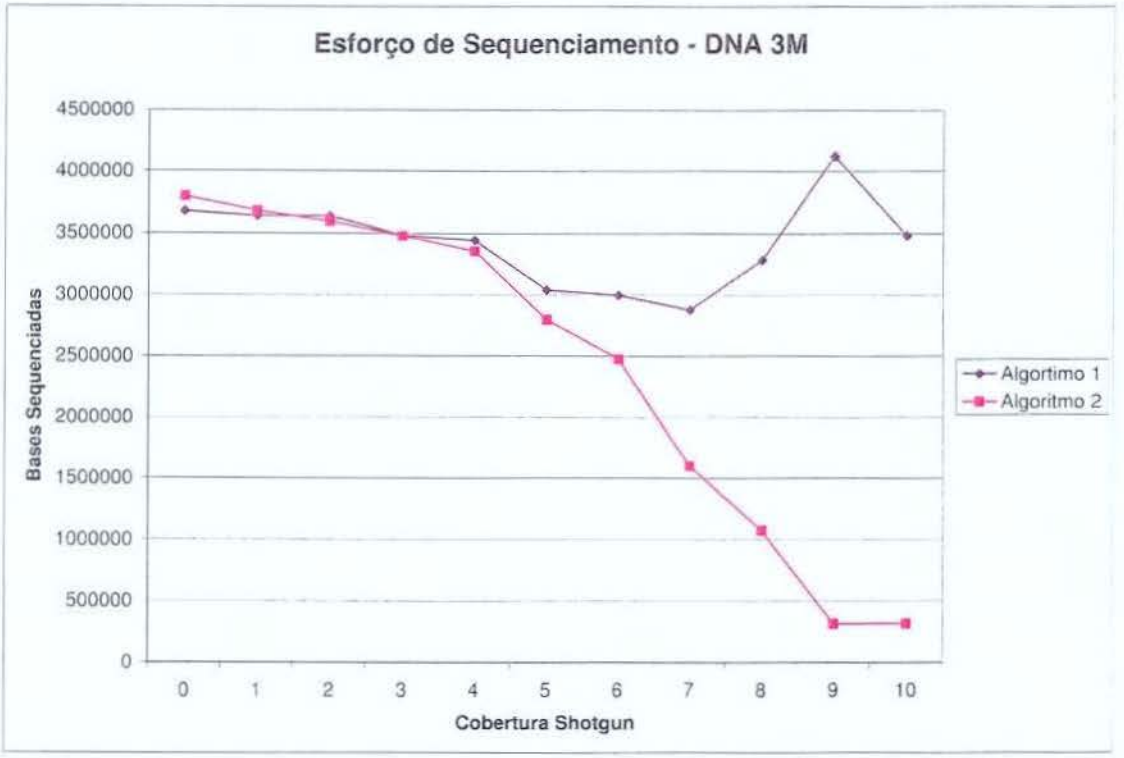

Figura B.35: Esforço de seqüenciamento para DNA3M_375c40K_p450.

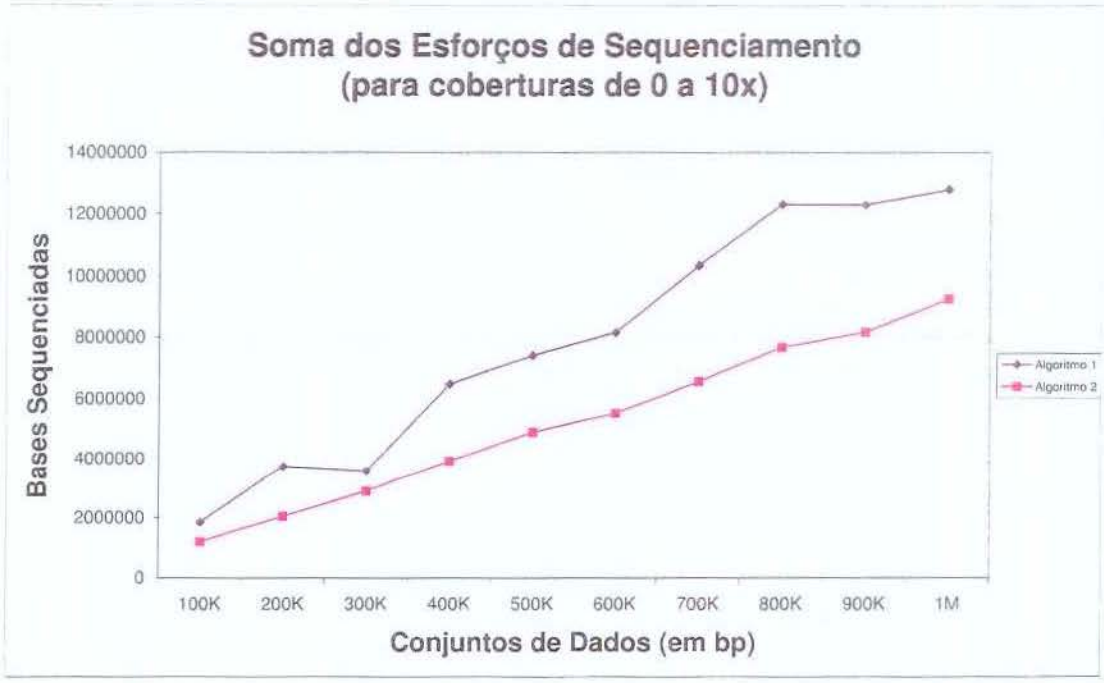

Figura B.36: Esforço de seqüenciamento total por conjunto.

B.4 Gráficos: tempo total da montagem OSS

Os gráficos dessa seção são subsídios para análises feitas na seção 4.4.1.

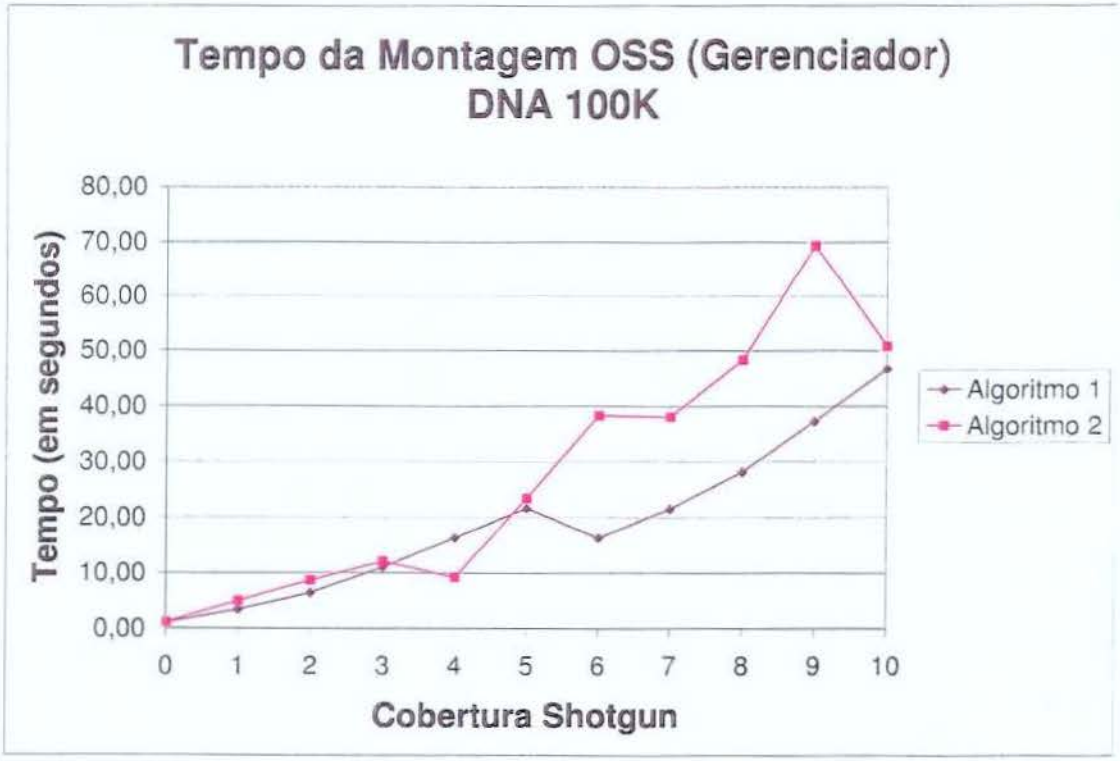

Figura B.37: Tempo da montagem OSS para `DNA100K_13c40K_p500`.

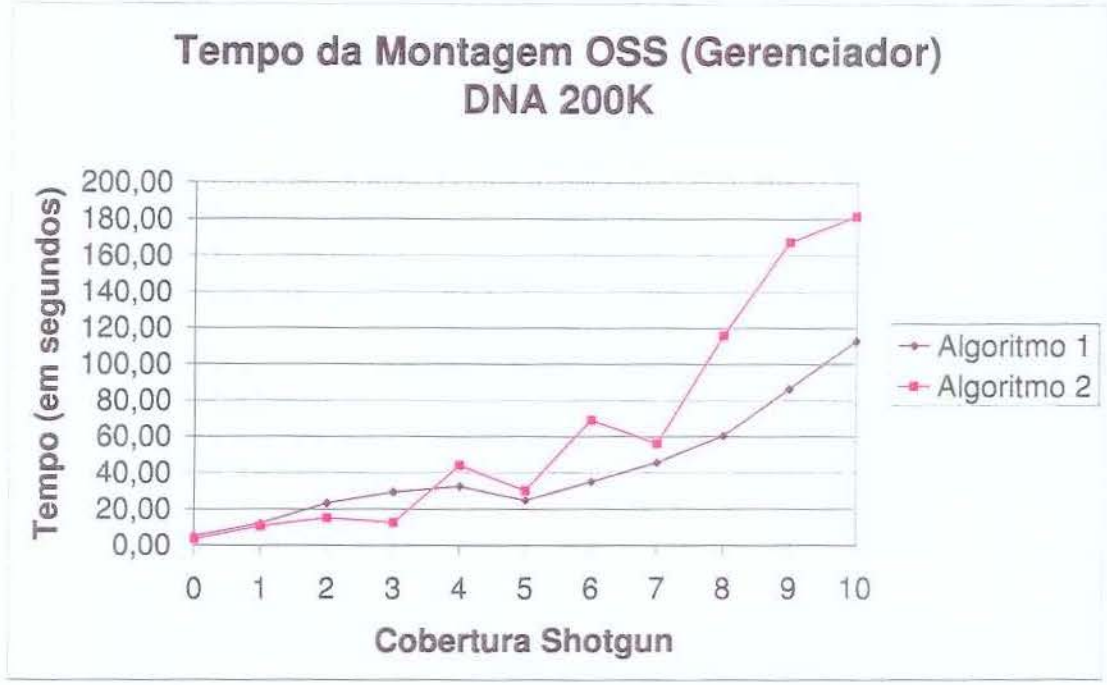

Figura B.38: Tempo da montagem OSS para DNA200K_25c40K_p500.

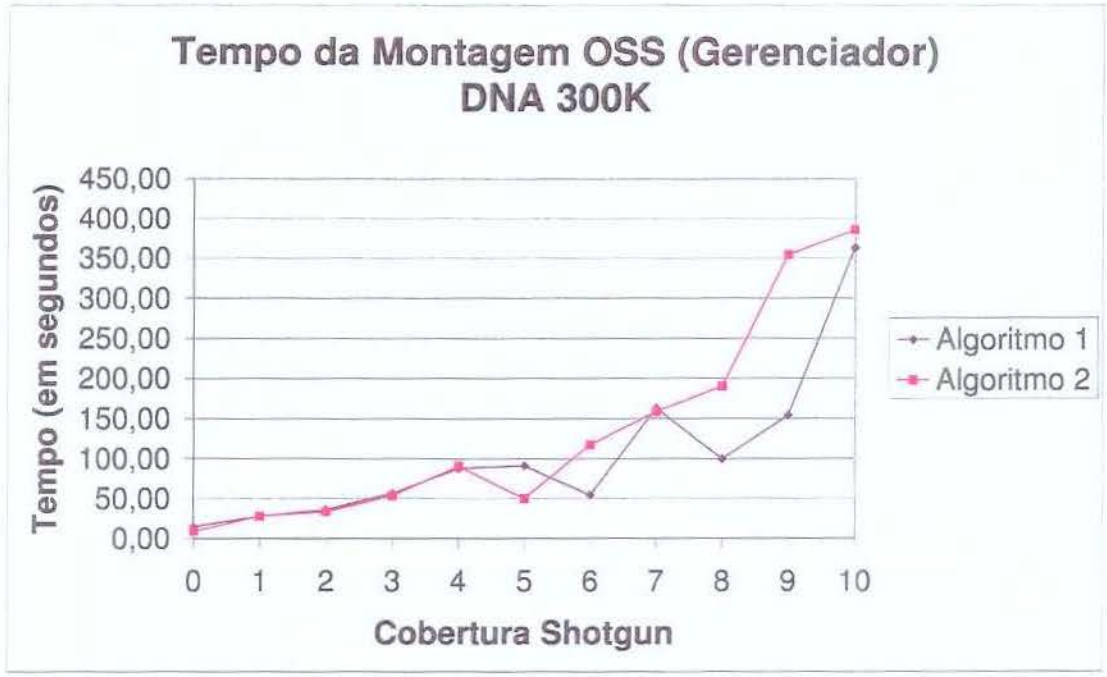

Figura B.39: Tempo da montagem OSS para DNA300K_38c40K_p500.

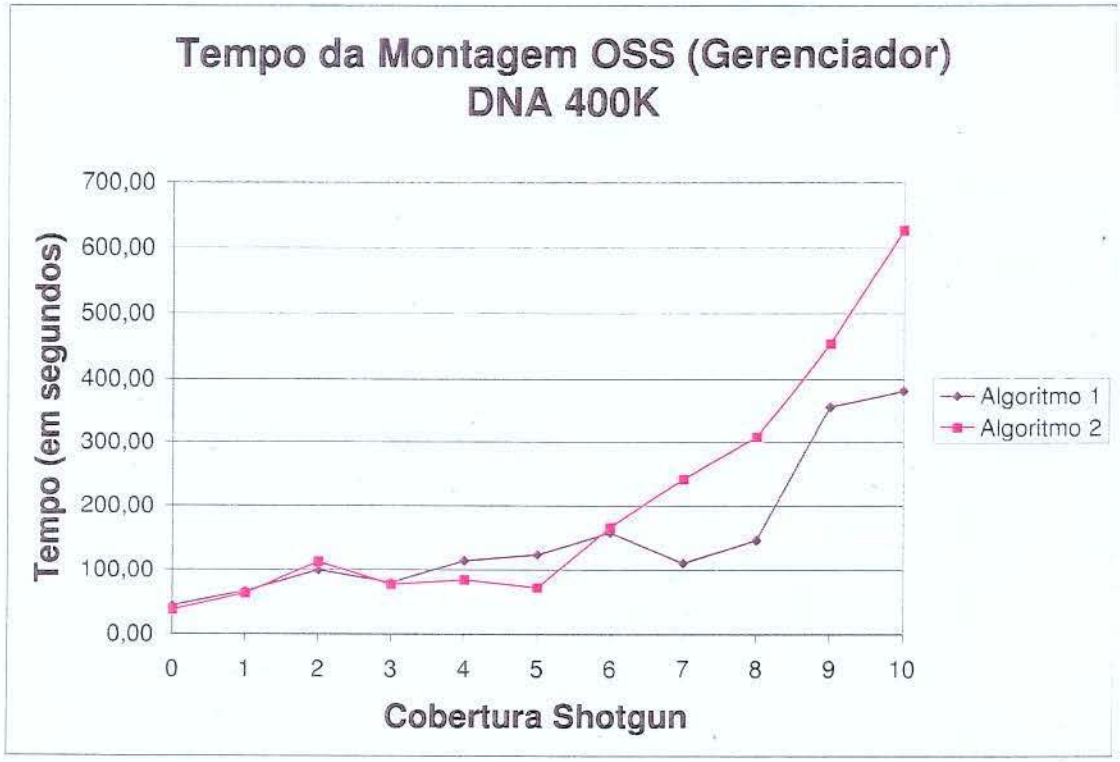

Figura B.40: Tempo da montagem OSS para `DNA400K_50c40K_p500`.

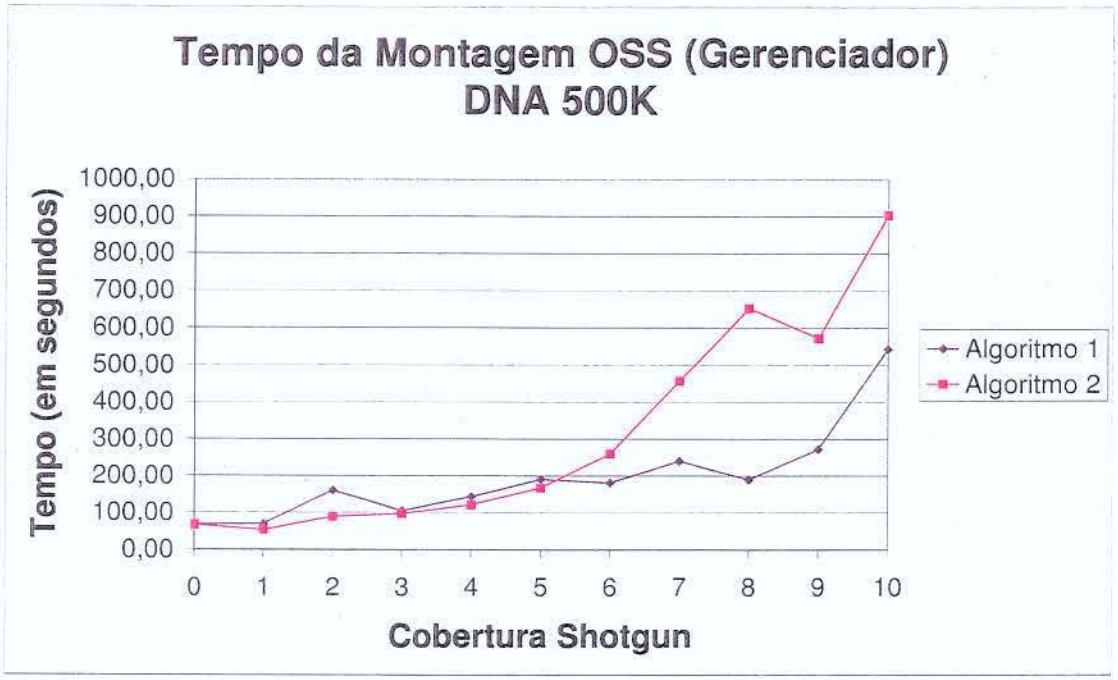

Figura B.41: Tempo da montagem OSS para `DNA500K_63c40K_p500`.

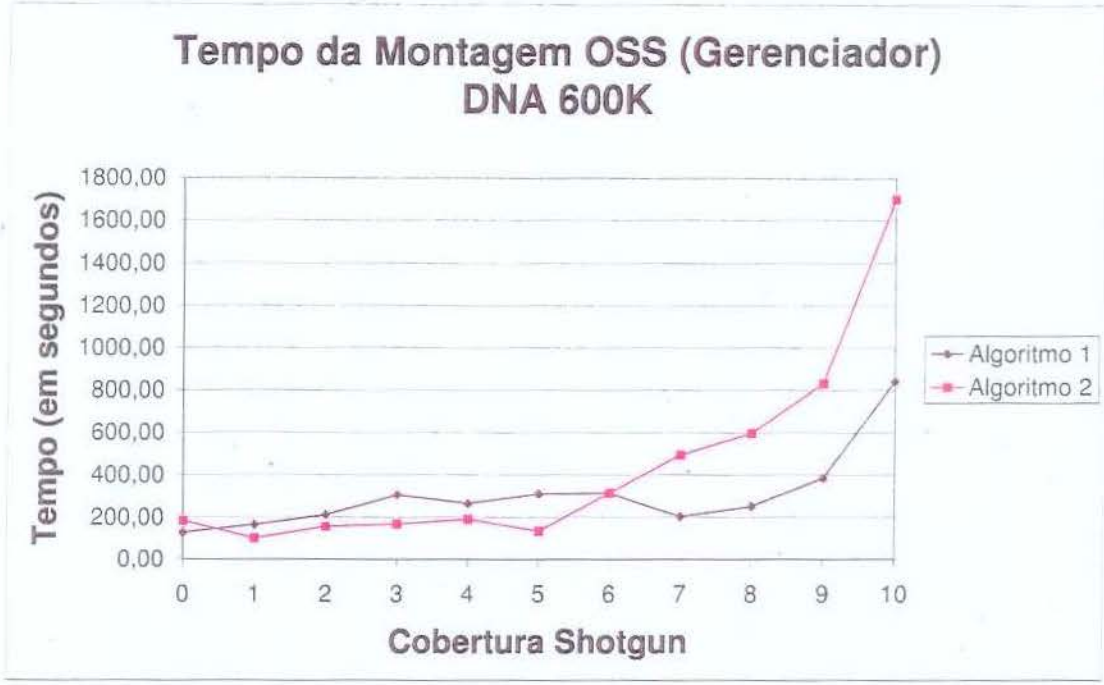

Figura B.42: Tempo da montagem OSS para DNA600K_75c40K_p500.

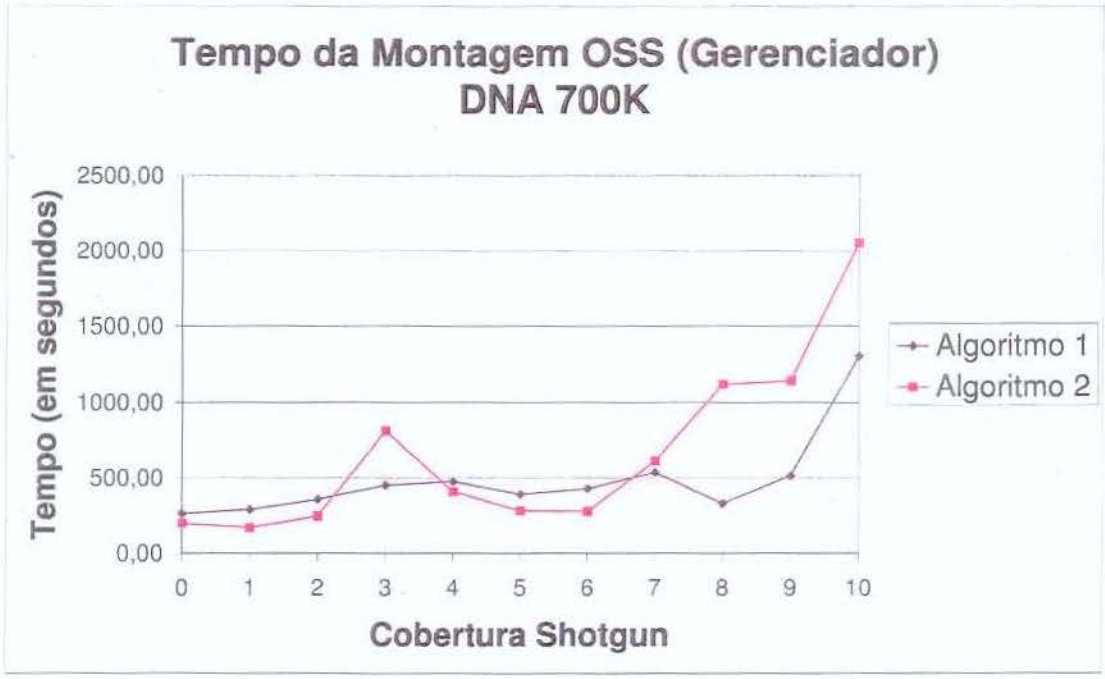

Figura B.43: Tempo da montagem OSS para DNA700K_88c40K_p500.

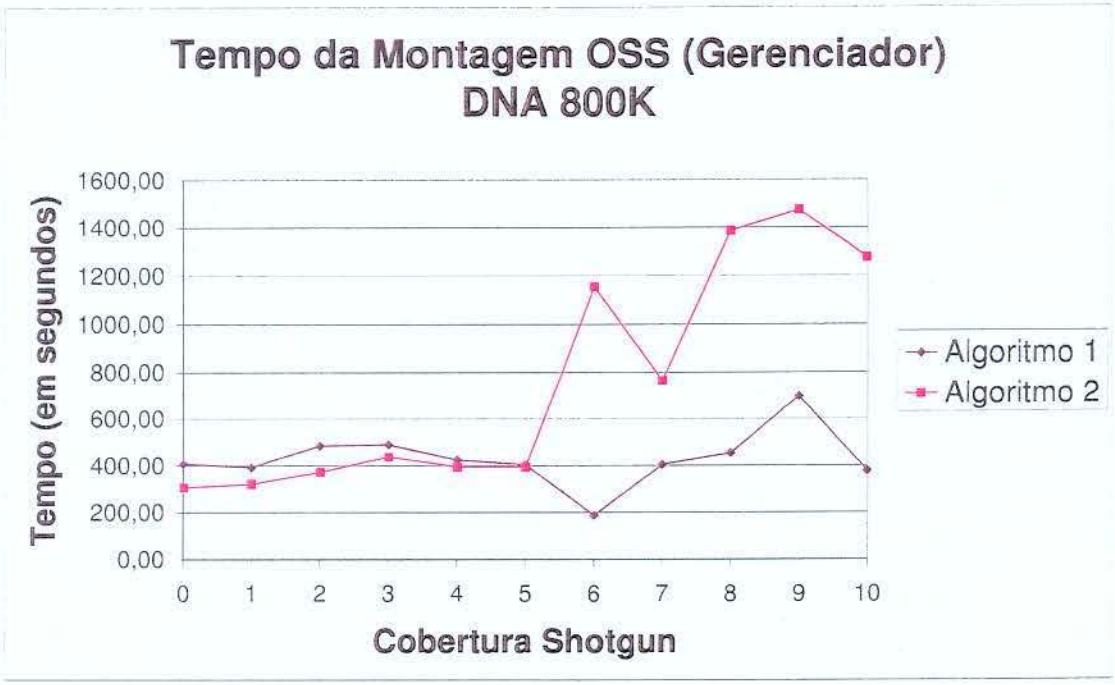

Figura B.44: Tempo da montagem OSS para `DNA800K_100c40K_p500`.

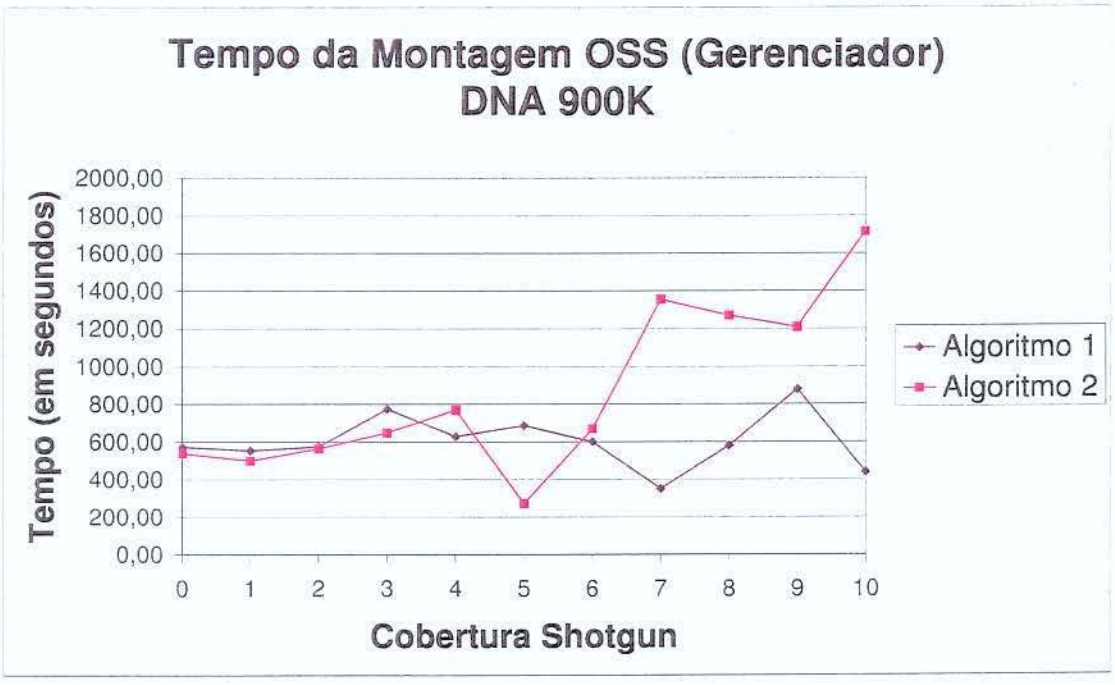

Figura B.45: Tempo da montagem OSS para `DNA900K_113c40K_p500`.

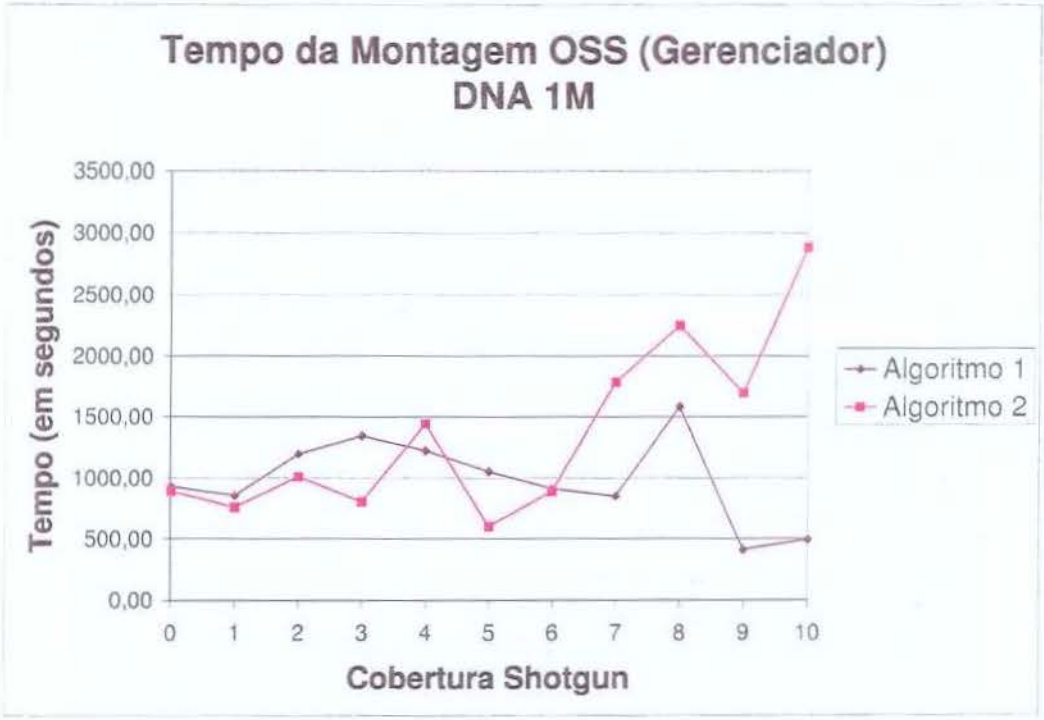

Figura B.46: Tempo da montagem OSS para DNA1000K_125c40K_p500.

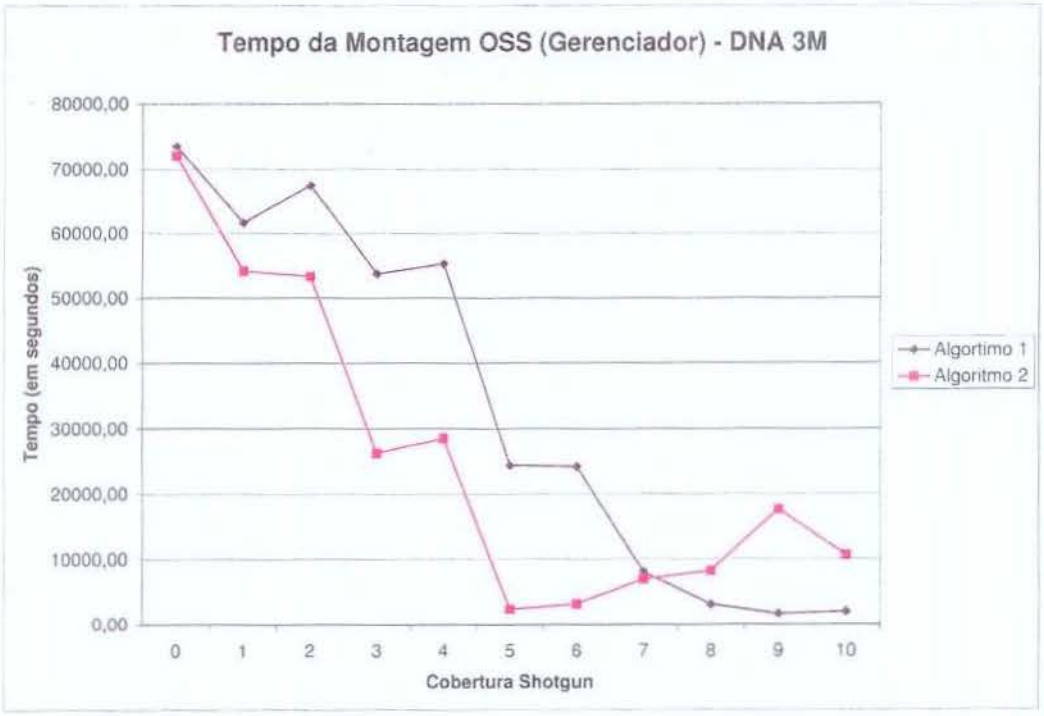

Figura B.47: Tempo da montagem OSS para DNA3M_375c40K_p450.

B.5 Gráficos: percentual de tempo dos módulos do gerenciador com algoritmo 1

Esta série de gráficos faz parte da análise feita na seção 4.4.2.

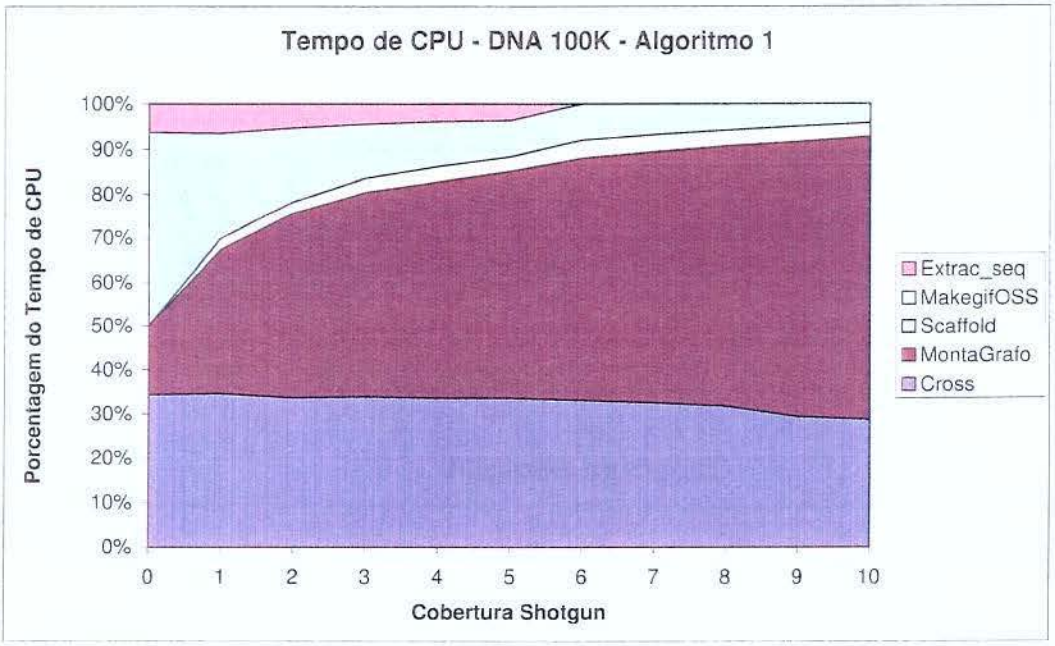

Figura B.48: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA100K_13c40K_p500`.

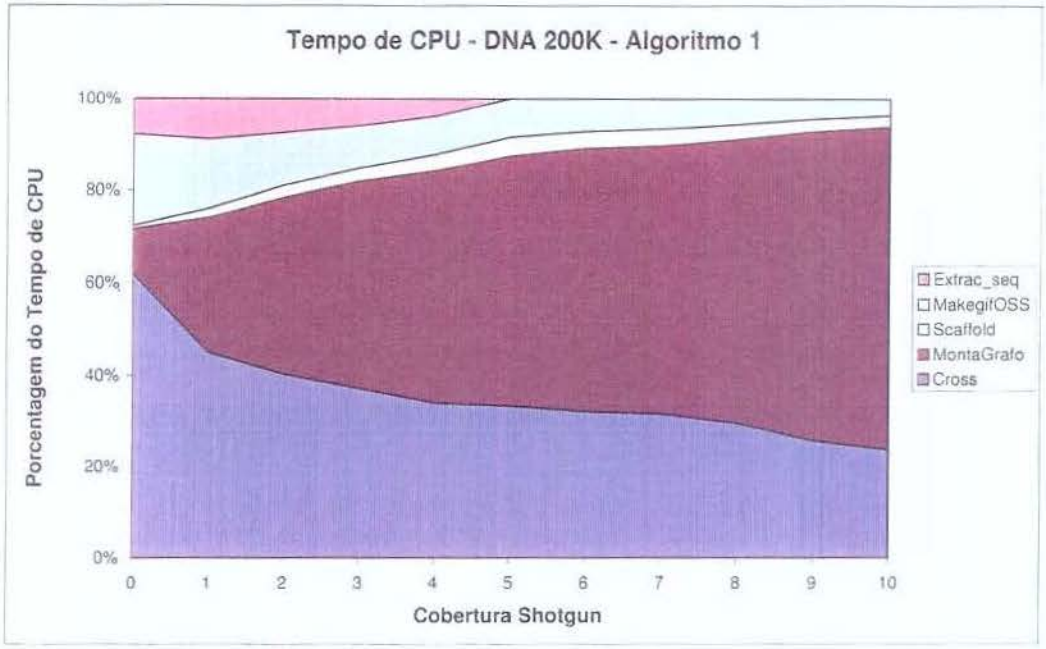

Figura B.49: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA200K_25c40K_p500`.

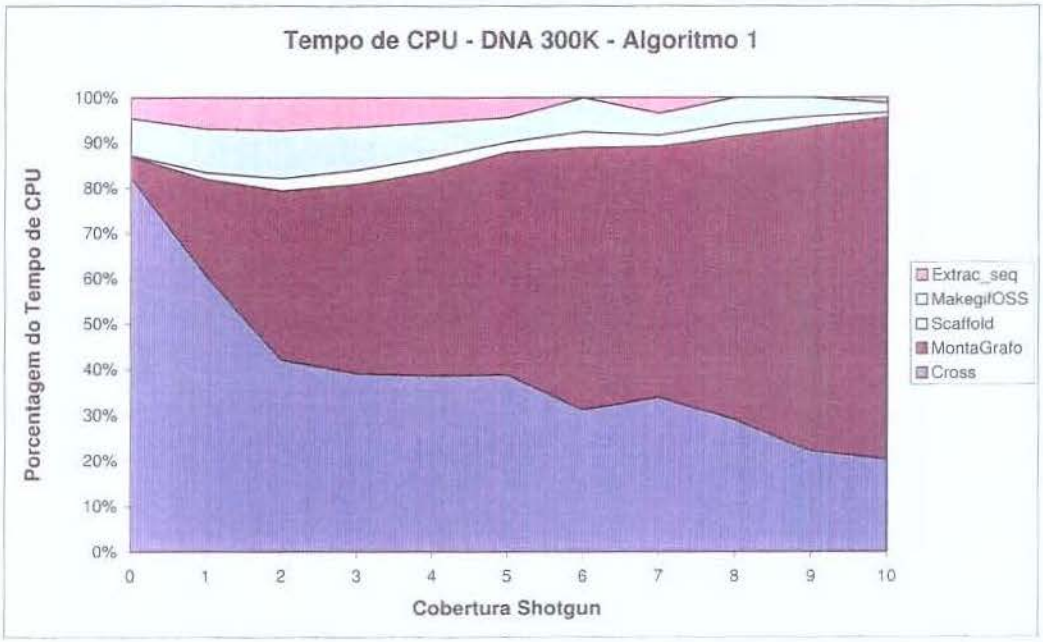

Figura B.50: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA300K_38c40K_p500`.

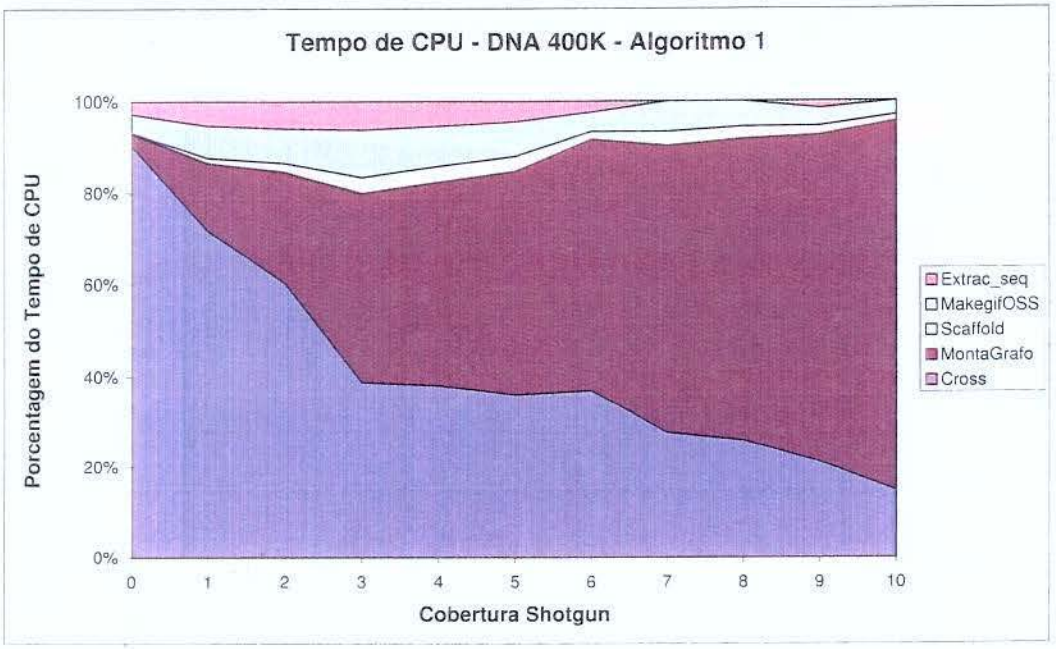

Figura B.51: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA400K_50c40K_p500`.

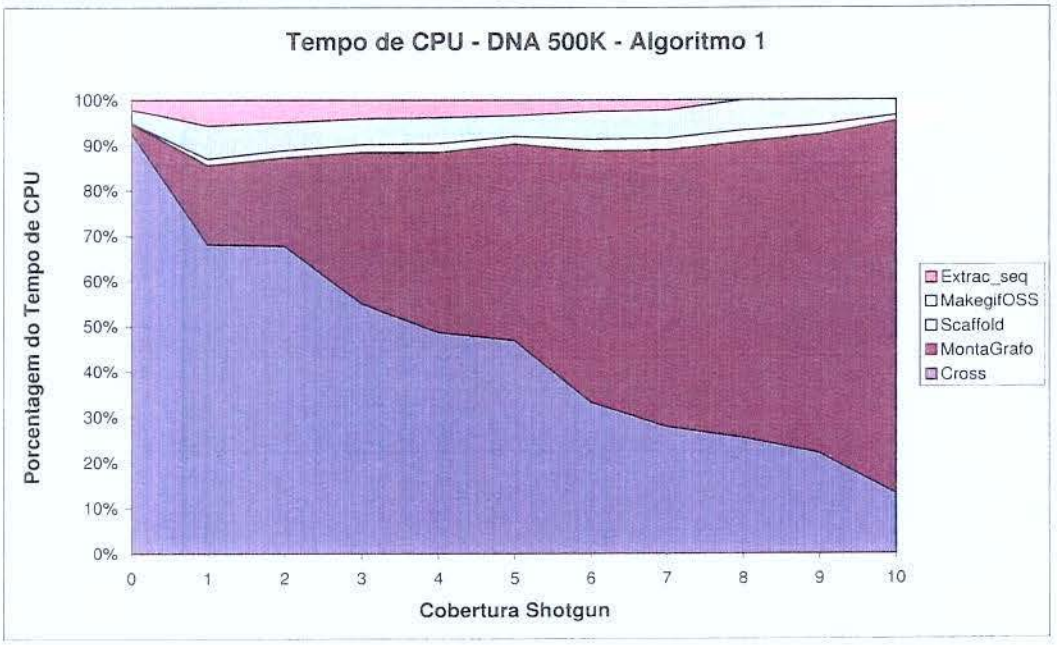

Figura B.52: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA500K_63c40K_p500`.

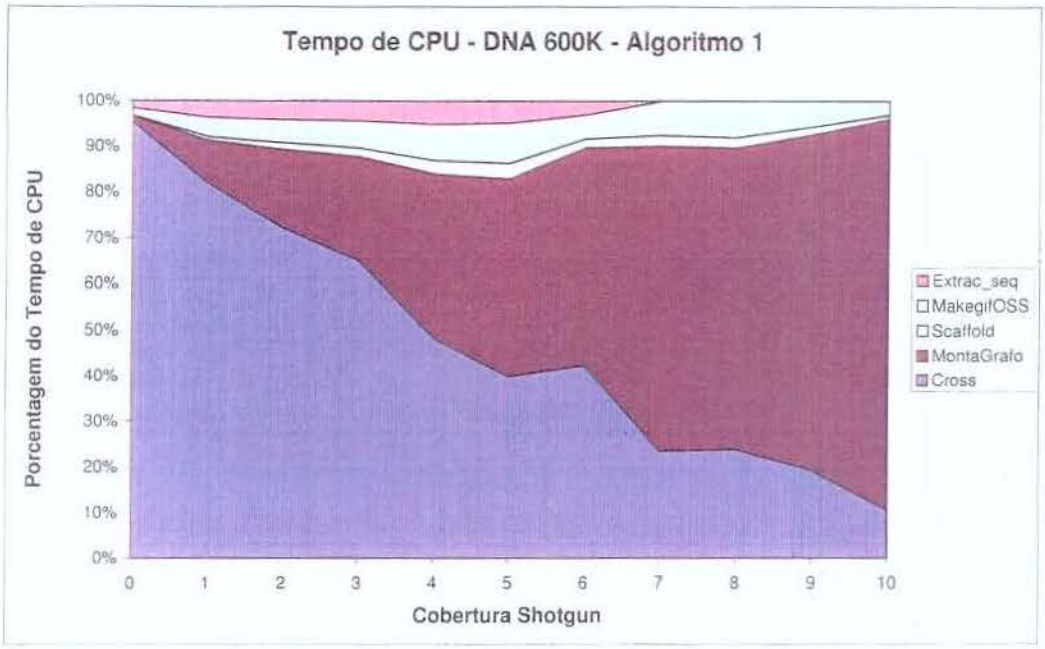

Figura B.53: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA600K_75c40K_p500`.

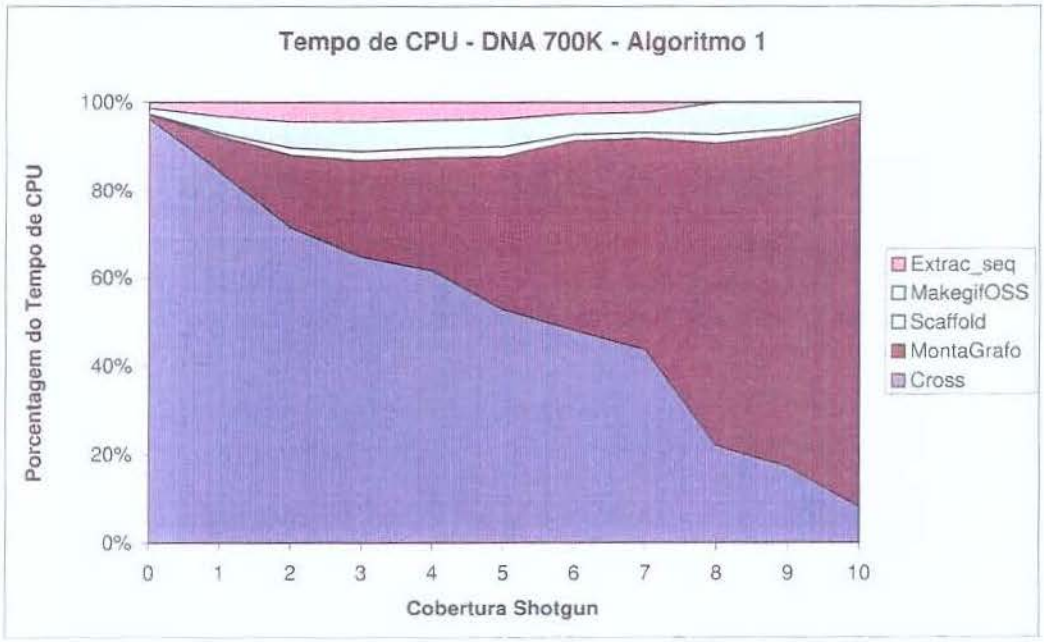

Figura B.54: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA700K_88c40K_p500`.

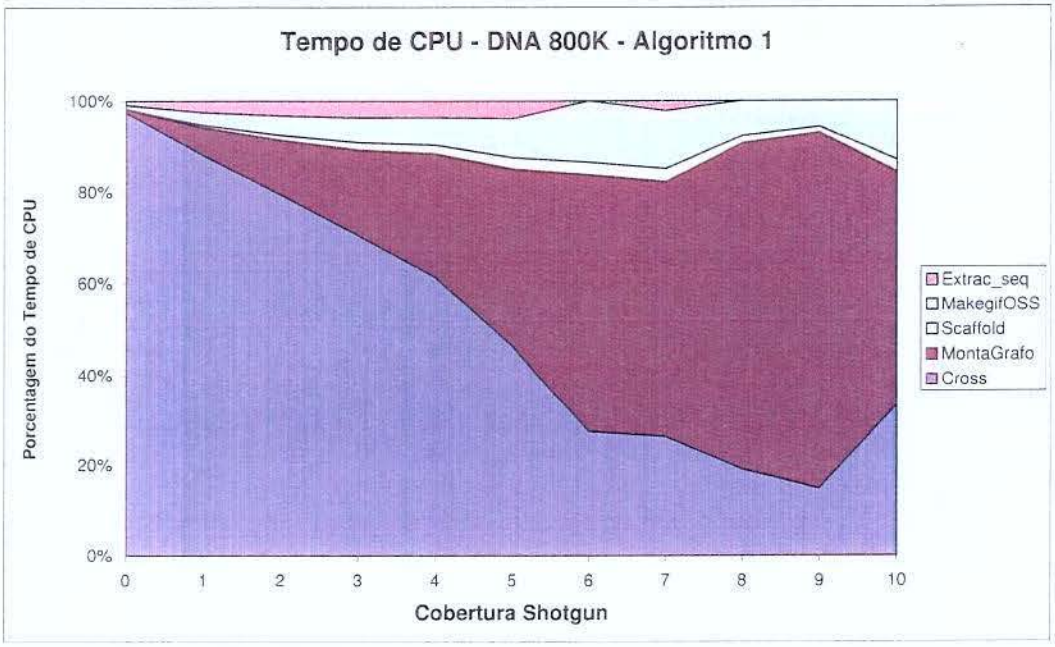

Figura B.55: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA800K_100c40K_p500`.

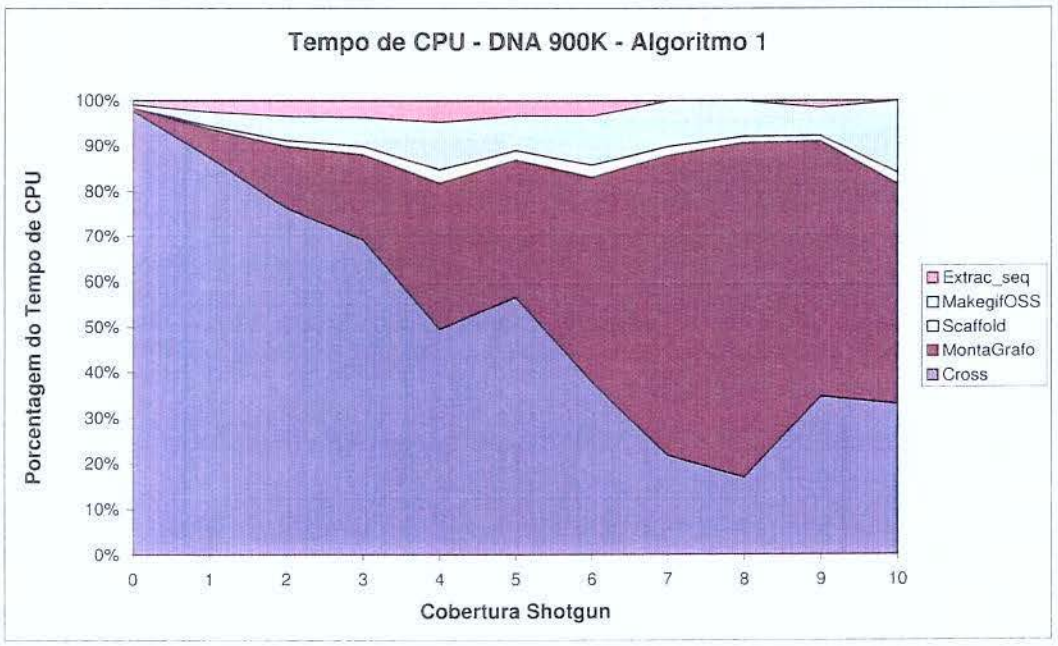

Figura B.56: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA900K_113c40K_p500`.

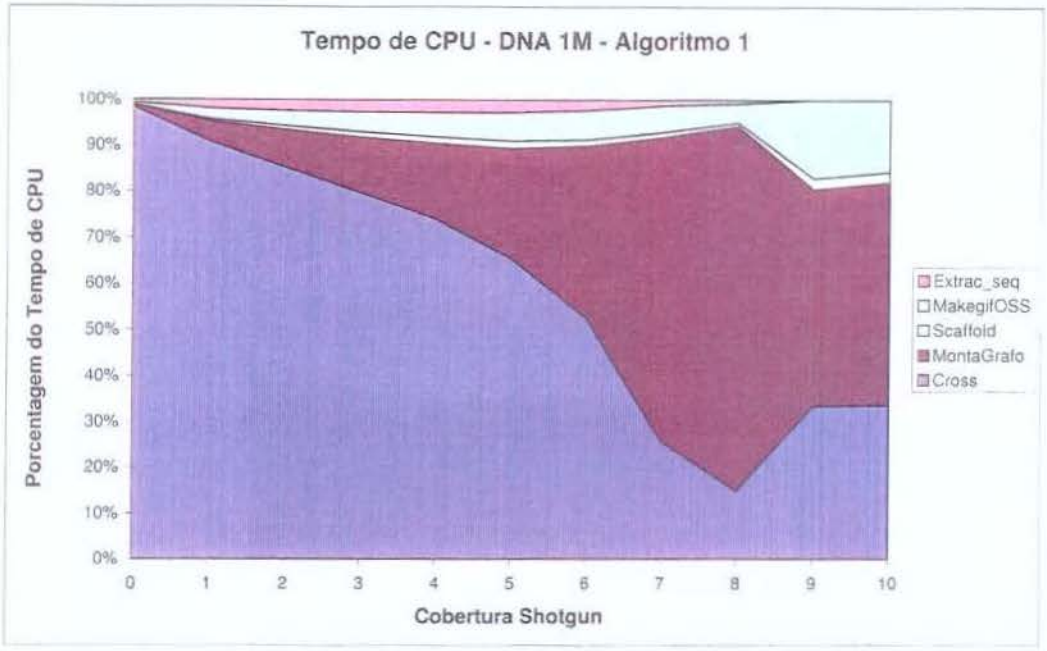

Figura B.57: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA1000K_125c40K_p500`.

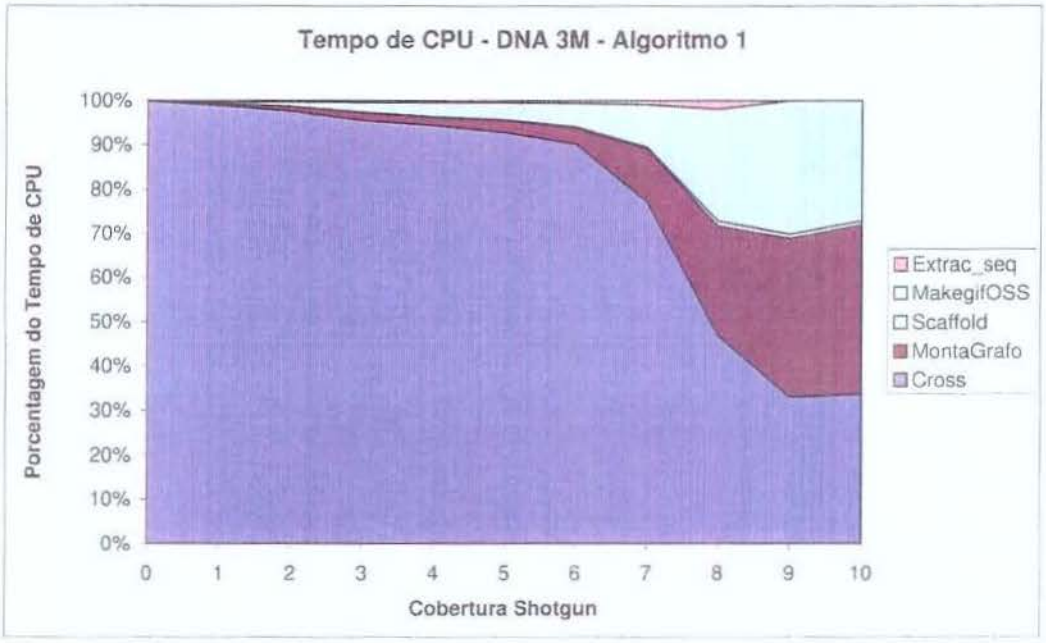

Figura B.58: Tempo percentual dos vários módulos do gerenciador com algoritmo 1 para `DNA3M_375c40K_p450`.

B.6 Gráficos: percentual de tempo dos módulos do gerenciador com algoritmo 2

Esta série de gráficos faz parte da análise feita na seção 4.4.2.

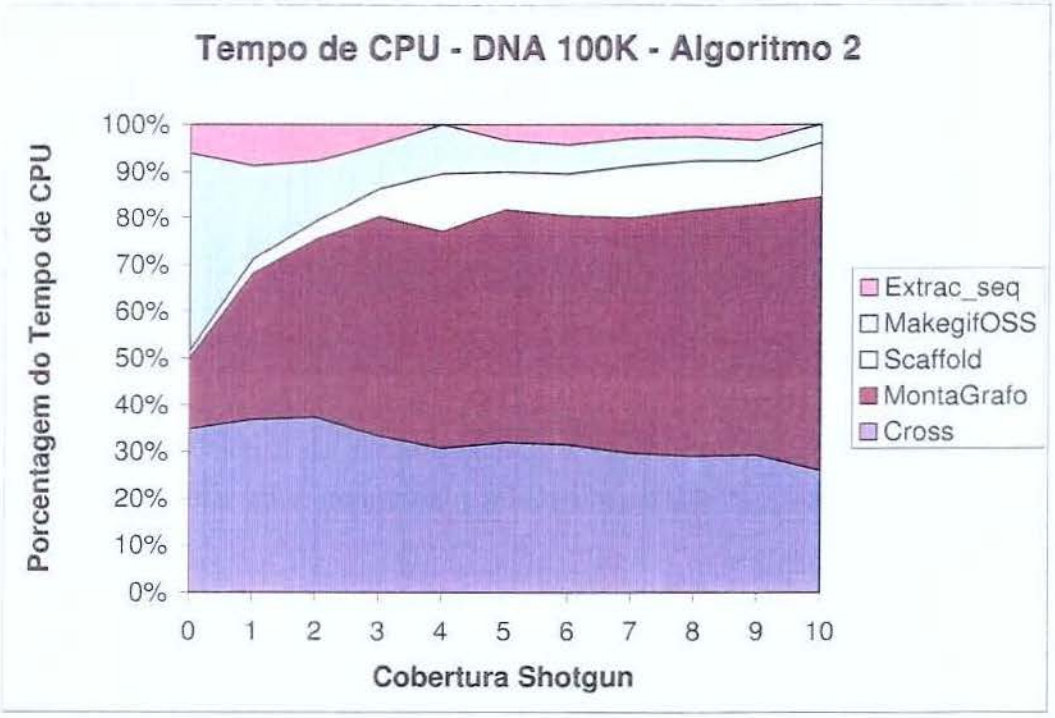

Figura B.59: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA100K_13c40K_p500`.

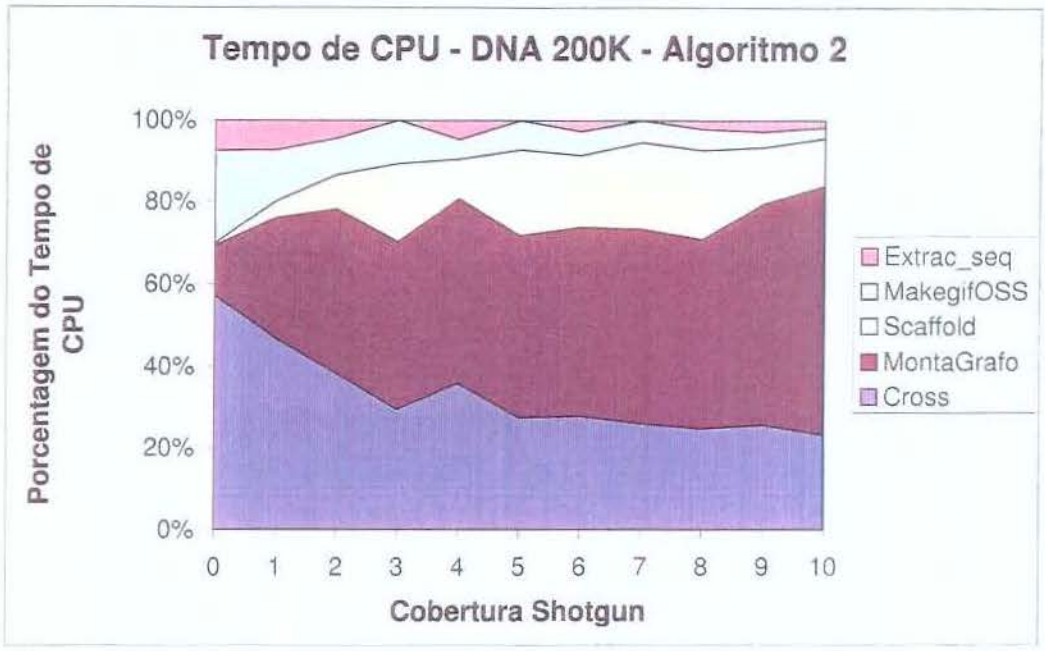

Figura B.60: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA200K_25c40K_p500`.

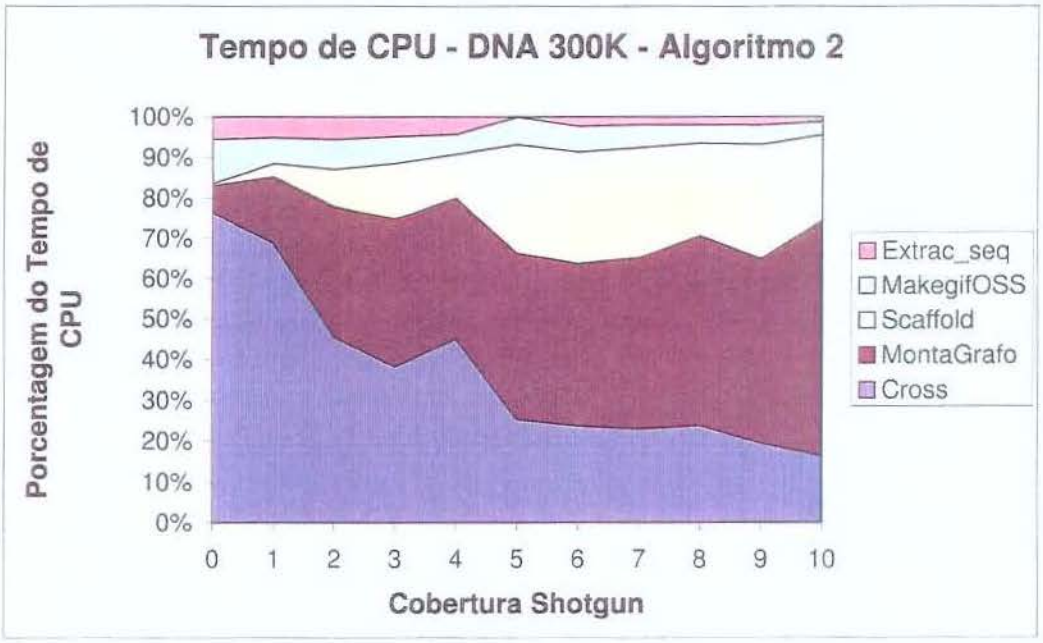

Figura B.61: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA300K_38c40K_p500`.

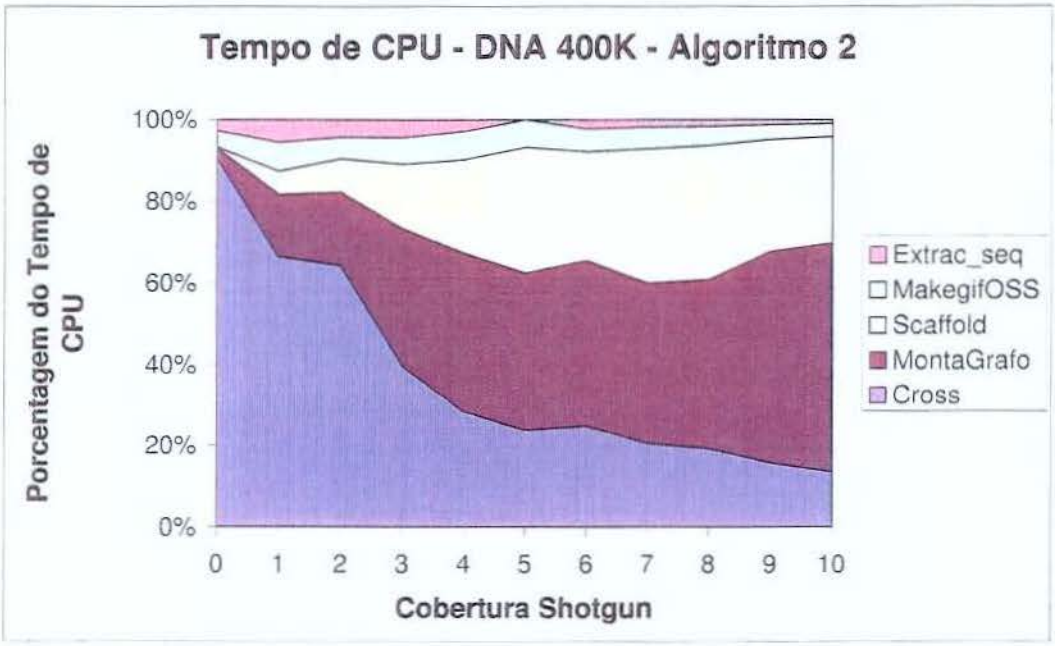

Figura B.62: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA400K_50c40K_p500`.

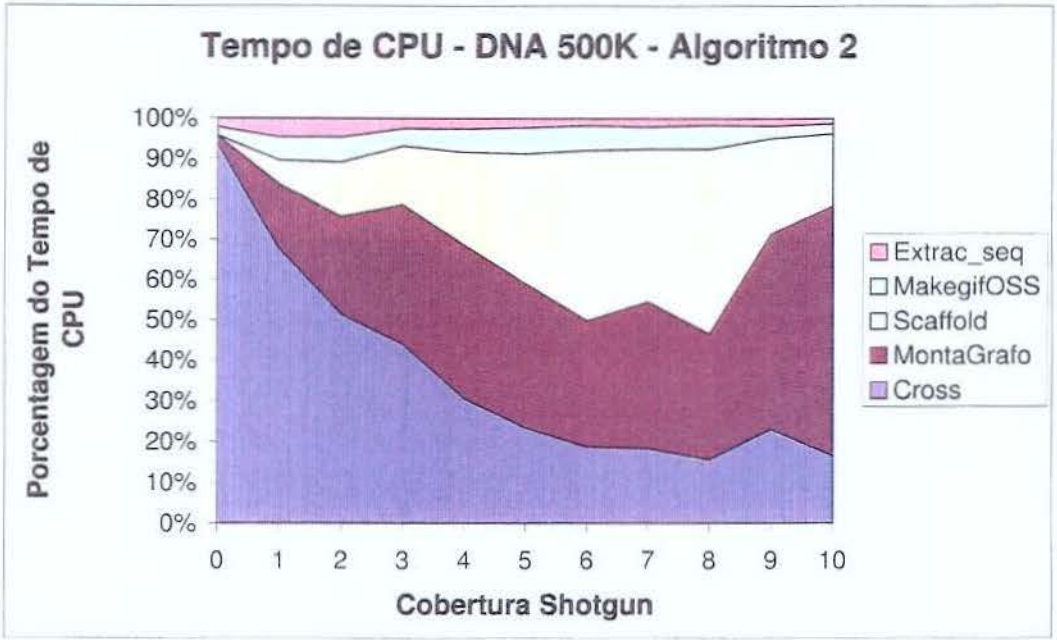

Figura B.63: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA500K_63c40K_p500`.

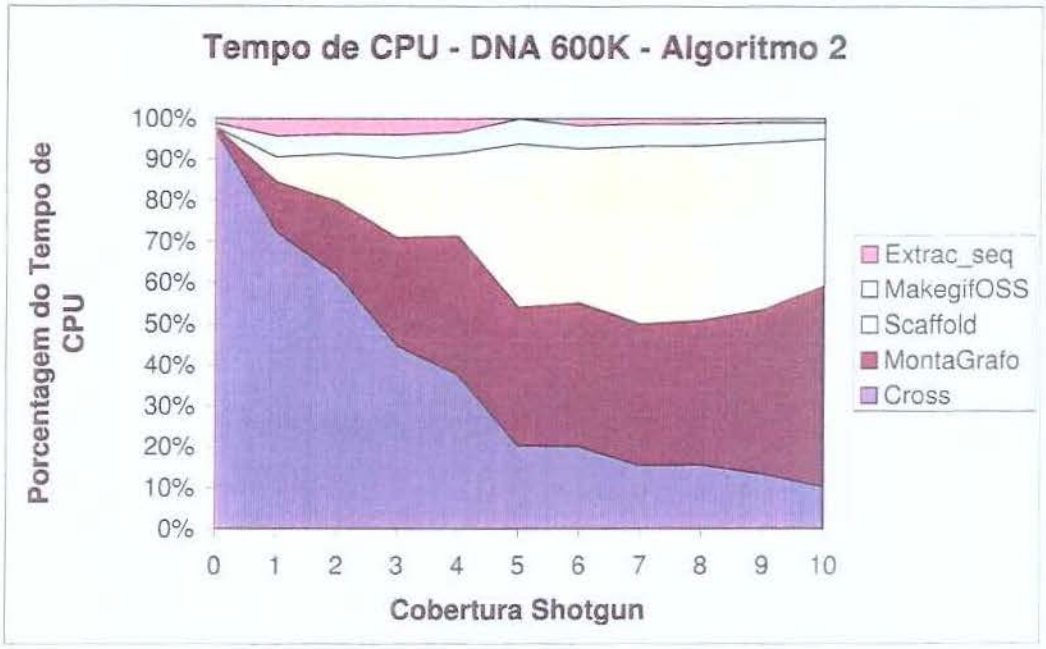

Figura B.64: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA600K_75c40K_p500`.

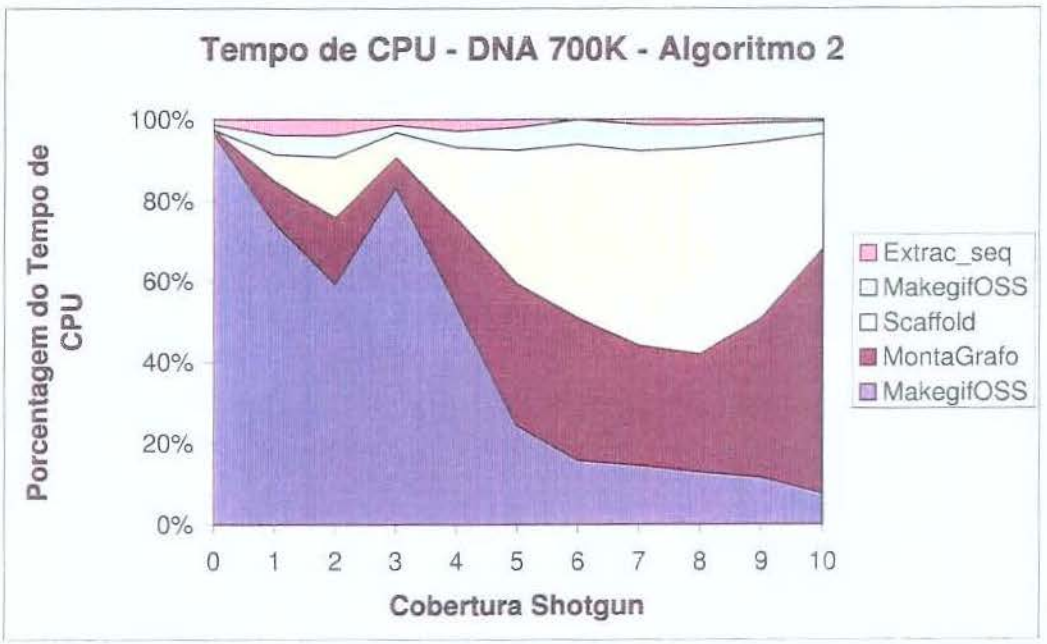

Figura B.65: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA700K_88c40K_p500`.

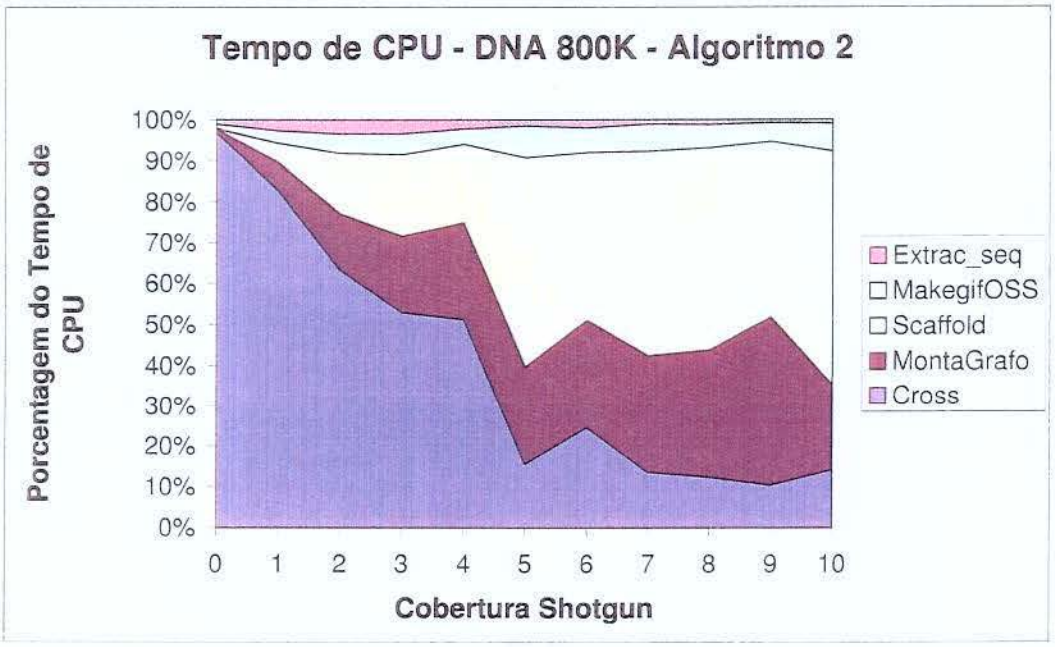

Figura B.66: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA800K_100c40K_p500`.

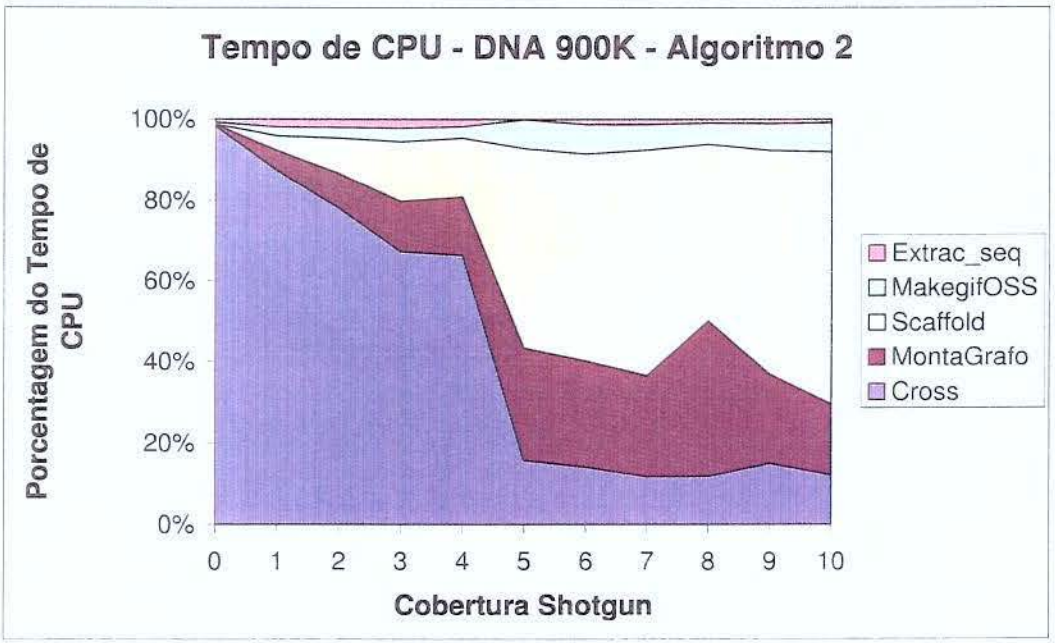

Figura B.67: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA900K_113c40K_p500`.

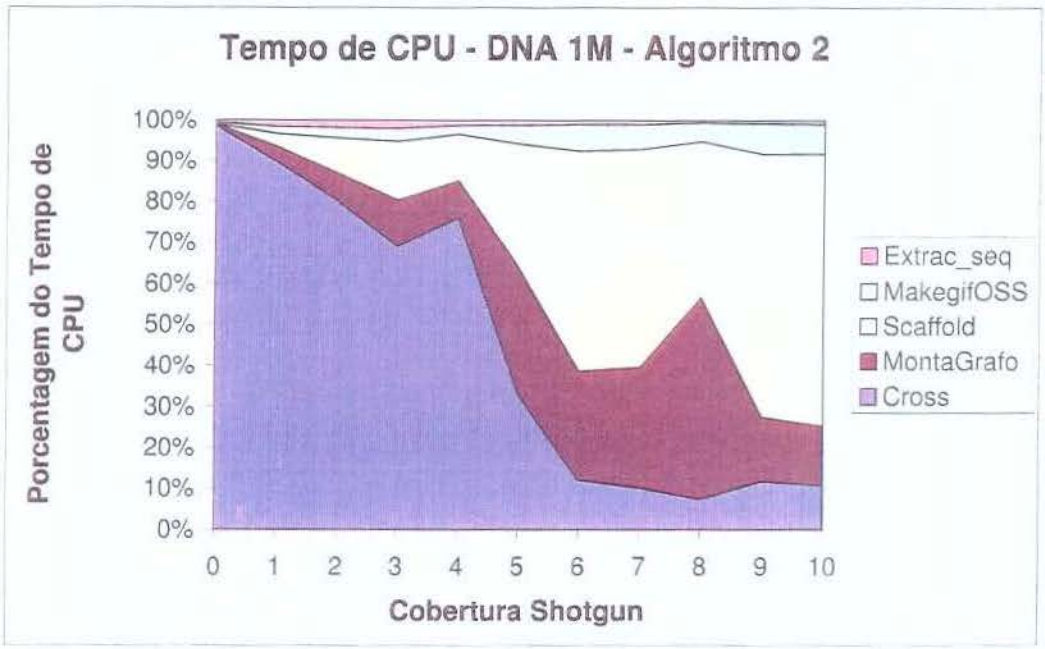

Figura B.68: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA1000K_125c40K_p500`.

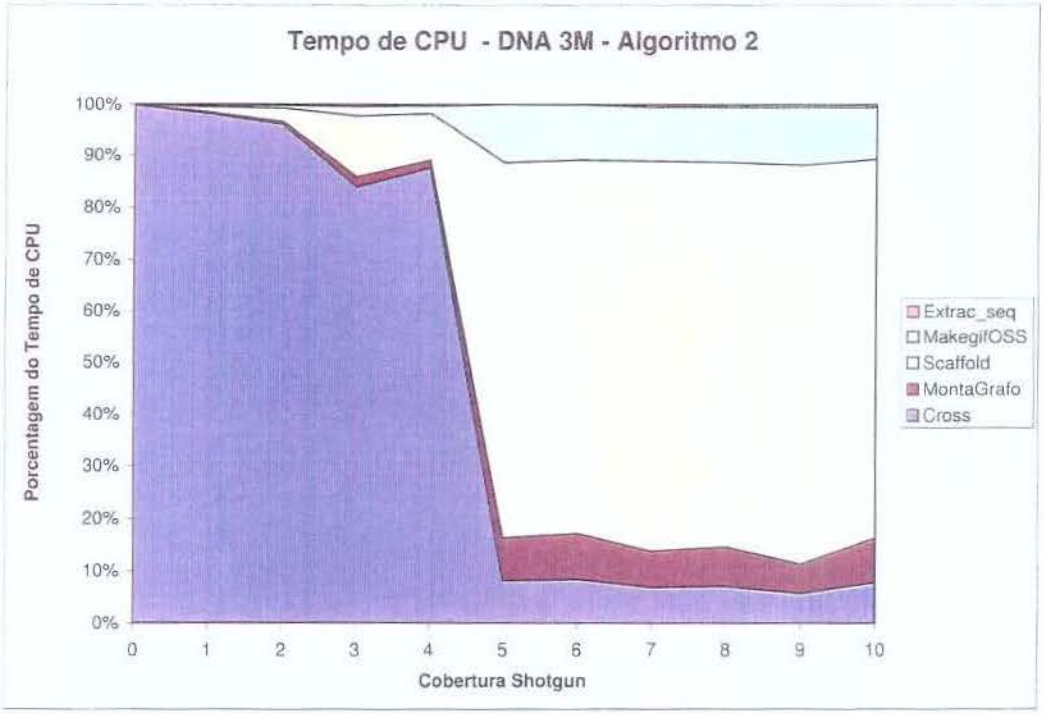

Figura B.69: Tempo percentual dos vários módulos do gerenciador com algoritmo 2 para `DNA3M_375c40K_p450`.

Bibliografia

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, New Jersey, USA, 1993.

[2] Frederick M. Ausubel et al. *A compendium of Methods from Current Protocols in Molecular Biology*, chapter Short Protocols in Molecular Biology. Greene Publishing Associates and John Wiley & Sons, USA, 2 edition, 1992. Harvard Medical School, by Current Protocols.

[3] Fábio Ribeiro Cerqueira. Montagem de fragmentos de DNA. Master's thesis, Universidade Estadual de Campinas, Campinas - SP, Janeiro 2000.

[4] Ellson Y. Chen, David Schlessinger, and Juha Kere. Ordered shotgun sequencing, a strategy for integrated mapping and sequencing of YAC clones. *Genomics*, 17:651–656, 1993.

[5] David P. Clark and Lonnie D. Russell. *Molecular Biology - made simple and fun.* Cache River Press, IL-USA, 1997.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill Book Company, USA, 1990. 15th printing, 1995.

[7] James Darnell, David Baltimore, et al. *Molecular Cell Biology.* Scientific American Books, New York, NY, 3 edition, New York, NY.

[8] Carlos E. Ferreira, Cid C. de Souza, and Yoshiko Wakabayashi. Rearrangement of DNA fragments: a branch-and-cut algorithm. Relatório técnico rt-mac-9701, USP - IME - DCC, Brasil, 1996.

[9] Michael R. Gary and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[10] Phil Green. Phrap documentation. http://www.phrap.org/.

[11] Phil Green. Against the whole-genome shotgun. *Genome Research*, 7:410–417, 1997.

[12] Eugene W. Myers et al. A whole-genome assembly of *Drosophila*. *Science*, 287:2196–2204, March 2000.

[13] Eugene W. Myers and James L. Weber. Is whole human genome sequencing feasible? In S. Suhai, editor, *Computational Methods in Genome Research*, pages 73–79. Plenum Press, New York, 1997.

[14] *Xanthomonas axonopodis* pv. *citri* genome project. http://www.lbi.ic.unicamp.br/. ONSA consortium.

[15] B. F. Francis Ouellette. The GenBank sequence database. In Andreas D. Baxevanis and B. F. Francis Ouellette, editors, *Bioinformatics: A Pratical Guide to the Analysis of Genes and Proteins*, chapter 2, pages 16–45. Wiley-Liss Inc., New York, 1998.

[16] João C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[17] João C. Setubal and Renato F. Werneck. A program for building contig scaffolds in double-barreled shotgun genome sequencing. Relatório Técnico 00-20, IC-Unicamp, Dezembro 2000.

[18] Andrew J. G. Simpson et al. The genome sequence of the plant pathogen *Xylella fastidiosa*. *Nature*, 406:151–157, July 2000.

[19] Maxine Singer and Paul Berg. *Genes & Genome – a changing perspective*. University Science Book. Backwell Scientific Publications, USA, California, 1992.

[20] Gautam B. Singh and Stephen A. Krawetz. CLONEPLACER: A software tool for simulating contig formation for ordered shotgun sequencing. *Genomics*, 25:555–558, 1995.

[21] David T. Suzuki, Anthony J. F. Griffiths, Jeffrey H. Miller, and Richard C. Lewontin. *Introdução à Genética*. Editora Guanabara Koogan S.A, Rio de Janeiro, RJ, 4 edition, 1992.