ADRIANO DONATO COUTO

PROPOSTA DE UMA NOVA ABORDAGEM PARA O PROCESSO DE MONTAGEM "DE NOVO" DE SEQUÊNCIAS DE DNA OBTIDAS DE SEQUENCIADORES DE NOVA GERAÇÃO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae.*

VIÇOSA
MINAS GERAIS – BRASIL
2014

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
C871p
2014

Couto, Adriano Donato, 1989-
Proposta de uma nova abordagem para o processo de montagem "de novo" de sequências de DNA obtidas de sequenciadores de nova geração / Adriano Donato Couto. – Viçosa, MG, 2014.
xii, 77f. : il. (algumas color.) ; 29 cm.

Inclui apêndices.
Orientador: Fábio Ribeiro Cerqueira.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f.67-71.

1. Bioinformática. 2. Genômica. 3. Sequenciamento de nucleotídeo. 4. Teoria dos grafos. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-graduação em Ciência da Computação. II. Título.

CDD 22. ed. 570.285

ADRIANO DONATO COUTO

PROPOSTA DE UMA NOVA ABORDAGEM PARA O PROCESSO DE MONTAGEM "DE NOVO" DE SEQUÊNCIAS DE DNA OBTIDAS DE SEQUENCIADORES DE NOVA GERAÇÃO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae.*

APROVADA: 27 de fevereiro de 2014.

Juliana Lopes Rangel Fietto

Ricardo dos Santos Ferreira

Fábio Ribeiro Cerqueira
(Orientador)

Dedico este trabalho àqueles que são minha base: meus pais, Adilson e Leida, e minha irmã, Larissa.

"Todas as vitórias ocultam uma abdicação"
(Simone de Beauvoir)

Agradecimentos

Primeiramente, agradeço a Deus pela inspiração e força nos diversos momentos que passei durante este trabalho. Em segundo lugar, agradeço aos meus pais, Adilson e Leida. Sem o apoio deles desde cedo, não teria chegado até aqui. Agradeço à minha irmã, pela parceria e companheirismo, pelos ouvidos abertos e pelos conselhos amigos. Reconheço aqui a paciência e a participação essenciais do meu orientador, Fabio Ribeiro Cerqueira, que soube lidar com os momentos diferentes pelos quais passei durante o curso. Sem sua ajuda, jamais teria conseguido finalizar este trabalho. Meu muito obrigado ao Lucas Vegi, que me deu a notícia da aprovação no mestrado e me orientou durante os primeiros momentos do curso. Agradeço à minha amiga Renata Moreira, que tornou a experência em Viçosa mais leve e tranquila com nossas conversas durante a hora do almoço. À amiga Mayara, aos professores e a outros amigos da graduação, deixo minha gratidão por terem me incentivado a fazer a inscrição e seguir com o processo seletivo para o mestrado. Agradeço à CAPES, pelo financiamento da minha pesquisa e ao Departamento de Informática da UFV, pela oportunidade concedida. Por fim, sou grato à Divisão de Apoio ao Desenvolvimento Científico e Tecnológico da UFV e à Dra. Ana Tereza Ribeiro de Vasconcelos, líder do Laboratório de Bioinformática do Laboratório Nacional de Computação Científica, por permitir o uso de seus servidores de alto desempenho. Todos os atores deste período da minha vida foram, cada qual à sua maneira, essenciais para que o processo acontecesse até o fim. A todos, um forte abraço!

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

COUTO, Adriano Donato, M.Sc., Universidade Federal de Viçosa, fevereiro de 2014. **Proposta de uma nova abordagem para o processo de montagem "de novo" de sequências de DNA obtidas de sequenciadores de nova geração**. Orientador: Fabio Ribeiro Cerqueira. Coorientador: Alcione de Paiva Oliveira.

O estudo de genomas trouxe muitos ganhos para a medicina, farmacologia e diversas outras áreas do conhecimento. Porém, muitos desafios também surgiram como consequência, em especial os computacionais. A grande massa de dados e sua complexidade para ser analisada tornam difícil o ato de extrair as informações necessárias. Como os sequenciadores de DNA não conseguem trabalhar com a molécula completa de uma vez, o sequenciador quebra a molécula e trabalha com diversos fragmentos, que precisam ser montados em um próximo passo. Nos sequenciadores de nova geração, o processo de sequenciamento é muito mais rápido e mais barato, mas também traz fragmentos muito menores do que aqueles obtidos na tecnologia Sanger. Além do maior número de fragmentos, a interferência dos erros de sequenciamento torna o processo complicado, sendo classificado como NP-Difícil. Para resolver esta situação, os montadores de fragmentos de DNA atuais executam uma série de pré e pós-processamentos nos dados gerados, a fim de eliminar ou pelo menos diminuir os desafios presentes. Neste trabalho, propõe-se uma nova abordagem de emparelhamento máximo de peso máximo (anteriormente proposta para fragmentos da tecnologia Sanger) em grafos k-mer, que retorna componentes mais simples de trabalhar (conhecidos como *unipaths*), junto com uma heurística para combinar esses componentes. Assim, busca-se fazer o processo de montagem em menos etapas. Este cruzamento trabalha com a extensão de caminhos através da combinação dos *unipaths*. Com um sistema de pontuação, o montador busca construir caminhos maiores e com o mínimo de áreas repetidas. Por se tratar do início de uma nova abordagem para fragmentos curtos, este trabalho não visa obter um

montador de DNA completo em sua primeira versão, mas validar o conceito proposto. Neste trabalho, foca-se especificamente na montagem de sequências obtidas de sequenciadores da nova geração. Durante os testes, percebeu-se que o protótipo consegue montar satisfatoriamente os genomas em metade dos casos, necessitando de alguns ajustes em próximas versões para os demais casos. Estes ajustes solucionariam problemas específicos que esta ferramenta ainda não soluciona. Levantou-se também que a cobertura de *reads* é fator determinante para bons resultados desta ferramenta. Há alguns trabalhos futuros necessários, como corte do gasto de memória, diminuição do tempo de processamento e comparação com outras ferramentas disponíveis atualmente.

Abstract

COUTO, Adriano Donato, M.Sc., Universidade Federal de Viçosa, February, 2014. **Proposal of a new approach for the "de novo"assembly process of DNA sequences from new generation sequencers**. Adviser: Fabio Ribeiro Cerqueira. Co-Adviser: Alcione de Paiva Oliveira.

The study of genomes brought a lot of gains to Medicine, Pharmacology and many other important science fields. However, many challenges emerged as a consequence, specially in Computer Science. The big amount of data and its complexity to be analyzed make arduous to extract the information needed. As the DNA sequencers are not able to get the sequence of the whole molecule at a glance, the sequencing process breaks the molecule and works with a lot of fragments. Those fragments need to be assembled in a next step. In the new-generation sequencers, the sequencing process is much faster and cheaper, but they return much smaller fragments compared to the output from Sanger technology. Besides the bigger set of fragments, the interference of sequencing errors complicates the process, classified as NP-hard. To handle this problem, the current DNA fragment assemblers run a lot of pre and postprocesses in the amount of data, aiming to eliminate or at least reduce the common problems. In this work, we propose a new approach of maximum-weighted maximum matching (first proposed for data from Sanger sequencing) for k-mer graphs that returns simpler components to work with (called unipaths), followed by an heuristic to combine those new components. Therefore, we want to do the assembly process in fewer steps. This crossing works with elongation of paths by the combination of unipaths. With a score system, the assembler tries to get longer paths combining the shorter ones, while minimizing the use of repeated regions. Because this is the beginning of a new approach for short fragments, it was not our intention to achieve a complete DNA assembler at its first version, but only to validate the proposed concept. In this work, we give focus specifically on the

assembly of sequences from new-generation sequencers. During the experiments, we identified that the assembler was able to return satisfatory results for half of the samples, but it needs some adjustments to improve the other results. Those adjustments would solve specific problems that this tool does not address yet. We also found out that the coverage by reads is determinant factor to get good outputs from this assembler. There are some future works that are necessary, such as decreasing memory usage and running time, as well as comparing this tool with other current available solutions.

Capítulo 1

Introdução

1.1 Motivação

O sequenciamento de genomas trouxe uma série de possibilidades para a Biologia Molecular. Como resultado, diversas áreas foram fortemente impactadas pelas novas descobertas possibilitadas pela genômica. Medicina, agricultura, pecuária, botânica, cosmetologia e farmacologia, para citarmos apenas algumas, não são as mesmas depois da revolução proporcionada pela gama de informações presentes no genoma que se tornam a cada ano mais acessíveis/interpretáveis.

Dentre todos os projetos de sequenciamento de DNA desenvolvidos, um dos mais notáveis, sem dúvida, foi o Projeto Genoma Humano. Com o estudo dos dados obtidos, torna-se possível levantar as causas de doenças e seus possíveis tratamentos (de cura ou paliativos), as relações de influência no indivíduo por fatores ambientais e de herança genética, além da evolução do ser humano [Venter et al., 2001].

Dentre as primeiras iniciativas de tecnologias para sequenciamento, destaca-se a iniciativa de Sanger et al. [1977]. Segundo [Men et al., 2008, p. 8], o custo para sequenciar 1 genoma de 1 Gb (bilhão de bases) seria de USD$10000000, com uma cobertura (número de cópias do mesmo DNA sequenciadas) de apenas 10 vezes. Já de acordo com Kircher & Kelso [2010], aplicando-se a tecnologia Sanger, o sequenciamento seria feito a um custo estimado de USD$500/Mb (milhão de bases) e a capacidade média de sequenciamento chegaria a 6 Mb por dia [Kircher & Kelso, 2010]. Nas tecnologias de sequenciamento de nova geração (NGS, na sigla em inglês), com maior capacidade de execução em paralelismo, chega-se a resultados em menor tempo e a um custo muito inferior. Ainda segundo Kircher & Kelso [2010], a tecnologia SOLiD (apenas como exemplo), consegue sequenciar aproximadamente 5000 Mb/dia a um custo próximo de USD$0,50/Mb.

Apesar das vantagens das tecnologias NGS, estas possuem como ponto fraco o tamanho reduzido dos fragmentos sequenciados. Enquanto a tecnologia proposta por Sanger pode sequenciar fragmentos de até 1000 a 2000 nucleotídeos [Pop, 2009], na nova geração os valores suportados são, em situações comuns, bem menores. A Tabela 1.1 apresenta o tamanho suportado por algumas tecnologias.

Tabela 1.1. Tamanho de *read* obtido em cada plataforma

Platforma[a]	Tamanho do *Read* (Bases)
Roche/454's GS FLX Titanium	330[b]
Illumina/Solexa's GAII	75 or 100
Life/APG's SOLiD 3	50
Polonator G.007	26
Helicos BioSciences HeliScope	32[b]

[a] Dados coletados da Tabela 1 em Metzker [2010].

Com fragmentos menores suportados na nova geração de sequenciadores, a complexidade do processo de montagem cresce de forma proporcional, já que são mais fragmentos e consequentemente um maior número de comparações entre eles. Além disto, com sequências reduzidas as chances de acontecerem situações ambíguas na ordem em que as sequências são remontadas é muito maior. Com isso, novas ferramentas de montagem precisaram ser desenvolvidas, enfrentando este novo cenário. Não obstante, os montadores precisam administrar topologias problemáticas nos grafos utilizados para modelar as relações entre os *reads* (fragmentos de sequência que o montador recebe como dados de entrada) durante o processo. As topologias são geradas por zonas repetidas do genoma, polimorfismos ou erros de sequenciamento [Miller et al., 2010], que afetam a complexidade da estrutura de grafos e, consequentemente, tornam o processo de montagem mais custoso computacionalmente.

Basicamente, segundo Pop [2009], há dois tipos de montagem de fragmentos de DNA. O primeiro é o resenquenciamento, que ocorre quando o pesquisador tem à disposição durante a montagem de uma espécie X_a o genoma previamente montado de uma outra espécie X_b e ambas são notavelmente similares. Sendo assim, parte do processo de montagem está na comparação de áreas semelhantes dos dois genomas para alinhamento das sequências. O segundo tipo de montagem é conhecido como *de novo* e é o foco deste trabalho. Nela, apenas a informação obtida dos sequenciadores de DNA é conhecida, sem informação extra de outras espécies. A montagem *de novo* é considerada um problema NP-Difícil [Pop, 2009]. Como apontado por Molaei et al. [2013], um problema NP-Difícil possui um campo de busca (conjunto de resultados

possíveis) muito extenso, sendo problema intratável em computadores tradicionais (baseados em circuitos de silício), ou seja, sem solução conhecida em tempo viável.

Para tratar as topologias problemáticas, importantes montadores para fragmentos NGS, como ALLPATHS [Butler et al., 2008], Euler [Pevzner et al., 2001] e Velvet [Zerbino & Birney, 2008], executam diversos pré e pós-processamentos nos *reads*, tornando o processo de montagem mais custoso.

A grande quantidade de dados produzidos pela tecnologia NGS, o alto custo computacional de se manipular um número muito maior de fragmentos para montagem se comparado com a primeira geração e as dificuldades de se manipular as topologias problemáticas tornam necessária a criação de uma abordagem que busque a simplificação do processo de montagem. Neste trabalho, propõe-se uma abordagem que unifica esse processo, evitando atividades extras durante a montagem. Uma simplificação no número de passos oferece também simplificação para se obter o genoma montado, com menos parâmetros técnicos sendo fornecidos pelo usuário. O foco desta proposta é o conjunto de dados de sequenciadores de nova geração, mais especificamente na montagem *de novo* de fragmentos de DNA.

1.2 Objetivos

Este trabalho tem como objetivo o desenvolvimento de um protótipo com nova abordagem para montagem de fragmentos de DNA obtidos de sequenciadores de nova geração que evite pré e pós-processamentos desnecessários.

Como objetivos específicos, este trabalho propõe:

- Definir uma estratégia de montagem em menos etapas;
- Desenvolver o protótipo de montador; e
- Validar a proposta com espécies selecionadas.

1.3 Organização deste Documento

Para uma melhor compreensão deste trabalho de pesquisa, o documento foi dividido em diferentes seções. O capítulo 2 apresenta os conceitos necessários para a contextualização desta dissertação e o passo-a-passo do levantamento de trabalhos relacionados, com os resultados encontrados. No capítulo 3, o protótipo desenvolvido e as etapas executadas durante a montagem são apresentadas. O capítulo 4 descreve os resultados obtidos com os experimentos em genomas selecionados. Uma

discussão sobre os resultados obtidos é promovida no capítulo 5. O capítulo 6 (Conclusões e Perspectivas) disserta sobre o ponto atual da pesquisa e indica pontos para a continuidade do trabalho.

Capítulo 2

Revisão Bibliográfica

Neste capítulo, a seção 2.1 aborda alguns conceitos básicos. Essas definições são essenciais para compreender as tecnologias e os termos citados nas seções 2.2 e 2.3.

Na seção 2.2, uma definição geral de como o processo de sequenciamento ocorre nas diversas tecnologias é apresentada, com alguns exemplos. Na seção 2.3, há uma conceituação dos tipos de abordagem gerais para montagem de DNA. Por fim, a seção 2.4 apresenta alguns detalhes específicos de tecnologias em destaque na área.

2.1 Conceitos Básicos

2.1.1 DNA e Genoma

O ácido desoxirribonucleico (DNA, na sigla em inglês) é um composto químico que contém todas as informações necessárias para manutenção da vida da maioria dos seres vivos. Ele é composto por quatro unidades químicas, que são representadas por letras. Essas unidades, nomeadas bases nucleotídicas, são chamadas de Adenina (representada pela letra A), Timina (letra T), Guanina (G) e Citosina (C). A ordem dessas bases na molécula de DNA determina a função de um determinado trecho desse e as informações armazenadas no mesmo. O conjunto completo do DNA de um organismo é seu genoma (adaptado de National Human Genome Research Institute [2011]).

Apesar de haver DNA na forma de fita única, sua forma mais estável é na forma de duas fitas complementares unidas [Scitable, 2013]. Nesse contexto, a Adenina em uma fita se liga por pontes de hidrogênio à Timina na fita complementar (e vice-versa), já a Citosina se liga à Guanina na fita complementar. A Figura 2.1 apresenta uma representação gráfica simplificada de uma fita dupla de DNA.

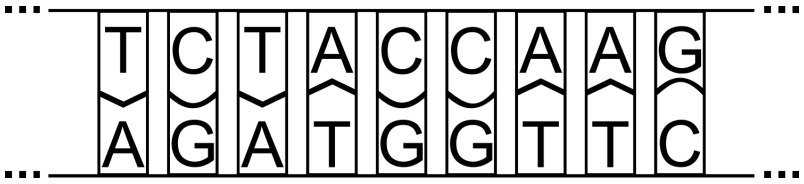

Figura 2.1. Representação de uma fita de DNA. Cada letra representa uma das quatro bases nucleotídicas. Como pode-se observar, as ligações de uma fita com a sua fita complementar só acontecem no padrão `A`–`T`, `T`–`A`, `C`–`G` e `G`–`C`.

Toda fita de DNA possui uma extremidade anterior chamada $5'$ e outra posterior chamada $3'$. A fita padrão tem orientação $5' \rightarrow 3'$ e seu complemento reverso é sempre representado na orientação $3' \rightarrow 5'$ [Scitable, 2013].

2.2 Sequenciamento de DNA

O sequenciamento de DNA consiste na determinação da sequência de nucleotídeos de uma molécula de DNA. Conforme já citado no Capítulo 1, as tecnologias de sequenciamento possuem limitação no tamanho dos fragmentos que processam, não conseguindo ainda sequenciar genomas complexos de forma única [Pop, 2009; Miller et al., 2010]. Sendo assim, uma etapa de montagem dos fragmentos sequenciados se torna necessária.

A primeira tecnologia de sequenciamento amplamente aceita foi a proposta por Sanger et al. [1977]. Neste método, são utilizados, além de fragmentos de DNA:

***Primer*:** pequeno bloco de DNA utilizado para demarcar o início de um trecho replicado de DNA;

DNA polimerase: enzima utilizada para copiar uma fita de DNA;

Deoxinucleotídeos: são unidades químicas que, quando presentes no DNA, formarão as bases nucleotídicas (ver seção 2.1.1);

Terminadores de cadeia: (dideoxinucleotídeos): quatro terminadores, um equivalente a cada base nucleotídica. Essas substâncias encerram a cópia de DNA, limitando o tamanho de um fragmento.

Durante o processo, a fita de DNA é colocada em 4 tubos diferentes, junto com os 4 tipos de deoxinucleotídeos, o *primer* e um dos quatro terminadores. A Figura 2.2 demonstra o processo de sequenciamento na tecnologia proposta por Sanger et al. [1977].

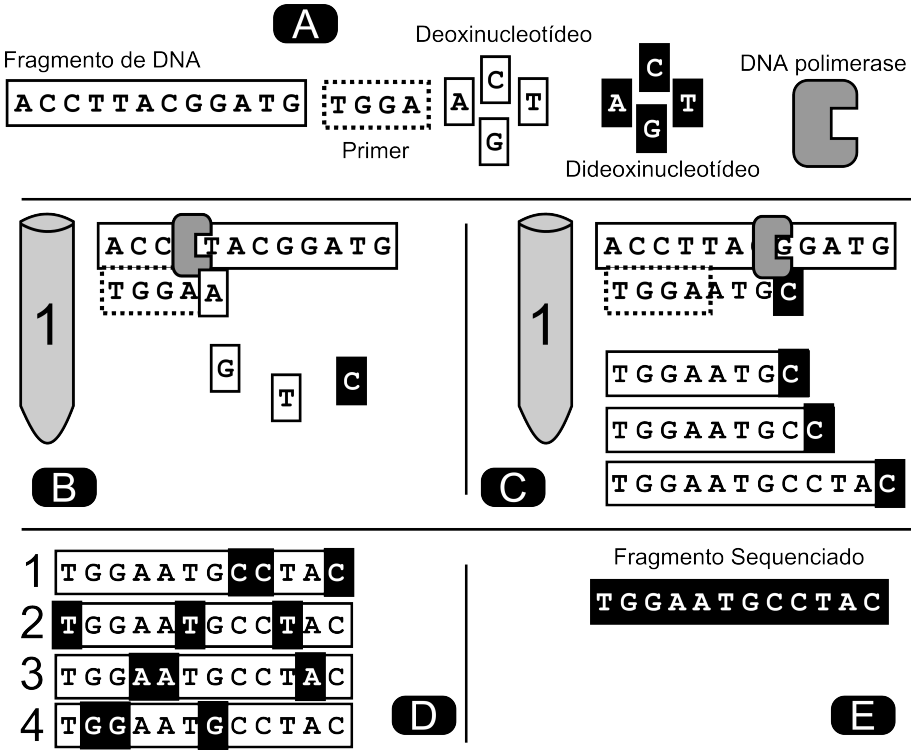

Figura 2.2. Representação do método de sequenciamento proposto por Sanger (exemplo com um dos quatro tubos).

A Figura 2.2 apresenta em (A) os elementos participantes do processo de sequenciamento na tecnologia Sanger em um dos quatro tubos (no exemplo, tubo "1"). Em (B), o *primer* é conectado à fita de DNA e a cópia começa a partir dele, com a DNA polimerase. Deoxinucleotídeos são unidos em cadeia, respeitando a fita original. Como apresentado em (C), um dideoxinucleotídeo é ocasionalmente conectado à fita e, como não possui capacidade de ligação com próximos deoxinucleotídeos, o fragmento copiado se encerra neste ponto. Com as operações em paralelo no mesmo tubo, tem-se fragmentos de diversos tamanhos, todos com o terminador dideoxinucleotídeo fornecido no tubo 1. Cada um dos quatro tubos produz fragmentos com seu respectivo terminador presente em cada ponto possível, no passo (D). Em (E), como os terminadores são marcados pelo sequenciador, este consegue descobrir, com a leitura dos resultados dos quatro tubos, a sequência do trecho de DNA que foi submetido ao processo de sequenciamento. É necessário observar que, durante o sequenciamento, as fitas-cópia geradas possuem as bases complementares às bases nucleotídicas presentes na fita original. Ou seja, são sua complementação reversa (ver tópico 2.1.1).

Dentro dos tubos, a cópia é iniciada no ponto indicado pelo primer e a en-

zima segue conectando os deoxinucleotídeos na ordem permitida pela fita original de DNA. Quando a enzima conecta acidentalmente o terminador, este encerra o processo de cópia e o fragmento está pronto. Como os terminadores são inseridos nos tubos na proporção de 1 para 100 em relação à quantidade de deoxinucleotídeos, produz-se diversos fragmentos em cada tubo, cada um com um tamanho diferente. Organizando-se os fragmentos de acordo com o seu tamanho, tem-se então a base no fim de cada fragmento representando a sequência da fita original.

Estas fitas de DNA são, em passos seguintes, combinadas para formar a sequência completa (processo de montagem).

Como citado por Pop [2009], nos últimos anos surgiram importantes sequenciadores de nova geração (NGS, na sigla em inglês). Estes novos métodos seguem uma sequência de passos parecida, chamada *shotgun*, que foi proposta por Staden [1979]. Nesse processo, há o emprego de sequenciamento com posterior processamento computacional para montagem. Primeiramente, segundo Pop [2009], há a quebra da molécula em fragmentos com tamanho limitado. Esses fragmentos são então sequenciados e a informação obtida é utilizada como entrada em montadores de genoma. O que difere cada uma é a maneira como cada passo do sequenciamento é executado. Apesar disso, o fator comum — e que diferencia estas tecnologias da ideia utilizada na primeira geração de sequenciadores — é o emprego de um alto grau de paralelismo, onde milhões de fragmentos são sequenciados ao mesmo tempo. Apesar da vantagem do paralelismo, essa técnica trouxe a limitação de fragmentos muito menores, geralmente entre 25 e 400 bp (pares de bases). Essa diminuição no tamanho (e consequente aumento na quantidade) de fragmentos faz com que a complexidade do processo de montagem seja bem maior do que na primeira geração de ferramentas. Em analogia proposta por [Baker, 2012], o método de Sanger, com fragmentos maiores, fornecia um "quebra-cabeça" com cerca de 30 milhões de peças; já na nova geração, algo em torno de 2 a 3 bilhões de peças, com 100 cópias cada uma. Embora preserve uma razão parecida na quantidade de fragmentos, o número total em casos reais varia de acordo com o tamanho do genoma.

2.3 Montagem de Fragmentos de DNA

Em relação à montagem de fragmentos de DNA, há duas possibilidades distintas [Pop, 2009; Zhang et al., 2011; Paszkiewicz & Studholme, 2010]. Na primeira, a montagem é feita apenas com os *reads* obtidos dos sequenciadores; na segunda, há um genoma de referência, ou seja, há uma espécie similar com o genoma já montado

e, portanto, diversas áreas podem ser montadas apenas por comparação entre o material genético de ambas.

Em relação ao processo de montagem de genomas, Conway & Bromage [2011] cita duas abordagens principais, sendo elas os grafos *de Bruijn* e a estratégia de *overlap-layout-consensus* (utilizando grafos de sobreposição). Porém, para Schatz et al. [2010] e Miller et al. [2010], há uma terceira abordagem, em que não aplica as estruturas de grafo propriamente ditas, mas algoritmos gulosos. Por sua vez, Nagarajan & Pop [2013] e Paszkiewicz & Studholme [2010] defendem que há ainda um quarto tipo (derivado do grafo de sobreposição), chamado grafo de *strings*. Estas técnicas serão descritas a seguir.

2.3.1 *Unpaired Reads*, *Paired-End Reads* e *Mate-Pairs*

A forma mais básica de montagem de fragmentos de DNA utiliza *reads* sem informação de localização. Chamados de *unpaired* (não pareados), estes *reads* possuem apenas a própria sequência (e em alguns casos a qualidade de cada base), sem informação de localização. ALLPATHS [Butler et al., 2008] e outros montadores trabalham com *reads* nesse padrão. Alguns montadores trabalham também com um segundo tipo, os *paired-end reads*, que são pares de *reads* extraídos de cada uma das pontas de um fragmento de DNA e então sequenciados. Com sua distância aproximada conhecida e a posição relativa entre eles, ajudam na resolução de *repeats* e outros problemas que os *unpaired reads* geralmente não possibilitam corrigir. Como apontado por Kircher & Kelso [2010], esse par de *reads* pode ser obtido de fragmentos com até 1kb (mil bases) de tamanho. *Mate-pairs*, com ideia parecida mas outro protocolo de sequenciamento, são capazes de sequenciar as extremidades de moléculas com tamanhos definidos maiores, como 8 ou 20 kb. Neste trabalho, os dados sequenciados suportados serão do tipo não pareado.

2.3.2 Algoritmo Guloso

Nos montadores que utilizam estratégias de algoritmos gulosos, cada *read* é comparado com os demais em busca de sobreposição, mas a construção dos *contigs* começa de um único fragmento. Diferenças pequenas nas sequências são aceitas, já que pode ter ocorrido erro no sequenciamento [Schatz et al., 2010; Miller et al., 2010; Paszkiewicz & Studholme, 2010].

Primeiramente, todas os pares possíveis de fragmentos são calculados, ou seja, todas as sobreposições entre dois *reads*. Como é comum dos algoritmos gulosos em

geral, começa-se então com um fragmento. A partir dele, avalia-se entre os seus vizinhos possíveis aquele que tenha sobreposição com maior pontuação. Essa pontuação leva em consideração o número de bases comuns na sequência de ambos. Encontrado este fragmento, faz-se a conexão entre eles e busca-se o fragmento que tenha a maior sobreposição com o segundo *read*. O processo é repetido até que nenhuma outra extensão seja possível [Miller et al., 2010]. Os reads com as sobreposições mais longas são então concatenados, ou seja, passam a formar uma sequência única. Os *contigs* (sequências contínuas) passam pelo mesmo processo dos *reads* iniciais e o passo é repetido até todas as sobreposições estarem processadas.

Como os mesmos autores afirmam, este método consegue resolver satisfatoriamente a montagem de genomas mais simples, mas não resolvem totalmente diante de genomas que possuem áreas repetidas que sejam mais extensas do que a extensão dos *reads*. Como cita [Nagarajan & Pop, 2013], esta estratégia foi bastante utilizada nos primeiros montadores, mas também é utilizada em algumas ferramentas recentes, como o montador VCAKE [Jeck et al., 2007]. Esta abordagem não é amplamente utilizada por sua atuação local, que não consegue utilizar informações de maior abrangência, como *mate-pairs* de longa distância, para resolver *repeats*.

2.3.3 Grafos de Sobreposição (*Overlap-layout-consensus*)

Nos grafos de sobreposição, segundo diversos autores [Pop, 2009; Schatz et al., 2010; Miller et al., 2010; Conway & Bromage, 2011], cada *read* funciona como vértice. É a sobreposição entre estes fragmentos que determina a conexão entre eles. A Figura 2.3 mostra um exemplo de um grafo de sobreposição.

Primeiramente, há um processo de comparação par a par de *reads*, ou seja, são realizadas n^2 comparações para n *reads*. Cada par possível é conectado por uma aresta. Para este passo, há diversas estratégias de cálculo da sobreposição. Na estratégia de *seed & extension* (semear e extender, em tradução livre), há primeiramente um cálculo de áreas de tamanho K em todos os *reads*. Com essas áreas selecionadas, chamadas K-mers, todo vértice que compartilhe desses K-mers de sobreposição com outro vértice é conectado com esse último. É importante não misturar o conceito de K-mer abordado neste tópico com o grafo k-mer, abordado no tópico 2.3.5. A Figura ***** apresenta um exemplo deste processo.

A identificação de sobreposição, como lembra Miller et al. [2010], é influenciada por três parâmetros essenciais: o tamanho do K-mer, o tamanho da área mínima de sobreposição e o grau mínimo de semelhança na sobreposição, isto é, há um limite permitido no número de diferenças dentro de uma área de sobreposição. Como esta

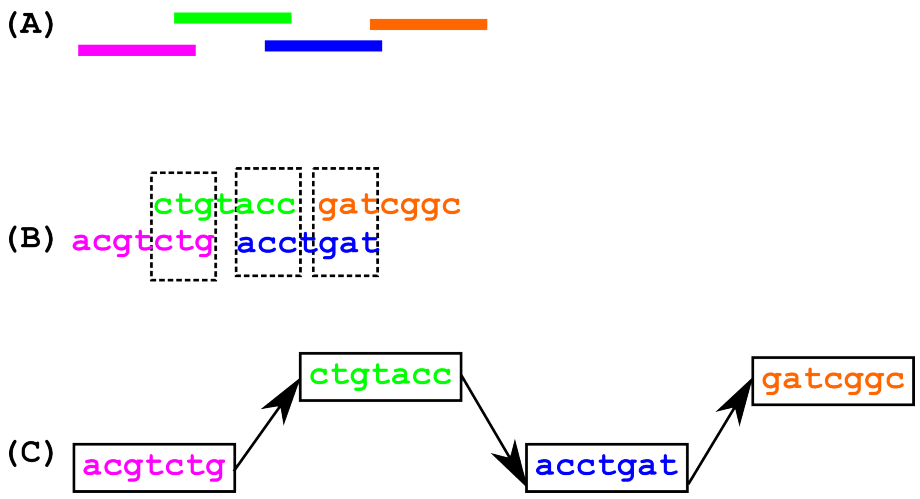

Figura 2.3. Exemplo da criação de um grafo de sobreposição. (A) Fragmentos de DNA. (B) *Reads* correspondentes de cada fragmento com suas áreas de sobreposição destacadas com quadros de borda tracejada. (C) Grafo de sobreposição correspondente ao conjunto de *reads* dado. Este exemplo é hipotético. Em casos reais, o tamanho dos *reads* e sua quantidade são maiores.

abordagem lida com relações de similaridade previamente calculadas, há a vantagem de não ter de trabalhar em memória diretamente com as sequências durante a construção ou manipulação do grafo de sobreposição, economizando memória. Apesar disso, a fase de construção dos *contigs* necessita do carregamento das sequências em memória. A Figura 2.4 apresenta um exemplo do cálculo de sobreposição neste tipo de grafo.

Na Figura 2.4, (A) apresenta três *reads* (r_1, r_2 e r_3) e três K-mers gerados a partir deles (u_1, u_2 e u_3). Baseados nestes K-mers, é possível calcular as taxas de sobreposição (So) e a semelhança da área de sobreposição entre dois *reads*. Em (B), a sobreposição está no limite aceitável (igual ou maior do que So_{min} e a semelhança também está acima do mínimo. Em (C), os *reads* r_2 e r_3 apresentam área de sobreposição adequada, mas a taxa de semelhança está abaixo do aceitável. Assim, o grafo de sobreposição resultante isola o *read* r_3, que não possui os valores mínimos de sobreposição e semelhança com nenhum outro *read*.

A solução para obtenção de todos os *contigs* poderia ser alcançada por meio do alinhamento de múltiplas sequências (em inglês, *multiple sequence alignment*, ou MSA). Porém, como o autor também lembra, não há solução ótima viável neste sentido. Esta solução é equivalente, segundo Paszkiewicz & Studholme [2010], ao problema de se encontrar um circuito Hamiltoniano no grafo, onde se deseja encontrar um caminho que passe por todos os vértices somente uma vez. Este problema é classificado como NP-difícil.

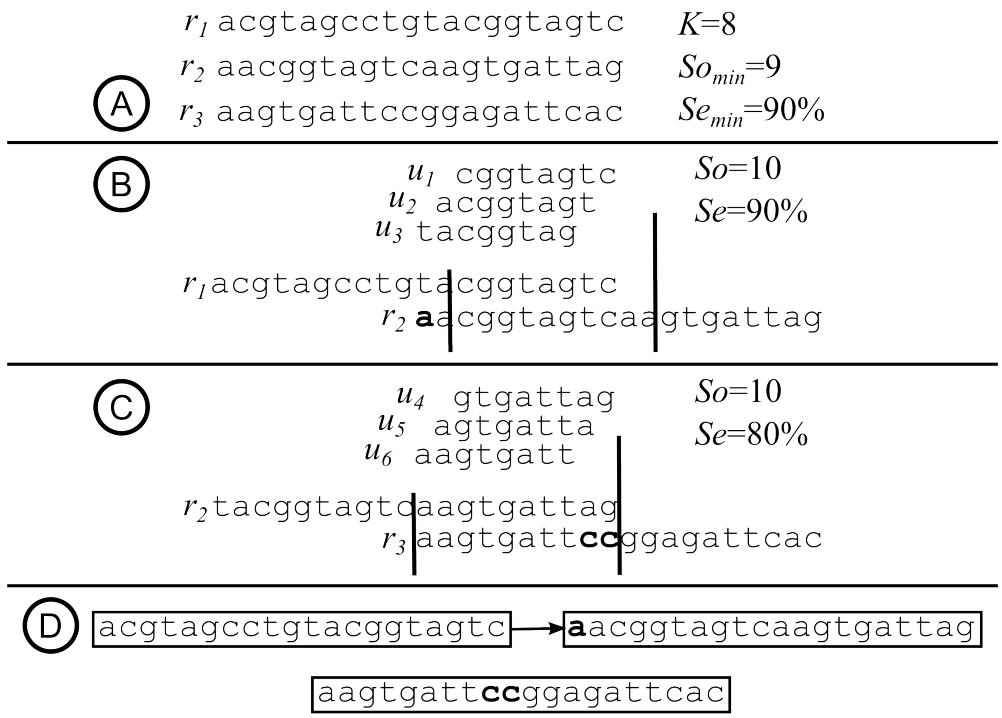

Figura 2.4. Cálculo da sobreposição entre *reads* no grafo de sobreposição. K é o tamanho dos blocos (K-mers), So_{min} é o tamanho mínimo de sobreposição e Se_{min} é o percentual de semelhança entre os trechos de sobreposição entre dois *reads*. Realce em negrito para trechos divergentes.

Esse tipo de abordagem foi muito comum nos primeiros montadores, geralmente aplicados a casos em que se tem fragmentos mais longos e em menor número [Conway & Bromage, 2011; Nagarajan & Pop, 2013]. Algumas ferramentas ainda o utilizam, mas o uso do grafo k-mer é mais frequente (seção 2.3.5), já que nele se evita as comparações diretas entre os *reads*.

2.3.4 Grafo de *String*

Grafo de string é uma outra abordagem para montagem de genomas [Nagarajan & Pop, 2013; Paszkiewicz & Studholme, 2010]. Este método simplifica o grafo geral de sobreposição pela eliminação de redundância. A execução é feita em quatro passos (segundo Paszkiewicz & Studholme [2010]):

- Geração de um grafo de sobreposição através da comparação par a par de todos os vértices;
- Conversão para grafo de *string*, através da fusão e redução de sobreposições e arestas redundantes;
- Eliminação de arestas e vértices falsos através de um algoritmo de fluxo de rede;
- Determinação de um caminho ou circuito Euleriano (sobre caminho Euleriano, ver seção 2.3.5).

Esta abordagem é usada apenas para montagem de fragmentos mais longos e em menor quantidade. Foi proposta por [Myers, 1995] e teve sequência com um trabalho mais recente [Myers, 2005].

2.3.5 Grafos *de Bruijn*

O grafo *de Bruijn*, também conhecido como grafo k-mer, também foca na sobreposição de *reads*, mas não faz a comparação par a par para isso. Cada autor utiliza uma das duas nomeações já citadas deste grafo, mas a definição permanece similar, com pequenas adaptações dependendo da tecnologia. Além disso, vale lembrar que este padrão de grafo é mais empregado em conjuntos de dados com *reads* curtos [Conway & Bromage, 2011; Nagarajan & Pop, 2013].

Conforme abordado por Nagarajan & Pop [2013], Pop [2009], Miller et al. [2010], Schatz et al. [2010] e Conway & Bromage [2011], este tipo de abordagem não representa diretamente os *reads*. Ao invés disso, cada fragmento é representado por inúmeras subsequências de tamanho fixo k. Cada fragmento de tamanho k possui sobreposição de $k-1$ caracteres com seus vizinhos. Conforme observado por Nagarajan & Pop [2013], neste modelo são permitidas somente sobreposições exatas, sem nenhum caractere divergente. Logo, correções de erros precisam ser executadas antes da montagem e também durante esta etapa para que se obtenha consensos (resultados da montagem) de boa qualidade. Nagarajan & Pop [2013] defende que esta abordagem deve provavelmente perder a importância à medida em que fragmentos fiquem mais longos e com melhor acurácia.

Durante a construção do grafo, como apresentado na Figura 2.5, cada *read* (a) é transformado em palavras de tamanho fixo igual a k. Essas "sub-palavras"são geradas iniciando-se de cada letra da sequência do *read*, copiando um trecho de k bases para cada uma (b). *Reads* que compartilham k-mers são representados de

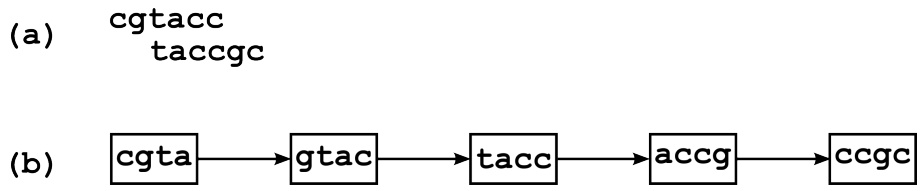

Figura 2.5. Montagem hipotética de um grafo *de Bruijn*. (A) *Reads* de entrada. (B) Representação do grafo *de Bruijn* baseado nos fragmentos dados, com $k = 4$. Em casos reais, o tamanho dos *reads* e o valor de k são maiores e variam.

forma unificada, sendo que a cada nova ocorrência de um dado k-mer, este tem seu peso incrementado. Este peso representa o número de vezes que ele apareceu nos *reads*.

Miller et al. [2010]; Pop [2009] citam que, em um cenário perfeito (livre de erros de sequenciamento), o grafo k-mer apresenta, além da possibilidade de abordagem via circuito Hamiltoniano dos grafos de sobreposição (ver seção 2.3.3), a possibilidade de montagem buscando-se o caminho Euleriano, ou seja, um caminho onde todas as arestas são visitadas uma única vez. Encontrar um caminho Euleriano é trivial e é a base da proposta do montador EULER, proposto por Pevzner et al. [2001]. Para remover todos os erros e levar o grafo obtido em situações comuns para a forma de Bruijn (grafo Euleriano), porém, uma complexa correção de erros deve ser aplicada [Pop, 2009].

Apesar do cenário perfeito em que se pode buscar o caminho Euleriano, na situação real o grafo apresenta diversos complicadores devido a características naturais do sequenciamento ou do próprio genoma [Miller et al., 2010]. Nesse sentido, o autor cita algumas situações que aumentam a complexidade do grafo, chamadas topologias problemáticas:

Erosão (*spurs*): pequena divergência em relação ao caminho principal, sem continuação. São provocadas por erros de sequenciamento em uma das extremidades de um *read*, mas podem também ser causadas por cobertura próxima de zero em regiões de polimorfismo que ocorram logo após áreas de *repeat*.

Bolha (*bubble*): caminho que se divide em dois e depois converge novamente no mesmo caminho. São causadas por erros de sequenciamento ou por polimorfismo (diferença de uma ou poucas bases), ambos no meio de *reads*.

Corda Desgastada (*frayed rope*): caminhos que convergem e novamente se dividem. São criadas quando há áreas de *repeat* (repetições) no genoma.

Ciclo: caminho que converge em si mesmo. Criado por regiões repetidas no genoma.

A Figura 2.6 apresenta um exemplo de cada topologia problemática.

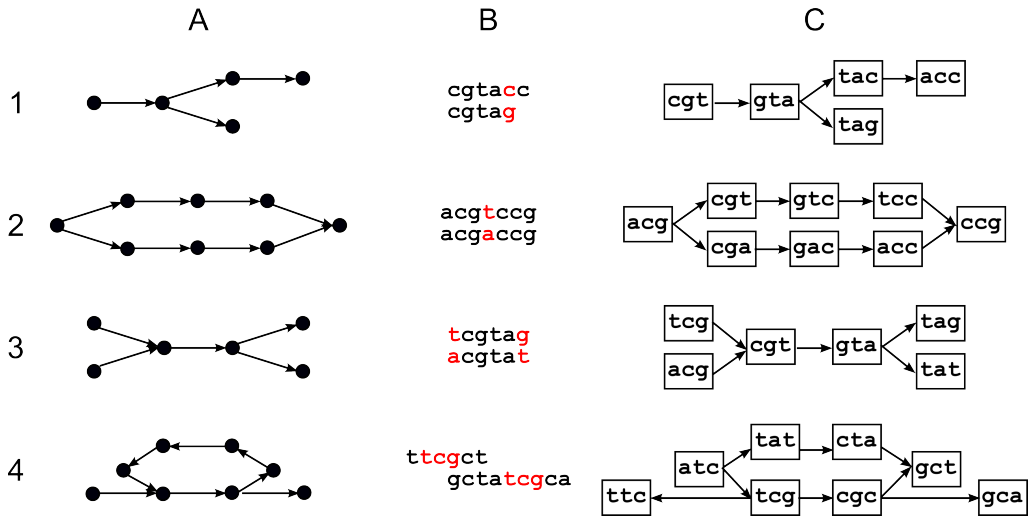

Figura 2.6. Topologias Problemáticas em Grafos k-mer. Em cada linha, tem-se uma topologia problemática (1–4). Em (1), pode-se ver a topologia de "Erosão"; em (2), há um exemplo de "Bolha"; em (3), um caso de "Corda Desgastada"; e em (4), ciclo. Nas colunas, observa-se: (A) Representação em grafos das topologias. (B) Sequências simuladas que causam as topologias, com trechos que as causam em vermelho. (C) Grafo k-mer correspondente aos *reads* dados em (B).

2.4 Principais Montadores de Fragmentos de DNA

Os principais montadores serão apresentados nos tópicos seguintes, levando-se em conta duas coletâneas de artigos [Miller et al., 2010; Nagarajan & Pop, 2013]. A partir destas revisões de área, o artigo de cada ferramenta foi levantado e todos os montadores analisados são apresentados a seguir, separados de acordo com a metodologia geral utilizada em cada um.

Como defendido por Nagarajan & Pop [2013], os montadores geralmente aplicam uma tecnologia de montagem de acordo com o tipo de *read* que eles manipularão. Para dados com maior acurácia e de fragmentos mais curtos (geralmente até 100 bases), a abordagem com o grafo de Bruijn é frequentemente utilizada. Para dados com menor acurácia e de fragmentos mais extensos, porém, a abordagem com grafos de sobreposição ou de *string* é mais frequente.

2.4.1 Ferramentas que se Baseiam em Algoritmos Gulosos

2.4.1.1 Phrap

Baseado na filosofia dos algoritmos gulosos (seção 2.3.2, página 9), este montador foi proposto por Green [1994] e faz parte do pacote Phrap/Phred/Consed [de la Bastide & McCombie, 2007].

Uma série de passos é executada durante a montagem de um genoma, como citado por de la Bastide & McCombie [2007]. Primeiramente, Phrap compara pares de *reads*, buscando regiões parecidas seguindo critérios estabelecidos (tamanho, taxa de erro máxima). Quando encontra um par, tenta comparar o segundo membro do par de modo a estender a sequência. À medida que fragmentos são sobrepostos, novos *contigs* são obtidos. Trechos que representem vetores utilizados durante o sequenciamento são marcados com a letra N, para que não sejam considerados durante a montagem e também não interfiram no alinhamento dos *reads*. Como é aceito que as sequências podem possuir erros de sequenciamento, uma taxa de erro (percentual de bases diferentes na zona de sobreposição) é configurada, visando não permitir alinhamentos errôneos e, ao mesmo tempo, tentando pegar o máximo de alinhamentos válidos.

Para contornar cada região de diferença nas sobreposições e construir o consenso final, Phrap utiliza as qualidades aferidas a cada base pela ferramenta Phred. Estes valores são combinados a demais parâmetros para criar uma pontuação para cada base do *contig*. A taxa de cobertura de um trecho, a cobertura ou não em ambas as fitas de DNA e outros fatores fazem parte do cálculo.

Por fim, áreas previamente marcadas como sequências do vetor utilizado durante a cópia do genoma são marcadas e ignoradas durante o processo de construção da sequência final. Uma diferença interessante desta ferramenta para as demais é que os *contigs* inicialmente propostos não são conjuntos de *reads*, mas conjunto de áreas de alta qualidade de cada *read* agrupado.

2.4.1.2 SSAKE

SSAKE foi desenvolvido por Warren et al. [2007] e trabalha com *reads* curtos, geralmente de 25 bases. Assim que lidos, os dados são cadastrados em uma tabela *hash*, que possui como chave da busca a sequência de cada *read* e, como valor retornado, o número de vezes que aquela sequência aparece nos dados. Este montador se baseia na ideia de algoritmo guloso (seção 2.3.2).

Uma árvore de prefixos é utilizada neste montador, posicionando as sequências

(e seus complementos) de acordo com suas 11 primeiras bases, partindo da extremidade 5′. Os *reads* são ordenados tomando-se como comparativo a ocorrência destes, partindo do mais frequente para o menos frequente. Isto ajuda, segundo o autor, a diminuir o uso de *reads* com erro de sequenciamento nos *contigs*.

No passo seguinte, cada *read* ainda não montado é utilizado como início de um *contig*. Um número de blocos de tamanho fixo k é lido na extremidade 3′. Esta sequência é procurada em outros *reads* a fim de encontrar um fragmento que tenha sobreposição com um limite mínimo de tamanho permitido. Em caso positivo, os caracteres após essa área comum são adicionados ao *contig*.

No final, o montador retorna os contigs construídos e as sequências que não foram conectadas a nenhum *contig*.

2.4.1.3 SHARCGS

Proposto por Dohm et al. [2007], este montador utiliza uma estratégia de algoritmo guloso [Miller et al., 2010] e visa especificamente *reads* de 25–40 bases. Ele consiste, basicamente, de três passos: filtragem de *reads* com erros de sequenciamento, montagem de *contigs* e fusão desses, para criar as sequências finais. Miller et al. [2010] defende que SHARCGS é basicamente a ferramenta SSAKE com adição de etapas de pré e pós-processamento.

Na fase de filtragem, há dois filtros-padrão. No primeiro filtro, há duas formas de se avaliar o *read*. Caso não esteja presente a qualidade das bases, a quantidade de cópias geradas deste *read* é considerada, tendo que respeitar um parâmetro de valor mínimo. Quando a entrada de dados fornece a qualidade das bases da sequência, porém, é considerado um *read* válido todo aquele que possua altos índices de qualidade (em ambas as fitas do DNA), respeitando um parâmetro de valor mínimo. No segundo filtro, busca-se os *reads* que possuam sobreposição em ambos os extremos de pelo menos metade da extensão deste *read*. Caso contrário, este pode ser descartado. Depois desta filtragem, apenas uma cópia de cada *read* é mantida e seu complemento reverso é criado.

Durante a criação de *contigs*, um esquema de "árvore de prefixos" é formado. Começando por um *read* (novo *contig*), o algoritmo busca um novo fragmento que tenha um prefixo de tamanho mínimo que seja comum ao final do *contig*. Se for, este é conectado ao *contig* provisório. Quando não é mais possível alongá-lo, o reverso deste *contig* é calculado e verifica-se se esta sequência da fita complementar pode ser alongada. Isto é repetido até que todos os *contigs* possíveis sejam criados. Sempre que um prefixo é analisado, é feita a busca deste em outros *reads*, tendo como

objetivo buscar ambiguidades (inclusive o mesmo prefixo em ambas as extremidades do fragmento).

O peso dos parâmetros na eficiência do montador é muito significativo. Se um filtro muito restritivo é aplicado, muitos *reads* são eliminados e, consequentemente, *contigs* mais curtos são obtidos. Se os filtros são muito permissivos, um número grande de *reads* errôneos são incluídos na fase de construção de *contigs*, causando também interrupções nos *contigs* sempre que um *read* fraco se faz presente. Por essa razão, há três execuções da primeira parte do algoritmo, respectivamente com parâmetros restritivos, médios e permissivos. Assim, os *contigs* obtidos em cada um são utilizados na fase final de fusão das sequências. *Contigs* são unidos caso tenham uma sobreposição mínima de pelo menos o comprimento de um *read*.

2.4.1.4 VCAKE

Baseado na ideia da ferramenta SSAKE, Jeck et al. [2007] propuseram o montador VCAKE. Segundo os autores, a principal diferença entre o VCAKE e o SSAKE é que o VCAKE busca todos os *reads* que se sobreponham com um *contig* sendo formado. Além disso, a sobreposição pode ter algumas diferenças, dentro do limite definido.

Baseado nesses possíveis *reads* candidatos à extensão de um *contig*, cada base nova é eleita entre as bases presentes na mesma posição em cada candidato. A base que tiver uma representatividade mínima definida por parâmetro do sistema é a escolhida. No entanto, se a segunda base mais presente exceder um nível de ocorrência também definido por parâmetro, esta divergência não é considerada como erro de sequenciamento e a extensão é encerrada, já que se trata de uma sequência duplicada em outra região. A extensão pode ainda ser finalizada quando o usuário tiver configurado uma quantidade máxima de *reads* em um *contig* ou quando nenhuma outra sobreposição existir. Após o fim da extensão, o complemento do *contig* recém-formado é criado e a extensão de *contigs* é executada na extremidade oposta.

Por fim, o *contig* finalizado é salvo em arquivo e uma nova semente é escolhida para iniciar outro *contig*.

2.4.2 Ferramentas que Implementam Grafos de Sobreposição

2.4.2.1 Mira

O montador Mira [Chevreux et al., 1999; Chevreux, 2005] trabalha tanto com *reads* longos quanto com aqueles mais curtos, segundo Miller et al. [2010]. Esta tecnologia baseia-se no grafo de sobreposição e manipula as seguintes informações durante a fase de preprocessamento de dados, segundo Chevreux et al. [1999]:

- Traços iniciais dos dados, correspondentes a marcas do gel utilizado no sequenciamento (quando for o caso);
- A sequência genômica;
- Os valores de qualidade de cada base da sequência de DNA (índice de exatidão daquela base obtida no sequenciamento em relação ao valor original na molécula);
- Partes do DNA em cada *read* marcados como região de alta confiança;
- Características gerais, como nome da espécie analisada, etc;
- Informações especiais, como repetições-padrão conhecidas ou trechos de sequência que representam o vetor utilizado durante a cópia do DNA.

Durante uma segunda fase chamada "escaneamento de *reads*", eles são comparados utilizando-se um algoritmo de comparação baseado no algoritmo de busca em textos *Shift-AND* que, segundo Chevreux et al. [1999], é bem mais eficiente do que as comparações tradicionais. Neste algoritmo, indicações de possíveis sobreposições são retornadas, junto com a direção em que ocorrem.

Na fase de inspeção de sobreposições, checagens da combinação de *reads* são efetuadas, determinando valores de qualidade para cada par de sobreposição. Todo par dentro dos critérios de qualidade (tamanho e qualidade da sobreposição) são considerados pares válidos. Alternativas de sobreposição, além das orientações dos fragmentos, são armazenadas para uso nas futuras fases do algoritmo. Todos os alinhamentos formam um ou mais grafos com pesos que determinam as variações possíveis de alinhamento de *reads*. Nesse conceito, cada arco do grafo é uma sobreposição entre *reads*, com o peso sendo relacionado com os pesos de qualidade de sobreposição.

Partindo de um nó com várias sobreposições de alta pontuação, o Mira atua de forma iterativa, sempre pegando o consenso já construído e o próximo *read* e avaliando o peso de sua conexão. Esse par passa a ser o consenso oficial e o próximo *read* é avaliado para união com o consenso recém-formado. Se esta área for um trecho de grande repetição no genoma, a estratégia poderia ser falha. Nesse contexto, uma análise em profundidade de 4 ou 5 nós em sequência (para cada vizinho) é realizada, buscando-se o vizinho que produza o melhor caminho parcial. O melhor é considerado e passa a ser candidato a fazer parte do *contig*. Se este read for muito divergente dos demais membros do *contig* (qualidade das bases, etc), outras alternativas são estudadas; caso contrário, este passa a fazer parte do *contig*. Quando não há mais possibilidade de expandir o consenso, um novo nó é escolhido e um novo *contig* é iniciado – até que todos os fragmentos tenham sido utilizados.

Depois da formação de grupos de *reads* para formação de *contigs*, os *reads* são avaliados e os trechos antes omitidos deles (partes do vetor usado na cópia do DNA, sinais do gel de sequenciamento, etc) são removidos e as adaptações necessárias são promovidas para que se obtenha os *contigs*. Com a remoção das áreas consideradas de baixa confiança, possíveis conexões entre contigs podem ser levantadas e, por fim, estes são conectados, reagrupados ou separados de acordo com correções específicas, como eliminação de erros de montagem, correção de *reads* individuais, entre outras opções.

Um trabalho posterior, com correções e ampliações, foi apresentado em tese doutoral de Chevreux [2005]. A Figura 2.7 apresenta as fases da montagem com a ferramenta Mira. Segundo Nagarajan & Pop [2013], este montador tem como característica interessante permitir uma montagem híbrida, com dados de mais de uma tecnologia de sequenciamento.

2.4.2.2 Celera

O montador proposto por Myers et al. [2000] foi, segundo Nagarajan & Pop [2013], o responsável pela popularização do método de montagem baseado na ideia de *overlap-layout-consensus* (grafos de sobreposição).

Após o sequenciamento (com informação da qualidade de cada base sequenciada), o montador Celera atua cortando os *reads* obtidos para alcançar taxas de qualidade média de 98%. Nesse contexto, sinalizadores e demais partes de baixa qualidade ou que não pertençam ao genoma original são removidas. Em seguida, estas partes resultantes são analisadas para a remoção de trechos de material genético de organismos utilizados para cópia do DNA durante o sequenciamento. Uma

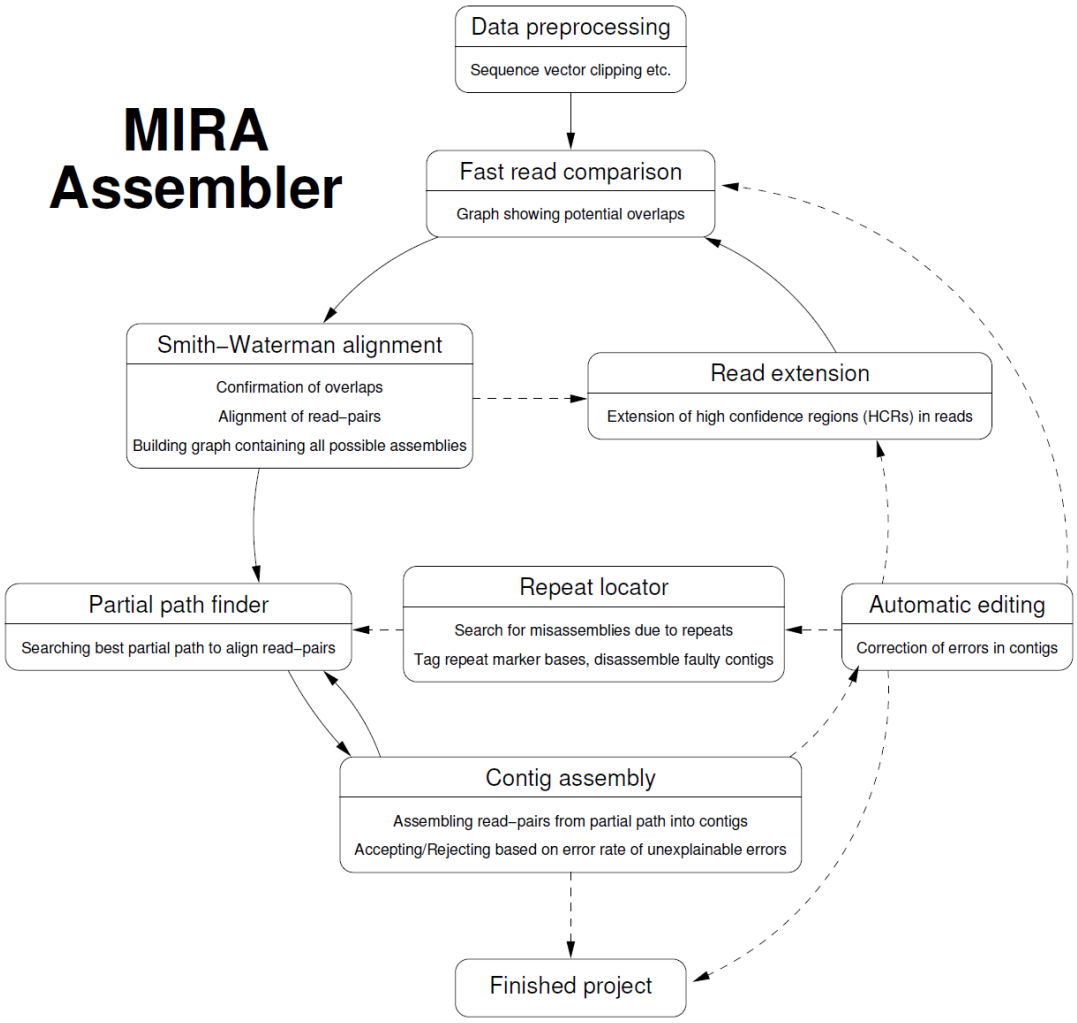

Figura 2.7. Fases do Montador Mira. Setas contínuas são passos obrigatórios e trechos tracejados representam operações opcionais. Imagem retirada do trabalho de Chevreux [2005].

etapa de eliminação de DNA ribossômico e outros tipos paralelos que não o DNA principal também é realizada, já que eles não representam o foco da montagem.

Após essa "purificação" dos *reads*, o cálculo de sobreposições é efetuado, considerando como sobreposição correta aquelas que apresentem uma margem de diferença dentro do percentual máximo aceitável e que tenham áreas comuns de tamanho mínimo. Sobreposições muito extensas são consideradas áreas repetidas (*repeats*) no genoma.

Baseado nas sobreposições detectadas, *unitigs* são calculados. Conforme definido pelo autor [Myers et al., 2000], *unitigs* são trechos cujas sobreposições dos *reads* são incontestáveis. Em casos de ocorrência de *unitigs* que sejam trechos de *repeat*, o algoritmo busca utilizar as áreas de repetição quantas vezes forem necessárias para

ligar os *unitigs* únicos, em um esforço para estendê-los.

Com a ajuda de *mate pairs* (pares de *reads* com uma longa distância entre eles e cuja distância e localização relativa entre os dois é conhecida), os *scaffolds* são construídos, ou seja, grupos de *unitigs* com orientação e localização relativa surgem. Após a fase de construção destes scaffolds e o alinhamento destes também baseando-se nas informações de vários *mate pairs*, o algoritmo processa o consenso possível desta estrutura.

Segundo Nagarajan & Pop [2013], o Celera foi o primeiro montador entre os pioneiros baseados em *overlap-layout-consensus* que montou o genoma completo de um ser multicelular (a mosca *Drosophila melanogaster* [Myers et al., 2000]).

2.4.2.3 Minimus

Com a proposta de ser uma ferramenta mais simples, leve e menos engessada do que os gigantes da montagem, geralmente desenvolvidos para larga escala, surge a ferramenta Minimus [Sommer et al., 2007]. Segundo o autor, esta ferramenta facilita a incorporação de novas técnicas de correção de intervalos não sequenciados (frequentemente causados por *repeats* não resolvidos) e outras técnicas de correção e aprimoramento que surgem, sendo agrupadas como novos módulos. É voltado para pequenos conjuntos de dados e é frequentemente utilizado para combinar resultados de mais de um montador [Nagarajan & Pop, 2013].

Sendo membro de um pacote de ferramentas chamado AMOS, o montador Minimus é composto de 3 ferramentas deste pacote. Baseando-se no conceito de *overlap-layout-consensus*, o montador é composto pelos sistemas: *hash-overlap*, que calcula a sobreposição de *reads* com estruturas que buscam minimizar o consumo de recursos de memória e processamento; *tigger*, que tenta buscar grupos de *reads* relacionados, que podem ser unificados posteriormente; e *make-consensus*, que progressivamente calcula os múltiplos alinhamentos tentando refinar o esboço obtido pelo *tigger*, criando o consenso da montagem. Ao contrário de outros montadores, Minimus utiliza as informações de qualidade de bases apenas no último passo da montagem.

Minimus executa uma série de passos durante o processo de montagem. No primeiro passo, as entradas de dados são carregadas para o banco do pacote AMOS. O *hash-overlap* calcula, em seguida, as sobreposições/alinhamentos entre todos os fragmentos fornecidos. Baseado nas informações de sobreposição, o *tigger* constrói o grafo correspondente. Após a construção do grafo, o mesmo componente ainda promove a simplificação deste grafo, seguindo três passos:

- Remoção de *reads* cuja sequência esteja contida em outros *reads* (sequências redundantes);
- Redução transitiva: se a sobreposição entre vértices a e c no grafo puder ser deduzida da sobreposição entre a e b e entre b e c, a aresta a–c é removida;
- Caminhos simples, ou seja, grupos de vértices que não tenham mais de um vizinho, são convertidos em um único vértice, que é chamado *unitig*;

Após a criação dos *unitigs*, o montador tenta aferir a sequência final percorrendo todos os vértices. Para determinar a ordem dos vértices, é considerada a informação de sobreposições de fragmentos.

2.4.3 Exemplo de Montador que se baseia em Grafos de *String*

2.4.3.1 SGA

SGA [Simpson & Durbin, 2012] é baseado no grafo de *strings*, que é uma categoria derivada do conceito de grafos de sobreposição. Como retratado por Nagarajan & Pop [2013], a abordagem baseada em sobreposição sempre encontrou maior limitação por conta do seu custo computacional, mas este montador implementou estruturas de dados mais eficientes para indexação de *strings*. A Figura 2.8 apresenta as atividades executadas.

Através do uso de indexação, as estruturas de cada sequência são representadas de forma comprimida. Uma sequência grande é representada por conjuntos de sequências menores que se repetem, em um esquema que evita armazenar a sequência de forma integral.

A primeira correção de erros em *reads* é feita baseando-se na correção de k-mers, como já proposto em outros trabalhos [Pevzner et al., 2001; Li et al., 2010] (seções 2.4.4.5 e 2.4.4.1). A diferença aqui é que os k-mers não são armazenados, mas calculados através da estrutura de indexação. A segunda correção, por sua vez, procura permitir sobreposições aproximadas como válidas. Como a primeira correção tem custo mais baixo para execução, ela é executava por padrão no montador.

No montador, antes da construção de fato do grafo, há a remoção de *reads* que contenham k-mers que não existam em nenhum outro *read*, já que é esperado que seja causado por erros. *Reads* com a mesma sequência ou cuja sequência de um seja idêntica ao complemento reverso de outro ou *reads* cuja sequência esteja contida em

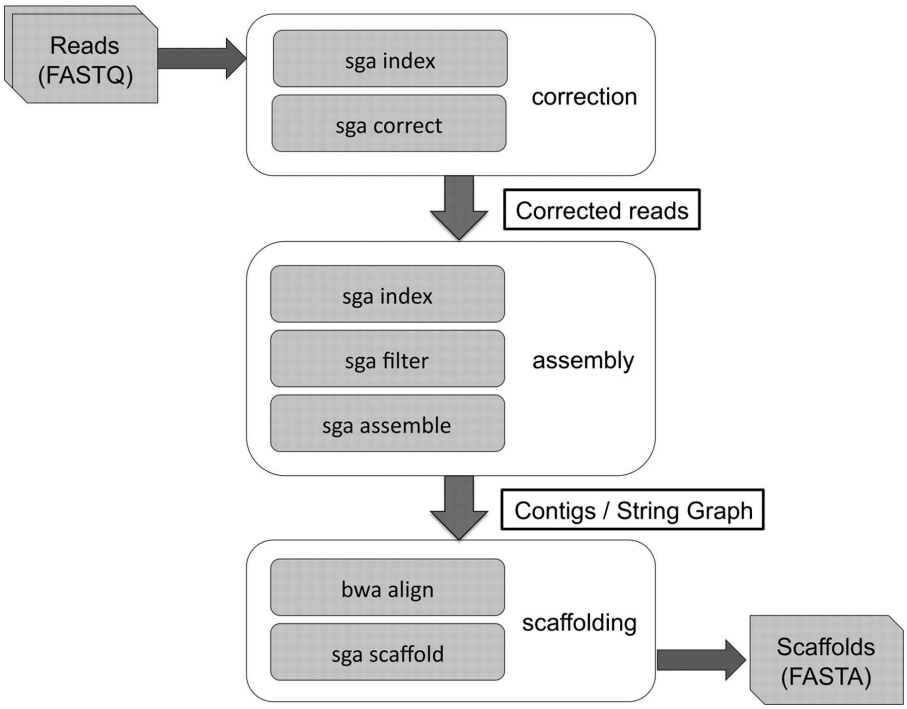

Figura 2.8. Representação em alto nível do SGA. Cada caixa representa uma das três fases principais, com suas respectivas subtarefas. Retirado do trabalho de Simpson & Durbin [2012]. Disponível em: <http://genome.cshlp.org/content/22/3/549.full> Acesso em: 23 jan. 2014.

outros *reads* também são eliminados. Sendo assim, somente *reads* únicos e legítimos são aproveitados para a construção do grafo, evitando redundância.

Antes de criar o grafo, há ainda a pesquisa no índice de sequências por casos de sobreposição entre os fragmentos que não possuam redundância (mais de uma opção de caminhos). Estes trechos são transformados em vértices únicos, já que podem ser deduzidos sem risco ou dúvida. Após esse passo, há a reconstrução do índice para as sequências que sofreram fusão. A nova versão do índice é então utilizada para construir o grafo de *strings*.

No grafo montado, é feita a remoção de "galhos" do grafo que sejam considerados errôneos, como já conhecido de outros trabalhos [Chaisson & Pevzner, 2008; Simpson et al., 2009; Li et al., 2010; Zerbino & Birney, 2008]. Uma remoção de bolhas também é executada. Caso os caminhos que componham a bolham tenham similaridade mínima de 95%, um caminho é escolhido e os demais são salvos em um arquivo FASTA para posterior quaisquer análises futuras.

Por fim, uma etapa de construção de *scaffolds* é executada de forma parecida com outras ferramentas já citadas, com base em *paired reads* e *mate-pairs*. Todo ciclo encontrado é isolado, então cada componente conexa é definida e seus vértices

de início e fim são encontrados (grau zero de saída ou entrada). Através desses vértices, caminhos possíveis são elaborados e aquele que cubra melhor os elementos da componente é escolhido como favorito. Intervalos sem cobertura que não puderem ser resolvidos são substituídos pela letra N, que representa uma base desconhecida.

2.4.4 Montadores que Utilizam Grafos *de Bruijn*

2.4.4.1 Euler

O montador de DNA Euler [Pevzner et al., 2001], ao contrário dos montadores anteriores a ele, abandona a filosofia do *overlap-layout-consensus* e parte para uma abordagem baseada no caminho Euleriano (sobre caminhos Eulerianos, ver seção 2.3.5, na página 13). Segundo o autor da ferramenta, ao contrário do montador Celera, que "disfarça" os *repeats*, o Euler toma essas áreas como parte do processo de montagem. Nagarajan & Pop [2013] defendem que o montador Euler foi o responsável pela popularização dos grafos *de Bruijn* e de estratégias baseadas em caminho Euleriano nos montadores.

Pevzner et al. [2001] defendem que alterar a filosofia do algoritmo de uma metodologia baseada no caminho Hamiltoniano (dos grafos de sobreposição) para o cálculo de caminho Euleriano no grafo *de Bruijn* é transformar um problema que não tem abordagem ótima conhecida em um problema que possui soluções conhecidas, em que se obtém resultados em tempo satisfatório mesmo em problemas mais complexos.

No Euler, o grafo *de Bruijn*, como já conhecido de sua definição, incorpora áreas de repetição em uma mesma sequência de vértices, sem duplicação de sequências. Porém, as arestas não são unificadas. Logo, se há mais de um trecho do DNA que cria uma ligação entre vértices a e b, há um número igual de arestas conectando ambos. Dessa forma, a filosofia de passeio uma única vez por todas as arestas do grafo é mantida, com cobertura de todos os vértices (vértices de área de *repeat* são visitados mais de uma vez).

Para resolver o problema de montagem, respeitando as informações fornecidas pelos *reads*, Pevzner propôs uma abordagem chamada *Eulerian Superpath* (super-caminho Euleriano). Seguindo esta abordagem, cada *read* se transforma em um "caminho de *read*" dentro do grafo e o objetivo é encontrar um super-caminho Euleriano que respeite os caminhos de cada *read*.

De acordo com a quantidade de *reads* que possuam um determinado vértice, este é considerado fraco ou forte no grafo, sendo esse valor um parâmetro do montador. Baseando-se nessa classificação, o autor define que uma aproximação da

sequência implícita no grafo pode ser obtida com o conjunto de tuplas (sequências de caracteres) de tamanho k fortes do grafo.

Após a fase anterior e alguns outros passos de correção de erros preliminares, o montador executa a fase de obtenção dos super-caminhos Eulerianos. No momento da montagem do super-caminho, os caminhos de *reads* são levados em conta para se obter a ordem da passagem de regiões de repetição para regiões que saem de repetições no grafo. A Figura 2.9 demonstra esse processo. Entretanto, nem sempre essas informações são suficientes, criando áreas de solução não conhecida até agora.

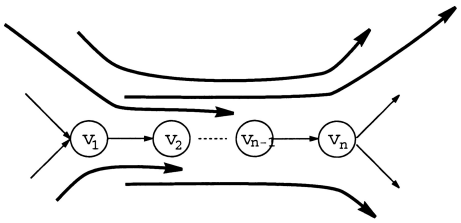

Figura 2.9. Indução de super-caminho pela trajetória de *reads*. As setas grossas representam *reads*. A cadeia de círculos representa uma área de *repeat*. Pode-se perceber que o *read* do topo mapeia o caminho de entrada e saída pela área de repetição. Sem o mesmo, este trecho não teria resolução. Imagem extraída do trabalho de Pevzner et al. [2001]. Disponível em: <http://www.pnas.org/content/98/17/9748.full> Acesso em: 23 jan. 2014.

Para contornar isto, transformações buscando simplificação do grafo são efetuadas, de modo a transformar trechos do grafo que não possuem divergência em blocos únicos, desde que as versões anterior e modificada do grafo possam retornar a mesma sequência. Após essa "aglutinação" de diversos caminhos de *reads*, o cenário volta ao formato comum do problema, ou seja, o caminho Euleriano tradicional. A Figura 2.10 apresenta um exemplo de *ciclo* Euleriano, ou seja, um caminho Euleriano que começa e termina na mesma aresta.

2.4.4.2 Velvet

Este montador [Zerbino & Birney, 2008] é, na verdade, um grupo de algoritmos para manipulação de grafos *de Bruijn*. Porém, diferentemente do grafo *de Bruijn* padrão, esta abordagem utiliza o grafo baseado nos *reads*, agrupando os k-mers de acordo com regras específicas.

Nesta proposta, cada nó do grafo é composto por um conjunto de k-mers que se sobrepõem em $k-1$ bases. A sequência constituída pelas últimas bases de cada um destes k-mers do nó forma a sequência do nó, ou $s(N)$. Cada nó N é ´´colado" a um nó gêmeo $\tilde{N}$, que contém todos os k-mers que representam a sequência reversa

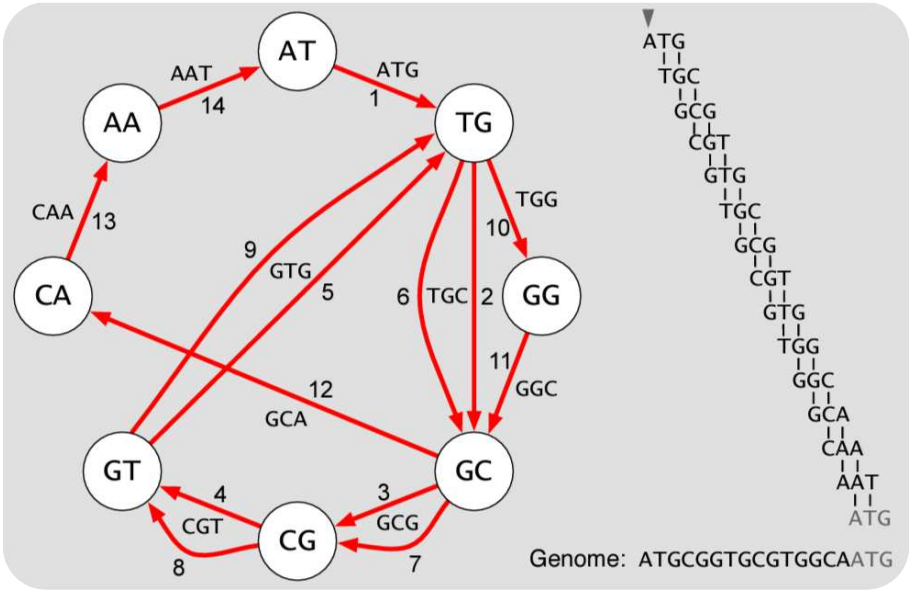

Figura 2.10. Ciclo Euleriano em um grafo. As setas vermelhas representam as arestas do ciclo Euleriano. Começando da aresta 1 (ATG), obtém-se a sequência de arestas descritas à direita na imagem. Esta imagem pertence ao material suplementar do trabalho de Compeau et al. [2011].

daqueles presentes no nó N. Cada par de um nó e seu nó gêmeo é chamado de bloco. Cada nó pode ser conectado com outro nó a partir de um arco. Obrigatoriamente, se há um arco entre nós a e b, haverá também uma conexão no sentido inverso entre seus nós gêmeos, ou seja, de $\tilde{b}$ para $\tilde{a}$. As modificações em qualquer arco afetam seu arco correspondente entre os nós gêmeos. A Figura 2.11 mostra um exemplo desta abordagem alternativa do grafo k-mer.

Com estruturas de mapeamento, este conjunto de algoritmos cria diversas informações de apoio durante sua execução. Todos os *reads* são "partidos" em k-mers de um tamanho fixo k. É controlado ainda, para cada k-mer, qual o primeiro *read* que o possui e em qual posição dentro deste. Todo k-mer é armazenado junto com seu complemento reverso. Um segundo banco de dados é criado com a informação, para cada *read*, de quais de seus k-mers que o compõem são cobertos por outro *read*. A cada início ou fim destes *reads* que o cobrem, há o fim de um nó do grafo e início do próximo.

Após essa fase de construção do grafo, há uma fase de simplificação. Sempre que um nó a possui apenas um arco de saída e seu vizinho b possua apenas um arco de entrada (o arco que conecta a a b), estes podem ser transformados em um único nó. O mesmo é feito com seus nós gêmeos e, consequentemente, os dois blocos anteriores viram apenas um bloco. Isso é feito sucessivamente até que não haja mais

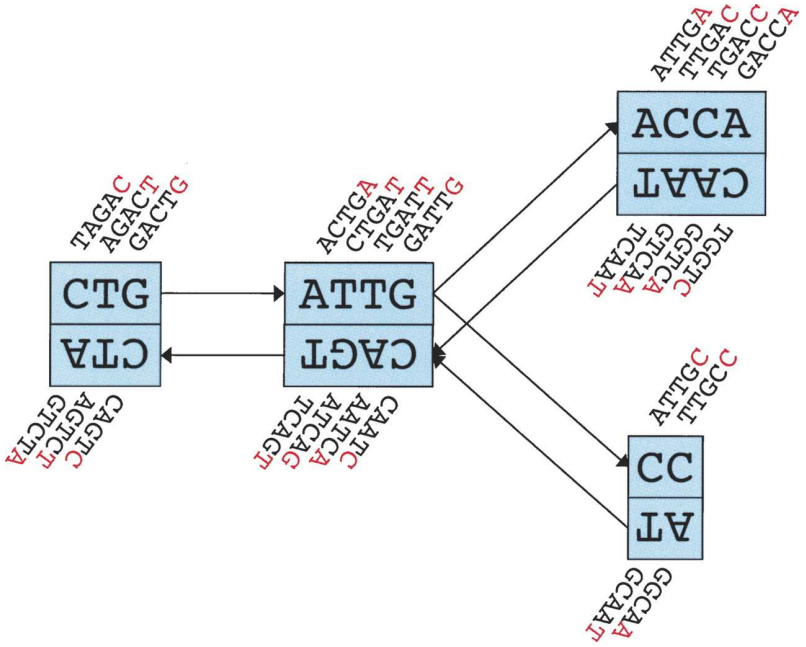

Figura 2.11. Exemplo da estrutura de grafo no montador Velvet. As letras vermelhas representam as últimas bases de cada k-mer, que representam a sequência dos nós. Neste exemplo, $k = 5$. Imagem retirada do trabalho de Zerbino & Birney [2008]. Disponível em: <http://genome.cshlp.org/content/18/5/821.full> Acesso em: 10 dez. 2013.

simplificação possível. Na Figura 2.11, por exemplo, os nós `CTG` e `ATTG` (com seus respectivos nós clones) poderiam ser simplificados.

Após a fase de simplificação, o Velvet executa uma remoção de erros, seguindo três características de topologias: *tips* (equivalentes às erosões apresentadas na subseção 2.3.5), bolhas e conexões errôneas. Após essa fase de remoção dos erros, uma simplificação é novamente realizada no grafo.

Na fase de construção dos *contigs*, os blocos longos de um tamanho mínimo são alinhados e, de acordo com informações extra de distância aproximada entre estes trechos fornecidas pelos sequenciadores, ele tenta caminhar entre estes blocos longos através dos blocos menores que os conectam. Se conseguem encontrar um caminho que respeite esta distância, os blocos são fundidos em um bloco único. Este algoritmo é chamado de "migalha de pão", pelo processo de marcação de blocos durante o cálculo do caminho entre dois blocos. Como abordado pelo autor [Zerbino & Birney, 2008], a eficiência deste montador depende diretamente do tamanho dos *reads*, do genoma, da qualidade da sequência e da cobertura. Além disso, a solução se mostra, segundo o mesmo autor, muito sensível ao tamanho de k.

2.4.4.3 ALLPATHS

Proposto por Butler et al. [2008], o ALLPATHS trabalha com montagem de reads *unpaired* ou *paired* (seção 2.3.1), sendo voltado para montagem de *reads* mais curtos (25–50 bases). Este montador baseia-se nos grafos *de Bruijn*.

Antes de tudo, há a fase de correção de erros em *reads*. A correção de erros se baseia nas técnicas empregadas por Pevzner et al. [2001], levando em conta a quantidade de vezes que um k-mer aparece nos *reads*, além de outros fatores. Além disso, ALLPATHS testa diversos tamanhos de K para cada *read* e promove alterações em bases que tenham pouca cobertura, tentando substituir por opções de bases com alta qualidade de sequenciamento encontradas em outros *reads* similares. Se uma alteração melhora a representatividade de todos os k-mers dentro do read, este é mantido.

Após a fase de correção, todos os k-mers são calculados nos *reads*. Cada k-mer é relacionado a um número que o representa, facilitando a representação de qualquer *read* através de uma sequência de números. Esta representação numérica torna desnecessária a busca por sobreposições, já que elas estão implicitamente calculadas. Além disso, caminhando pelas sequências de k-mers torna-se possível o cálculo dos *unipaths*. *Unipaths* são caminhos obtidos percorrendo o grafo até um ponto em que haja bifurcação. A partir daí, uma nova estrutura de grafo está formada, chamada "grafo de *unipaths*".

Nos “grafos de *unipaths*”, cada *unipath* é uma aresta (em sua representação gráfica). Aresta, neste caso, é uma linha que representa todos os vértices de um mesmo *unipath*. Os pontos (vértices) deste grafo são as áreas de conexão entre *unipaths*. A Figura 2.12 apresenta esta estrutura. *Unipaths* são vizinhos quando o último k-mer do primeiro *unipath* é adjacente ao primeiro k-mer do segundo *unipath*. Grafos de *unipaths* e os grafos retornados no fim da montagem são classificados como “grafos de sequências”. A diferença entre este e o anterior é que o grafo de sequências possui as sequências obtidas dos *unipaths*, já como um consenso local. Cada sequência é armazenada também com o número de vezes que ocorre no genoma (ou no que se aferiu dos dados de sequenciamento). Na montagem com *reads* do tipo *unpaired*, este é o ponto máximo onde o montador consegue chegar. O grafo de sequências é então retornado em arquivo.

A montagem de fragmentos usando *reads* pareados possibilita que se avance mais nos resultados, mas também torna o processo ainda mais complexo. Além disso, segundo Butler et al. [2008], nem sempre os resultados nesta modalidade são melhores do que a versão *unpaired*.

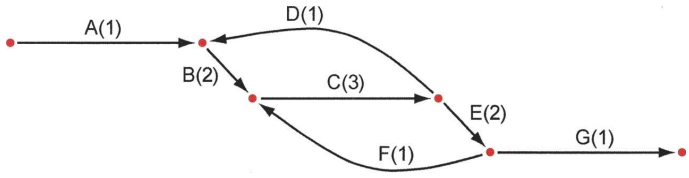

Figura 2.12. Representação do grafo de *unipaths* da espécie *C. jejuni* com k=6000. Os *unipaths* estão nomeados por letras, com o número de cópias entre parênteses. Figura extraída do trabalho de Butler et al. [2008]. Disponível em: <http://genome.cshlp.org/content/18/5/810.long> Acesso em: 23 jan. 2014.

Na montagem com *paired reads*, ALLPATHS se baseia em dois objetivos durante a montagem. Primeiro, baseando-se na distância entre dois *reads*, há o cálculo dos diversos caminhos possíveis entre eles. Em segundo lugar, ele se baseia no conceito de localização, tentando mensurar a localização dos elementos na montagem.

Para se obter caminhos entre dois *reads* com distâncias conhecidas, diversos *reads* com uma sobreposição de tamanho mínimo k são alinhados. Em situações ideais, consegue-se conectar ambos os *reads* externos com o alinhamento. Esses esquemas regionais formam as montagens locais. Com a combinação das montagens locais, busca-se a montagem geral. Como diversas opções de caminhos entre dois *reads* podem ser obtidos, pode ser inviável decidir qual dos caminhos escolher para a montagem local.

A ferramenta trabalha ainda com a combinação dos *unipaths*, da localização de *reads* pareados, com a eleição de "sementes" locais de montagem e com a edição posterior da montagem obtida.

Uma versão mais recente deste montador, chamada ALLPATHS-LG, foi apresentada por Gnerre et al. [2011]. Esta versão foi desenvolvida para trabalhar com paralelismo e memórias compartilhadas, além de algumas melhorias em relação à versão inicial.

2.4.4.4 AbySS

Desenvolvido por Simpson et al. [2009], este montador foi criado com foco em sistemas computacionais distribuídos, como as redes de computadores geograficamente distribuídos (*computational grids*)[Nagarajan & Pop, 2013]. Como mencionado por Miller et al. [2010], o uso dessa estrutura possibilitou contornar a limitação de memória para grandes e complexos genomas, possibilitando inclusive a montagem de 3,5 bilhões de *reads* de genoma de um ser humano obtidos do sequenciador Solexa. A estrutura utilizada pelo algoritmo para processar a montagem é baseada nos gra-

fos *de Bruijn* e é voltado, principalmente, para montagem de *reads* curtos gerados por montadores de nova geração.

Um dos detalhes mais importantes desta ferramenta é a adoção de um grafo k-mer distribuído em diversos nós (computadores) da rede. Para que isso funcione, porém, todo k-mer deve ter sua localização facilmente obtida através de sua sequência e sua relação de vizinhança com demais vértices deve ser salva de maneira independente de sua localização física real. Sequências que contenham bases indefinidas (N ou .) são descartadas, restando apenas *reads* com as bases-padrão.

Para definir a localização de um k-mer, uma função *hash* é aplicada na sequência dos *reads*, de forma a dividir de forma equilibrada o número de vértices em cada nó da rede. Para a relação de vizinhança, por sua vez, uma cadeia de 8 bits de informação é criada para cada vértice. Tendo o DNA apenas 4 bases (A, C, G e T), cada vértice pode ter até quatro vizinhos de entrada e quatro de saída. O bit 1, nesse contexto, representa a ligação ou não de um vértice com um vizinho que possua prefixo ou sufixo de uma das quatro bases.

Para a etapa de montagem, há dois passos [Simpson et al., 2009]. No primeiro, sem o uso de *paired reads*, os *contigs* são estendidos até que se encontre ambiguidades ou que haja um fim da sequência por falta de cobertura. Na segunda etapa, *paired reads* são levados em conta para remoção de ambiguidades e fusão de *contigs*.

A correção de erros no grafo montado é efetuada de duas formas. Os "galhos" formados no grafo são considerados errôneos caso seus tamanhos sejam inferiores a um valor de parâmetro. Para a remoção de bolhas, todo ponto de divergência de caminhos no grafo é analisado. Partindo-se de cada um destes pontos, espera-se que os caminhos se encontrem novamente após n vértices ($k \leq n \leq 2k$). Se estes caminhos se juntam, o caminho com menor cobertura de *reads* é removido do grafo, mas ambos são salvos em um arquivo de log.

Após essa correção de erros, a criação de *contigs* é iniciada. Para tanto, todas as arestas ambíguas remanescentes são removidas do grafo, sobrando apenas aquelas não ambíguas. Assim, todos os *contigs* podem ser obtidos pela leitura de cada grupo de vértices conectados.

Na segunda fase da montagem, os *reads* pareados (caso existam), são utilizados para fusão de *contigs*. Tendo como indicador cada um desses *reads*, o algoritmo tenta alinhar *contigs* através deles, para que se busque um caminho que passe por cada *contig* ao fim da montagem. Para que um *contig* seja considerado corretamente alinhado, porém, um número p de pares de *reads* deve assegurar sua localização.

2.4.4.5 SOAPdenovo

Proposto por Li et al. [2010], este montador tem maior apelo em sistemas de memória compartilhada, como servidores com múltiplos núcleos de processamento. Segundo Nagarajan & Pop [2013], esta ferramenta tornou possível a primeira montagem *de novo* do genoma de um mamífero através apenas de *reads* curtos. Assim como boa parte dos montadores mais recentes, este também se baseia nos grafos *de Bruijn*. É voltado para *reads* obtidos do sequenciador Solexa (Illumina). A Figura 2.13 apresenta uma visão geral dos passos para montagem através do SOAPdenovo.

A remoção de erros dos k-mers é feita através de uma tabela *hash* com todos os fragmentos de tamanho k possíveis. Então, a frequência de todos eles é calculada pela leitura dos dados. Começando por cada *read*, a área de maior cobertura é lida e então é feita uma navegação para os extremos do *read*, buscando aqueles k-mers de menor cobertura. Encontrado um bloco nessas condições, este é testado em todas as variações possíveis na sua extremidade que difere do restante do *read* (3, sendo 1 para cada base nucleotídica). Caso uma dessas versões possua cobertura satisfatória, a mudança é feita.

Além dessa correção, há métodos para correção do grafo *de Bruijn* montado. Todo "galho" com tamanho de até $2k$ é removido. Para bolhas, um algoritmo percorre ambas as "bordas" da bolha e une os caminhos em um só (dando preferência pelo mais extenso deles), mas somente quando as bolhas têm apenas uma base de diferença entre seus dois caminhos ou até 4 bases, caso a identidade seja de mais de 90%. Pequenos *repeats* são resolvidos em casos onde há N arestas de entrada e de saída no nó e *reads* traçam caminhos passando por arestas de entrada e saída sem conflito. Nesse caso, caminhos alternativos são criados para cada par de arestas saída/entrada. Os *contigs* são então calculados a partir do grafo restante.

Após o cálculo dos *contigs*, o montador utiliza um número mínimo de pares de *reads* para confirmar os alinhamentos entre dois *contigs*. Estes alinhamentos formam então os *scaffolds*, ou seja, conjuntos de *contigs*. Por fim, *mate-pairs* são utilizados para formar outros *scaffolds*. Os *gaps* (áreas faltantes) existentes nos *scaffolds* são resolvidos através de pares de *reads* que tenham um *read* alinhado com um *contig* e o outro com uma área vazia. Estes trechos são geralmente formados por áreas de repetição.

Uma nova versão deste montador foi proposta por Luo et al. [2012], tendo 5 melhorias em relação à primeira versão: (1) melhoria da etapa de correção de erros; (2) redução do consumo de memória para a construção do grafo; (3) resolução de *repeats* mais longos; (4) aumento do tamanho de montagens na fase de construção

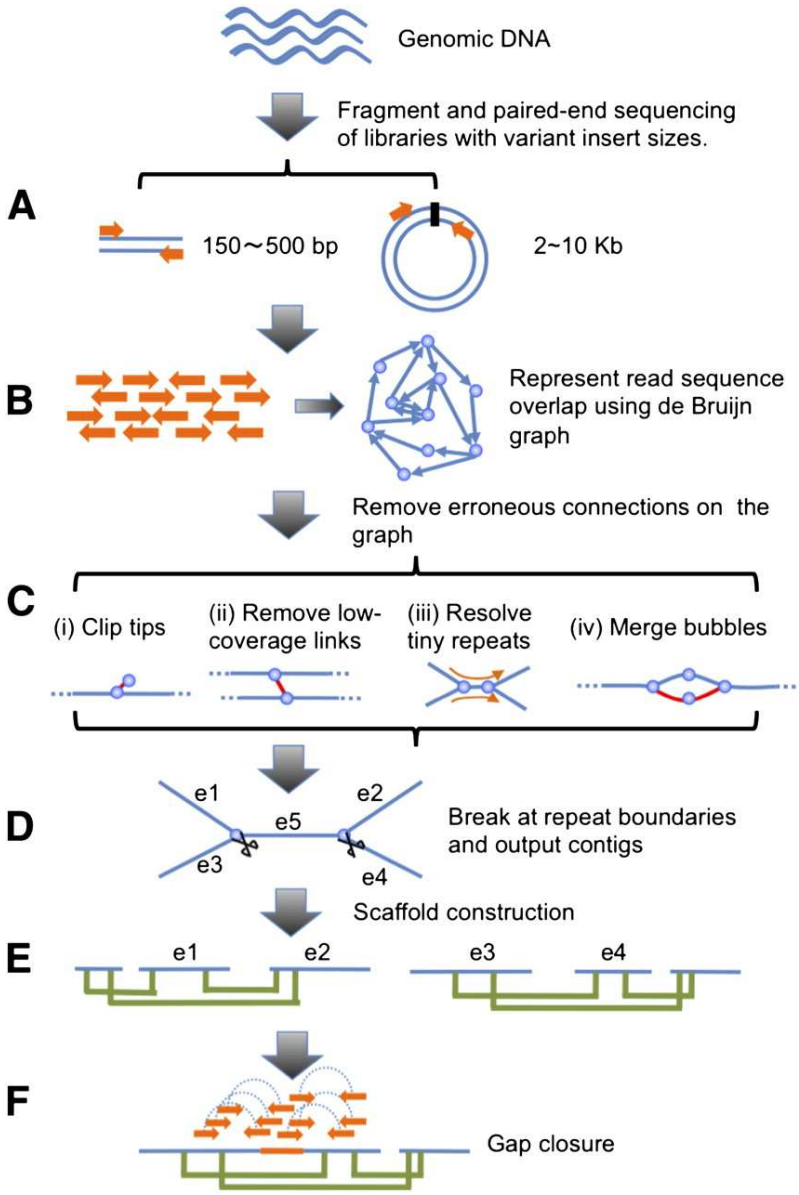

Figura 2.13. Montagem de fragmentos de DNA com SOAPdenovo. Em (A), os fragmentos são sequenciados com informação de *paired reads*. Em (B), os *reads* sequenciados são processados na forma de grafo k-mer. A remoção de erros ocorre no passo (C), com eliminação de erosões e conexões com baixa cobertura, além da solução de *repeats* menores e fusão de bolhas. Na fase (D), os extremos de *repeats* são desconectados, formando *contigs*. Nos passos (E) e (F), é feita a construção de *scaffolds* e a cobertura de lacunas, respectivamente. Figura extraída do trabalho de Li et al. [2010]. Disponível em: <http://genome.cshlp.org/content/20/2/265.full> Acesso em: 23 jan. 2014.

de *scaffolds*; e (5) melhoria na resolução de *gaps*.

2.4.4.6 Ray

O montador Ray [Boisvert et al., 2010] é baseado no grafo *de Bruijn* e utiliza o esquema de sementes para extensão de *contigs*.

Através dos dados de entrada, esta ferramenta calcula uma distribuição dos k-mers de acordo com sua cobertura. Sendo assim, todo k-mer tem confiança c, sendo c o seu valor de cobertura. Através dos cálculos da distribuição desta cobertura, o pico inferior nesta distribuição, formado por vértices menos cobertos, possibilita determinar o parâmetro de corte de k-mers de baixa qualidade.

O grafo *de Bruijn* neste montador é formado através dos *reads* obtidos (e seus complementos), um tamanho k definido para os vértices e o nível de corte c. Todo *read* tem seu primeiro k-mer avaliado. Caso a cobertura deste k-mer tenha cobertura maior do que 255, este *read* é considerado membro de uma área de repetição e, como consequência, é excluído do conjunto de *reads*.

No grafo, o cálculo das sementes, que serão usadas na extensão dos *contigs*, é feito pela determinação de um valor de corte alto baseado na cobertura. Para calculá-lo, soma-se a cobertura média no grafo e o valor mínimo de cobertura onde ainda se encontra mais sequências corretas do que errôneas. O valor de corte é a metade desta soma. As sementes são então obtidas através do cálculo dos caminhos máximos que é possível recuperar do grafo. Estes caminhos são formados de cada grupo de vértices com grau de saída e entrada de no máximo 1.

Depois de definir a semente, cada uma delas é usada como base para expandir *contigs*. Quando houver mais de uma opção de caminho, um cálculo baseado nos *reads* que sobrepõem as opções é feito para tentar decidir a melhor opção. Neste processo, *reads* que possuam sobreposição mais extensa com os *contigs* têm valor mais alto do que aqueles que se sobreponham apenas ao início de um *contig*.

O autor afirma que esta técnica restritiva limita o tamanho dos *contigs* retornados, mas garante uma maior qualidade desses.

2.5 Ponderações da Literatura Levantada

É possível perceber, diante dos montadores analisados, que a estrutura de grafos k-mer é a mais utilizada. A Tabela 2.1 apresenta um resumo das ferramentas levantadas e da tecnologia utilizada em cada uma.

Como pode-se perceber na tabela, os montadores mais novos são os que mais utilizam os grafos k-mer. Neste trabalho, a abordagem proposta também implementa este tipo de grafo. O fato de evitar a comparação par a par de sobreposição,

Tabela 2.1. Relação de Montadores × Tecnologia Implementada

Montador	Publicação	Abordagem
AbySS	2009	Grafo k-mer
ALLPATHS	2008	Grafo k-mer
Celera	2000	Grafo de sobreposição
Euler	2001	Grafo k-mer
Minimus	2007	Grafo de sobreposição
Mira	1999	Grafo de sobreposição
Phrap	1994	Algoritmo guloso
Ray	2010	Grafo k-mer
SGA	2012	Grafo de string
SHARCGS	2007	Algoritmo guloso
SOAPdenovo	2010	Grafo k-mer
SSAKE	2007	Algoritmo guloso
VCAKE	2007	Algoritmo guloso
Velvet	2008	Grafo k-mer

já citado previamente, também representa forte argumento para seu uso neste trabalho. Mais detalhes sobre a proposta do protótipo desenvolvido são apresentados no capítulo 3.

Capítulo 3

Materiais e Métodos

Neste capítulo, apresenta-se toda a base teórica e, ainda, questões práticas importantes para a implementação da ferramenta, além de detalhes dos experimentos que foram realizados. Na seção 3.1, discute-se o trabalho anterior que inspirou esta pesquisa. Na seção 3.2, faz-se uma demonstração da aplicação do conceito proposto nas topologias problemáticas. Na seção 3.3, por sua vez, há uma análise de fatores que afetam essa ideia inicial. Após a análise, a seção 3.4 descreve toda a abordagem do montador proposto nesta dissertação. Os detalhes da codificação da ferramenta e da ambientação dos testes são apresentados nas seções A.3 e 3.5.

3.1 Ponto de Partida

Como pode-se observar nas ferramentas levantadas na Revisão Teórica (seção 2.4), o objetivo principal nos grafos (quando não há ciclos), é a obtenção de um caminho único por todo o grafo, que represente a sequência original do genoma. Devido aos desafios já citados na seção 2.3, esta tarefa se torna impraticável. Sendo assim, a tarefa mais plausível passa a ser a busca por caminhos disjuntos longos que representem os *contigs*, i.e., regiões contínuas da molécula original. De uma forma mais precisa, o que se procura é uma cobertura mínima de caminhos, ou seja, o número mínimo de caminhos disjuntos que consigam abranger todos os vértices do grafo e de forma que tais caminhos comecem e terminem em vértices de grau de entrada e saída iguais a zero, respectivamente. A minimização do número de caminhos representa, por consequência, a obtenção de caminhos mais longos. Este processo de minimização segue o critério de parcimônia, como defendido pela navalha de Occam, onde "a resposta mais simples é a melhor"[Thorburn, 1918], principalmente no que diz respeito a ciclos causados por *repeats*, já que a solução, quando não hou-

ver uso de *paired-end reads*, será a de utilizar os ciclos o menor número possível de vezes para gerar os *contigs*. Além do mais, *contigs* mais longos fazem mais sentido biologicamente falando.

A proposta deste trabalho é baseada no emparelhamento máximo, ou seja, obtenção de um conjunto maior possível de arestas não adjacentes no grafo, em uma versão bipartida do grafo k-mer. Essa ideia segue um trabalho já proposto por Cerqueira & Meidanis [2001] para grafos de sobreposição.

Seja $G = (V, A)$ um grafo direcionado. O *dobro* de G é um grafo bipartido não direcionado $G' = (V1, V2, E)$, onde $V1 = V2 = V$ e E é composto como a seguir. Para cada arco $(u, v) \in A$, tem-se uma aresta $(u, v) \in E$, onde $u \in V1$ e $v \in V2$. Assim, $|A| = |E|$. A Figura 3.1 mostra um exemplo. Baseado nesta definição, pode-se propor o Teorema 1 (prova completa no trabalho de Cerqueira & Meidanis [2001]).

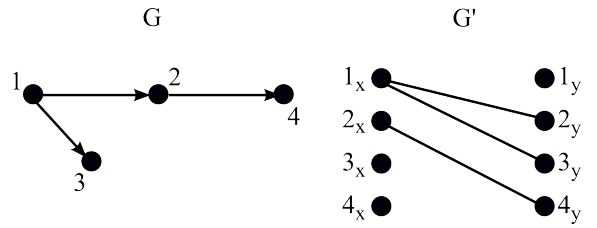

Figura 3.1. Grafo direcionado G e seu dobro G'. Os nós em G' possuem rótulos x e y para distinção das partições do grafo.

Teorema 1 *Seja G um grafo direcionado e G' seu dobro. Todo emparelhamento M em G' corresponde a um conjunto P de caminhos disjuntos e ciclos que cobrem todos os vértices em G de forma que $|M| + |P| = |V|$.*

Como pode ser observado na equação do Teorema 1, a maximização do emparelhamento (inclusão de mais arestas não adjacentes) corresponde diretamente à minimização da quantidade de caminhos, já que o número total de vértices é fixo. As Figuras 3.2, 3.3 e 3.4 demonstram a formação de caminhos a partir do emparelhamento no grafo dobro. Como visto no teorema já referido, o emparelhamento pode retornar ciclos, então esta topologia precisa ser detectada em um segundo passo. Com componentes mais simples obtidos pelo emparelhamento, porém, isso se torna mais praticável de calcular.

3.2 Aplicabilidade da Proposta em Topologias Problemáticas

A técnica de emparelhamento é aplicada aos grafos *de Bruijn* ao invés de grafos de sobreposição, como era originalmente. Isto se deve à questão já mencionada de *reads* muito curtos produzidos por sequenciadores NGS, que podem resultar num grafo de sobreposição grande e complexo. Assim, segue-se a tendência de se utilizar grafo *de Bruijn* [Conway & Bromage, 2011; Nagarajan & Pop, 2013], de modo que os *reads* de mesma sequência sejam representados pelos mesmos vértices e, ainda, para que não haja a necessidade de comparação par a par que pode onerar demasiadamente o tempo de execução. A ideia é que o emparelhamento forneça caminhos que correspondam a uma simplificação do grafo, ou seja, a quebra das chamadas topologias problemáticas em componentes mais simples de se trabalhar posteriormente na geração dos *contigs* finais.

Nas figuras a seguir, demonstra-se a aplicação do emparelhamento máximo nas topologias problemáticas citadas na seção 2.3.5 (exceto ciclos). Na Figura 3.2, há um exemplo de aplicação do emparelhamento em erosões. Na Figura 3.3, é apresentado um caso de bolha com o respectivo cálculo de emparelhamento. Na Figura 3.4, percebe-se um emparelhamento obtido em casos de corda desgastada.

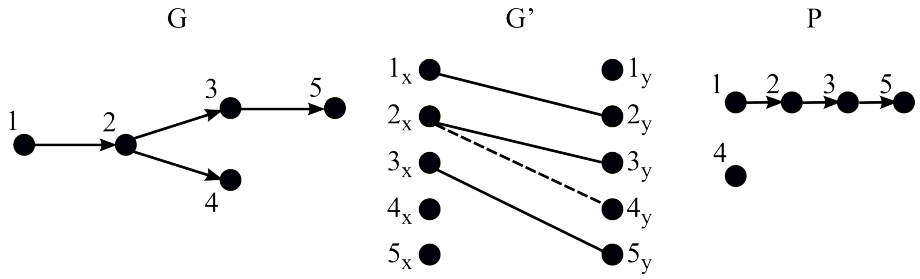

Figura 3.2. Cálculo de emparelhamento em trecho com Erosão. As linhas sólidas em G' representam o emparelhamento máximo escolhido e linhas tracejadas representam arestas excluídas. O conjunto P de caminhos pode facilmente ser obtido deste emparelhamento, bastando calcular as componentes conexas do grafo após o emparelhamento. Observa-se que as arestas escolhidas neste caso representam apenas uma das possibilidades de emparelhamento.

Esta fase, no caso de aplicação da técnica de emparelhamento em um grafo *de Bruijn*, trata da construção de *contigs* (caminhos) iniciais, que numa fase posterior serão combinados para gerar *contigs* maiores, diferentemente do caso de grafos de sobreposição, onde o emparelhamento já fornecia os caminhos finais, correspondentes aos *contigs* finais, restando só a combinação de possíveis ciclos a estes caminhos.

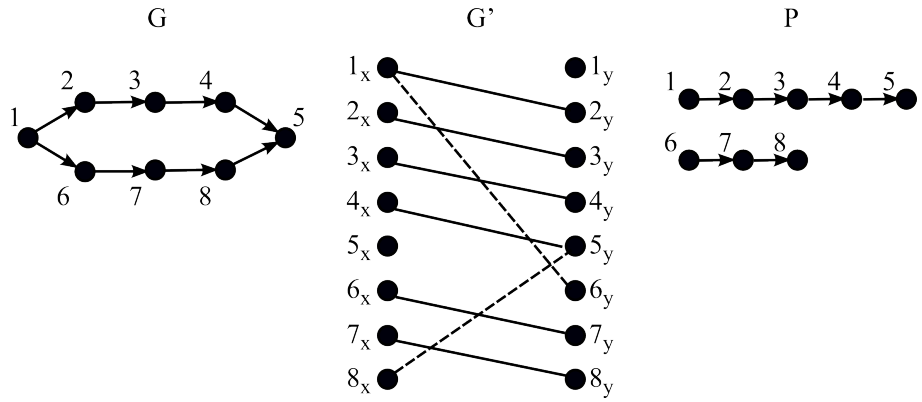

Figura 3.3. Cálculo de emparelhamento em trecho com Bolha. As linhas sólidas em G' representam um emparelhamento máximo escolhido e linhas tracejadas representam arestas excluídas. P é o conjunto de caminhos obtidos do emparelhamento. O emparelhamento apresentado é apenas uma das possibilidades (obtidas alternando-se a escolha entre as arestas 1–2, 1–6, 4–5 e 8–5).

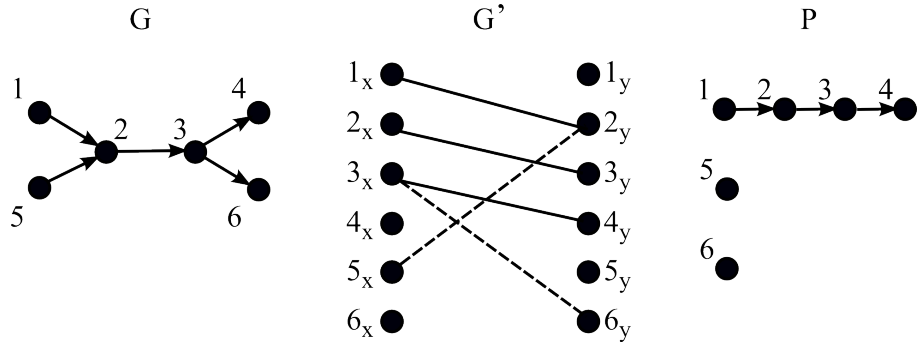

Figura 3.4. Emparelhamento em trecho com Corda Desgastada. Um dos possíveis emparelhamentos em G' é indicado pelas linhas contínuas. Linhas tracejadas representam arestas não selecionadas. Os caminhos obtidos de G' são representados pelo conjunto P. Alternando-se as arestas excluídas com suas vizinhas, obtém-se as demais possibilidades de emparelhamento.

Nas Figuras 3.2, 3.3 e 3.4, percebe-se que há mais de uma possibilidade de emparelhamento (alternando-se arestas tracejadas com uma vizinha contínua). A decisão de qual emparelhamento é mais adequado, ou seja, aquele que produzirá caminhos longos de *reads* corretos e isolará caminhos curtos correspondentes a *reads* incorretos, pode ser tomada baseando-se no peso das arestas. Neste caso, o emparelhamento máximo passa a ser emparelhamento máximo de peso máximo. Portanto, esta etapa também abrange, de forma implícita, a detecção de erros. Veja que o peso seria o número de vezes que a aresta ocorre no processo de montagem do grafo k-mer. Assim, um *read* real, que não seja consequência de algum problema

de sequenciamento, tende a aparecer várias vezes nos dados de entrada resultante do processo de sequenciamento e isto faz com que, no procedimento de geração de k-mer de tais *reads*, as mesmas arestas (que ligam os mesmos pares de k-mers) apareçam várias vezes. Já os *reads* incorretos aparecerão de forma isolada no conjunto de dados e os k-mers resultantes dos mesmos, bem como as arestas que os conectam, ocorrerão com baixa frequência.

Usando os pesos já mencionados, os emparelhamentos máximos de pesos máximos produzirão caminhos que abrangerão *reads* corretos, enquanto que os incorretos serão isolados em caminhos bastante curtos, até mesmo como um vértice único. O vértice 4 da Figura 3.2 e o caminho 6–7–8 da Figura 3.3 podem ser exemplos de casos assim, ou seja, em que a divergência de sequência que levou à topologia em questão foi ocasionada por erro de sequenciamento. Estes pequenos caminhos isolados de baixo peso são facilmente detectados e poderão ser eliminados antes de proceder à fase seguinte de combinação dos caminhos para geração de *contigs* maiores. A Figura 3.5 apresenta um exemplo do tipo de emparelhamento utilizado em nossa proposta, ou seja, o emparelhamento máximo de peso máximo, aplicado na topologia de corda desgastada.

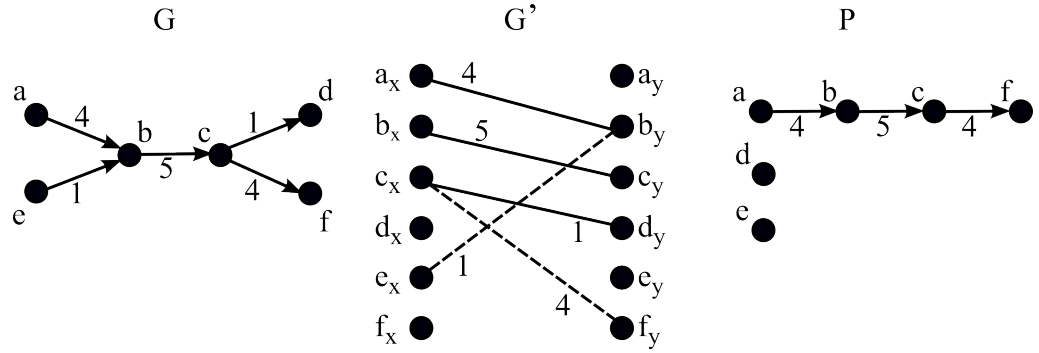

Figura 3.5. Emparelhamento máximo de peso máximo. Vértices são nomeados com letras e os números são o peso de cada aresta. Em G, temos o grafo original com pesos. G' representa o grafo dobro com o emparelhamento máximo de peso máximo (representado pelas arestas de traço contínuo). P é o conjunto de caminhos resultante. Neste caso, não há outra possibilidade de resposta.

Como é possível analisar na Figura 3.5, o cálculo do emparelhamento máximo que possua maior peso evita outras possibilidades de emparelhamento. Tendo em vista que áreas com erro de sequenciamento tendem a ter menos cobertura, este processo pode separar trechos indesejáveis do restante do grafo.

3.3 Variações que não são Ocasionadas por Erros de Sequenciamento

Apesar do emparelhamento máximo sem pesos resolver as topologias problemáticas nos casos em que há erro de sequenciamento, isto não acontece com topologias provocadas por situações inerentes do genoma. Os polimorfismos de um nucleotídeo (*single nucleotide polymorphism*, ou SNP), são casos em que dois *reads* semelhantes possuem apenas uma base diferente no meio da sequência desses. SNPs e outras características genuínas geram topologias em que não de deve descartar nenhuma das possibilidades de emparelhamento, ou seja, todas as possibilidades devem ser consideradas.

Na Figura 3.6, há um exemplo da abordagem em casos de topologias criadas por casos em que não há erro de sequenciamento envolvido, especificamente no caso de erosões. Da mesma forma que acontece no caso de erosões, as bolhas causadas por SNPs devem ter a sequência comum replicada, retornando ambas as possibilidades. A Figura 3.7 demonstra uma representação hipotética deste caso.

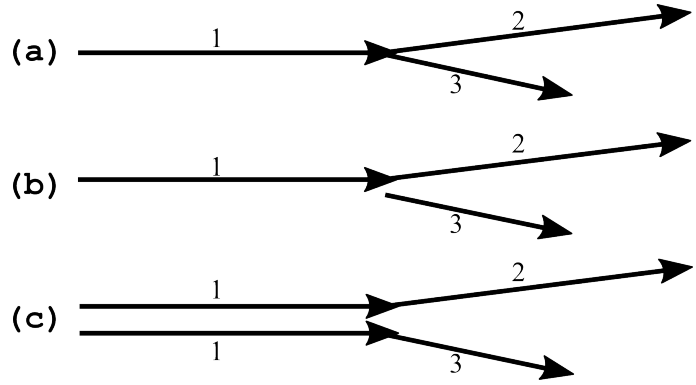

Figura 3.6. Solução em erosões causadas por situações válidas. Em (a), vê-se o grafo antes do emparelhamento (simplificado, com setas representando áreas contínuas). Em (b), tem-se a resposta do emparelhamento máximo de peso máximo. Por último, (c) apresenta a solução ideal. Os números nomeiam cada trecho.

Como pode-se perceber nas Figuras 3.6 e 3.7, trechos que são comuns em cada topologia (trecho 1, na erosão e trechos 1 e 4, na bolha) são duplicados para representar ambas as combinações possíveis. Assim, a erosão tem como resposta as combinações 1–2 e 1–3 e a bolha retorna as respostas 1–2–4 e 1–3–4.

Nos casos de corda desgastada e ciclo, sabe-se que ocorrem por regiões repetidas. Então, estas são claramente situações que não têm nada a ver com erros de sequenciamento. Mas, da mesma forma que em erosão e bolha causados por varia-

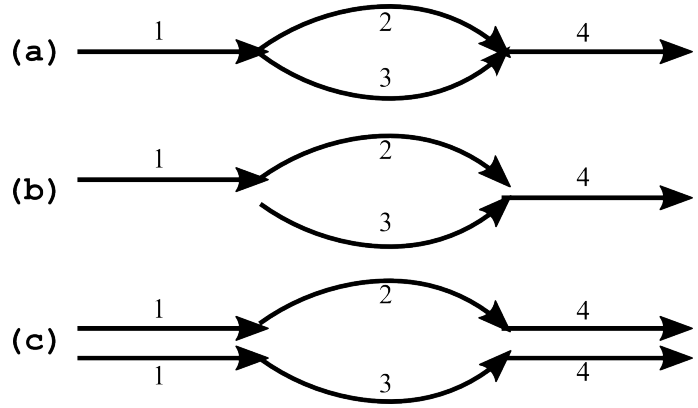

Figura 3.7. Solução alternativa para bolhas válidas. Em (a), tem-se o grafo antes do emparelhamento (simplificado, com setas representando áreas contínuas). Em (b), há o emparelhamento máximo de peso máximo. (c) apresenta a solução ideal. Cada trecho é rotulado com um número.

ções do genoma, há que se combinar os resultados do emparelhamento para geração de *contigs* que façam sentido. Esta fase pós-emparelhamento/pós-remoção de caminhos curtos de baixo peso é a fase, portanto, de combinação das componentes mais simples resultantes da primeira etapa. Em todos os casos, a região repetida é utilizada mais de uma vez para obtenção de *contigs* maiores. As Figuras 3.8 e 3.9 apresentam exemplos deste caso.

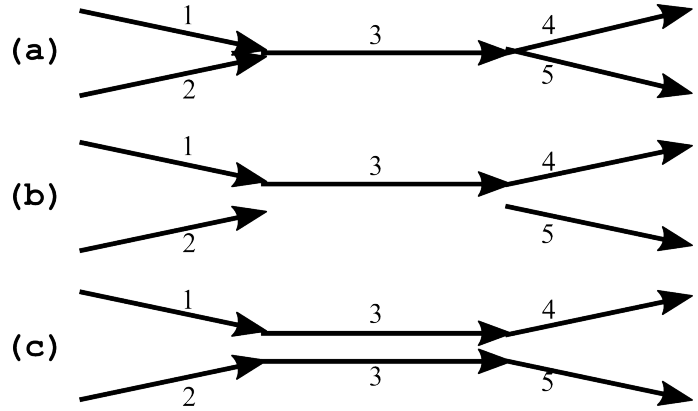

Figura 3.8. Proposta para a topologia de corda desgastada. Em (a), o grafo original (simplificado). Em (b), o emparelhamento. Já em (c), vê-se uma possível solução, seguindo a ideia de parcimônia. Números são usados para rotular cada parte.

Os casos de corda desgastada e ciclo também replicam os trechos comuns/repetidos para que se mantenha as possibilidades de combinação, sem descarte ou isolamento de partes mais curtas, já que ambas as opções são válidas. Assim, vê-se as respostas 1–3–4 e 2–3–5 para a corda desgastada. No caso de ciclo, por sua

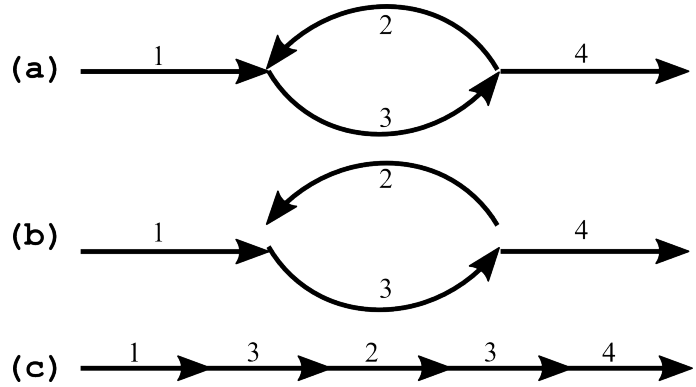

Figura 3.9. Uma solução para ciclos, seguindo a ideia de parcimônia. Em (a), o grafo inicial simplificado. (b) apresenta o emparelhamento máximo de peso máximo. (c) apresenta uma solução. Os algarismos representam cada trecho.

vez, o trecho de repetição (no examplo da Figura 3.9 representado pelo trecho 3) é replicado o mínimo de vezes necessário para que se tenha o ciclo "aberto", sem perda de informação (1–3–2–3–4).

3.4 Proposta de Montador

Em um primeiro momento, a meta-heurística Nuvem de Partículas foi implementada para o cálculo do emparelhamento máximo de peso máximo. Após testes comparativos com um método exato, verificou-se que a meta-heurística não oferecia ganhos de desempenho. Após estes testes, diversas estratégias para o cruzamento de caminhos foram levantadas. No fim, optou-se por uma abordagem heurística para extensão de caminhos. O funcionamento geral do protótipo desenvolvido é apresentado no Algoritmo 1.

Como pode-se perceber na linha 4 do Algoritmo 1, o montador começa construindo o grafo k-mer através da leitura dos dados de entrada (os *reads*). O grafo construído não é a sua forma "dobro", citada na seção 3.1. Ao invés disso, o grafo k-mer é montado da forma tradicional. Quando o grafo é analisado para cálculo do emparelhamento, porém, a navegação é feita de forma a simular um grafo dobro sobre o grafo simples. Assim, embora o procedimento seja baseado no emparelhamento sobre o grafo dobro, como já exposto, não há a geração explícita deste grafo, de modo a poupar memória e processamento.

Na linha 5 do Algoritmo 1, as componentes conexas são calculadas, simulando-se o grafo bipartido durante este processo. Desta forma, quando a leitura em profundidade é feita, esta leva em conta os dois papeis que um vértice do grafo simples

Algoritmo 1 Montador de DNA

```
1: procedure MontadorDeDNA(reads)                ▷ reads é uma lista de reads
2: Input: reads grupo de fragmentos sequenciados
3: Output: LP a lista de caminhos/passeios representando contigs
4:     G ← gerarGrafoKmer(reads)
5:     comps ← obterComponentesConexas(G)
6:     matchings ← EmparelhamentoMaxPesoMax(comps)
7:     G ← reagruparSolucao(matchings)
8:     NLP ← gerarUnipaths(G)▷ NLP: lista de unipaths (caminhos iniciais).
   Trechos com baixa cobertura são descartados, não se tornando unipaths.
9:     ScLP ← −1                  ▷ Score (pontuação) da lista atual de caminhos
10:    ScNLP ← calcLPScore(NLP)          ▷ Score da nova lista de caminhos
11:    while ScNLP > ScLP do      ▷ A nova lista de caminhos tem pontuação
   maior do que a anterior
12:        LP ← NLP                                ▷ LP: a lista atual de caminhos
13:        ScLP ← ScNLP
14:        for each outArc a do
15:            NLP ← gerarNovosCaminhos(LP, a)
16:        end for
17:        NLP ← removerRedundancia(NLP)
18:        ScNLP ← calcLPScore(NLP)
19:    end while
20:    return LP                                   ▷ LP: Lista de contigs
21: end procedure
```

pode ter no grafo dobro (partições x e y, por exemplo). Desta forma, um mesmo vértice é visitado duas vezes, sendo uma para cada papel. Para cada componente conexa, tem-se então o cálculo do emparelhamento máximo de peso máximo (linha 6). O algoritmo de Kuhn [Kuhn, 1955], posteriormente adaptado por Munkres [1957] efetua esta operação com complexidade $O(n^3)$, sendo n o número de vértices. Apesar da complexidade cúbica, não há impacto significativo de performance, já que o emparelhamento é calculado sobre cada componente conexa. As componentes conexas são de tamanho reduzido, tendo tamanho máximo $K_{4\times4}$ (devido ao alfabeto do DNA: `A`, `C`, `G`, `T`). A Figura

Como pode-se perceber na Figura 3.10, cada vértice pode ter no máximo 4 arestas de saída e 4 de entrada no grafo $k-$mer. Se pegarmos o vértice da `atcga`, por exemplo, os vértices vizinhos de arestas de saída possuem uma sequência que se inicia com `tcga`, seguida do sufixo de cada vizinho. Por haver 4 nucleotídeos no DNA, há apenas 4 possibilidades de sequência ara vizinhos. No caso dos vértices na partição da direita, ocorre a mesma limitação. Os vizinhos do vértice `tcgaa`, por exemplo, possuem sequências com um prefixo próprio, seguidos da sequência

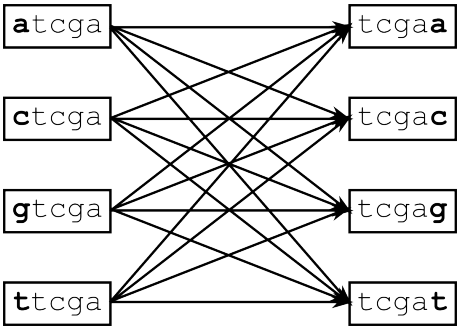

Figura 3.10. Exemplo de uma componente conexa com o tamanho máximo possível ($K_{4\times4}$).

comum `tcga`. Para que a componente conexa tivesse um grau máximo maior, ou seja, algum vértice com grau maior do que 4, novas letras deveriam ser adicionadas ao alfabeto do DNA, que só possui quatro membros (`A`, `C`, `G` e `T`), que representam os quatro nucleotídeos.

O cálculo do emparelhamento máximo de peso máximo é feito através de *threads*. Cada componente conexa é colocada em uma *thread* e novas são colocadas no lugar daquelas que já finalizaram o cálculo do emparelhamento. O número de *threads* é definido pelo usuário como parâmetro. Nota-se, portanto, que o fato de dividir o problema de emparelhamento em todo o grafo em diversos problemas de emparelhamento em pequenos grafos (componentes conexas) tem ainda a vantagem de possibilitar processamento paralelo.

Após o cálculo do emparelhamento máximo de peso máximo ser feito, há a leitura das arestas adicionais que geram alternativas para diferentes emparelhamentos. Estas arestas vizinhas às arestas não incluídas no emparelhamento encontrado são também removidas em seguida, a fim de criar os *contigs* iniciais (*unipaths*) em um próximo passo. A Figura 3.11 apresenta este conceito.

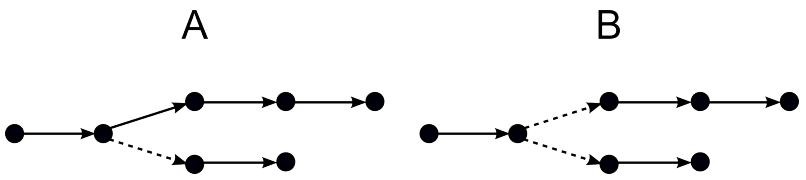

Figura 3.11. Emparelhamento com exclusão de arestas vizinhas. Em A, o emparelhamento calculado (linha tracejada representa aresta não escolhida na solução). Em B, a seleção (e exclusão) da aresta vizinha àquela que deixou de ser escolhida no emparelhamento. Ainda em B, os caminhos resultantes são os *unipaths* que serão combinados em fase posterior para gerar *contigs* maiores.

Na linha 7, procede-se com a junção dos emparelhamentos locais em um grafo

unificado. A combinação dos diversos emparelhamentos de cada componente conexa para gerar o emparelhamento final no grafo como um todo é um proxesso simples. Basta pegar a união de todos os emparelhamentos. No grafo unificado, há o cálculo das componentes conexas (linha 8), que representam os *unipaths* (*contigs* iniciais). Nesta fase, porém, o cálculo é feito sobre o grafo simples, não mais simulando um grafo dobro. Os *unipaths* são previamente salvos em um arquivo FASTA, para posterior manipulação. Vale ressaltar, porém, que componentes com cobertura equivalente a até 10% da cobertura geral do genoma são descartados durante a geração de *unipaths*, já na leitura de cada componente, para gerar a sequência do *unipath*. Esta é a fase antes da extensão dos *contigs*, em que se elimina os *unipaths* de baixo peso que são, provavelmente, consequência de *reads* com erro de sequenciamento.

Toda aresta excluída é armazenada para uso futuro em cruzamento de *contigs*. Esta exclusão das arestas vizinhas quebra ciclos que possam ter permanecido após o cálculo do emparelhamento máximo de peso máximo. A obtenção de *unipaths*, portanto, facilita a fase seguinte de combinação para geração de novos *contigs*.

3.4.1 Extensão de Caminhos

Após obter os *unipaths*, o montador executa um passo heurístico de extensão de caminhos (linhas 14–16 do Algoritmo 1). Basicamente, ele toma os arcos excluídos da construção de *unipaths* (emparelhamento e exclusão de arestas vizinhas) e utiliza-os para construir *paths* (caminhos, cada um sendo um conjunto de um ou mais *unipaths*), que são conjuntos de *unipaths* conectados. O primeiro grupo de caminhos considerado é a própria lista de *unipaths* (linha 8 do algoritmo). É a partir dela que se gera novos caminhos.

Todo caminho possui uma pontuação (*score*, em inglês) que é calculada de acordo com a Equação 3.1, elaborada neste trabalho.

$$S_{path} = \frac{c}{1 + \sum_{i=1}^{c} r_i} + c \tag{3.1}$$

A variável c na Equação 3.1 é o número de *unipaths* distintos dentro deste caminho. Por sua vez, r é a quantidade de vezes que um *unipath* se repete em todos os caminhos do grupo. Logo, r é o número de ocorrências de um *unipath* menos 1. O valor de ocorrência é atualizado constantemente e está armazenado dentro de cada *unipath*.

A Equação 3.1 beneficia caminhos que possuam mais *unipaths*, mas penaliza aqueles que tenham elementos repetidos, promovendo um incentivo a caminhos mais

longos e com o mínimo possível de *repeats*. Sendo assim, áreas de *repeat* em ciclos, por exemplo, são utilizadas somente o número de vezes necessário para cobrir os demais caminhos.

Finalizado o cruzamento de caminhos, há a remoção de caminhos idênticos no novo conjunto e também dos caminhos cujos *unipaths* cobertos já estejam presentes em caminhos maiores (e de maior pontuação), até que se tenha o mínimo possível de caminhos que mantenha a representatividade de todos os *unipaths* (linha 17).

Por fim, o conjunto de caminhos é avaliado e uma pontuação é gerada para esse (linha 18). Caso sua pontuação seja menor ou igual àquela da iteração anterior, o montador considera o conjunto anterior como resposta final e o salva em um arquivo FASTA. Se o *score* for maior, porém, outro cruzamento de caminhos é promovido, agora com os novos caminhos, em busca de outro conjunto melhor. A pontuação de um conjunto de caminhos é a média aritmética da pontuação de seus *paths* (caminhos), como mostrado na Equação 3.2.

$$S_{global} = \frac{\sum_{i=1}^{n} S_{path}}{n} \tag{3.2}$$

Na Equação 3.2, n representa o número de caminhos do conjunto de caminhos e o somatório é a soma da pontuação dos caminhos do grupo.

Uma demonstração da aplicação do algoritmo em um caso hipotético em que há ciclo é apresentada na Figura 3.12.

Conforme apresentado na Figura 3.12, os *unipaths* são a primeira lista de caminhos. Baseando-se nos arcos excluídos do arco original, combinações entre os *unipaths* são feitas (1ª rodada). Os novos caminhos gerados passam por um filtro que remove caminhos repetidos (ou aqueles contidos em caminhos maiores) e caminhos de menor pontuação, respeitando a representatividade mínima de todos os *unipaths*. A partir da primeira lista gerada por cruzamentos, promove-se nova rodada de geração de caminhos. Caso o *score* seja melhor, esta passa a ser a lista oficial de caminhos. Do contrário, a lista original é mantida. Este processo é repetido até que não se consiga melhorias com novos cruzamentos ou haja apenas um caminho restante. No exemplo dado, a 3ª rodada apresentou apenas um caminho após a remoção de redundâncias.

Como pode ser visto na Figura 3.12, o algoritmo minimiza o número de caminhos enquanto tenta fazer com que os caminhos cubram todos os *unipaths*, mas o menor número possível de vezes. Assim, ao final, nota-se a produção de somente um caminho, na verdade um passeio no grafo, que representa o *contig* final. A solu-

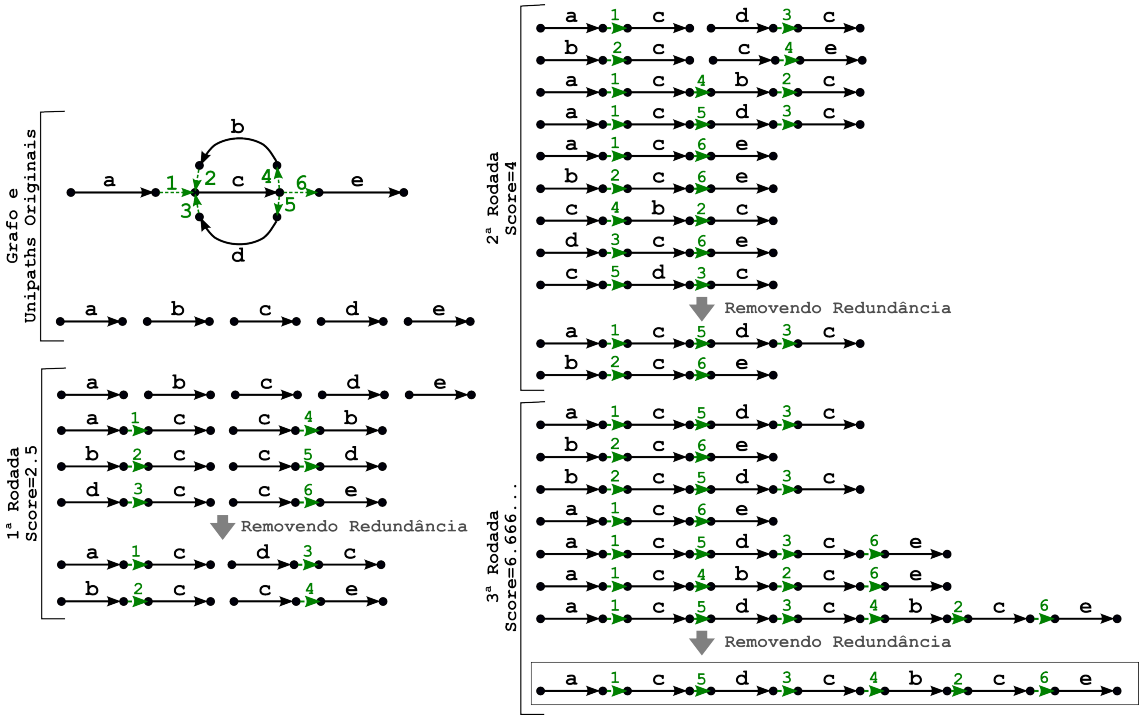

Figura 3.12. Simulação do algoritmo em caso de ciclo. Números rotulam arcos de conexão entre *unipaths*. Cada *unipath* é nomeado com uma letra.

ção é construída com o menor número de repetições possível, no caso do *unipath* *c*, para que seja gerado um único *contig*. Apesar de retornar uma solução possível, é perceptível que esta não é a única possibilidade de arranjamento dos *unipaths* para o *contig* final. Determinar qual a formação original, entre as diversas possibilidades, só seria possível com *paired-end reads*, o que foge do escopo desta abordagem.

3.5 Definição dos Testes

Para execução de testes na ferramenta, foi feita uma seleção de espécies de genomas pequenos para as primeiras execuções. Dependendo dos resultados, novas espécies seriam definidas. As primeiras espécies definidas para testes foram as listadas abaixo, por terem genomas bem pequenos:

***Candidatus Carsonella ruddii pv*[1]:** uma proteobactéria gama, endossimbionte (vive no interior da célula ou do corpo de outro ser vivo). O trabalho [Nakabachi et al., 2006] que publicou este genoma extraiu o material genético do interior de *Pachypsylla venusta* (um tipo de inseto que se alimenta de seiva). O genoma é composto por um único cromossomo circular de 159662 pares

de base, com uma média de 16,5% de conteúdo *GC* e está completamente montado no banco.

***Candidatus Nasuia deltocephalinicola*[2]:** segundo o trabalho responsável pela divulgação deste genoma [Bennett & Moran, 2013], é uma proteobactéria beta simbionte, que foi extraída de *Macrosteles quadrilineatus*, inseto que se alimenta de seiva. Seu genoma é circular e é considerado, segundo o responsável pela montagem, o menor genoma bacteriano já sequenciado, com 112091 pares de base. É um genoma finalizado.

***Candidatus Sulcia muelleri*[3]:** possui o genoma circular, com 190733 pares de base. É uma bactéria da ordem *Flavobacteriales*, simbionte e foi extraída durante o mesmo estudo [Bennett & Moran, 2013] que sequenciou o genoma da *Candidatus Nasuia deltocephalinicola*. O genoma possui o formato circular e foi totalmente montado.

***Candidatus Tremblaya princeps*[4]:** possui um genoma de 138931 pares de base, em formato circular. É uma proteobactéria endossimbionte e, no estudo que reportou seu genoma [López-Madrigal et al., 2011], foi extraída das células de *Planococcus citri*, um pequeno inseto. Seu genoma é considerado já completo.

Os dados de todas as espécies estão em formato FASTA, cuja estrutura é descrita no Apêndice A.

A tentativa de montagem foi realizada com *reads* simulados dessas espécies. A simulação dos *reads* foi feita através da ferramenta simLibrary[5], que simula dados da tecnologia Illumina. Cada espécie teve *reads* simulados com cobertura de 80 e 30, com fragmentos de 100 bases em ambos os casos. Os *reads* foram gerados de forma não pareada.

Em todos os testes, o valor de k utilizado foi 31. O valor ímpar de k, como informado por [Miller et al., 2010], evita a ocorrência de DNA palíndromo, ou seja, trechos de DNA cuja sequência reversa é a própria sequência direta, ou seja, um bloco de DNA que é igual ao seu complemento reverso. Segundo os mesmos autores, o montador Velvet, por exemplo, exige que o valor de k seja ímpar. Para escolher o valor de k, foi utilizado o parâmetro empregado pelo GAGE [Salzberg et al., 2011], comparativo de montadores de fragmentos de DNA. No site[6] do estudo, há a indicação da "receita" utilizada para os testes, em que o valor de $k = 31$ é mencionado

[5]European Bioinformatics Institute. Disponível em: http://www.ebi.ac.uk/goldman-srv/simNGS/simNGS/man/simLibrary.1.html

[6]http://gage.cbcb.umd.edu/

para todos os montadores testados em que o parâmetro se aplica. Mais detalhes sobre os testes serão apresentados no capítulo 4.

Capítulo 4

Resultados

Neste capítulo foi feita uma divisão dos resultados deste trabalho. Na seção 4.1, há uma descrição da complexidade de cada fase do processo de montagem desenvolvido. Em seguida, a descrição do ambiente onde os testes foram realizados é feita na seção 4.2. A seção 4.3 descreve os resultados obtidos durante os testes, tanto em relação à montagem dos genomas quanto ao tempo e à quantidade de memória gastos. Por fim, a seção 4.4 descreve os frutos deste trabalho.

4.1 Complexidade do Algoritmo

Considerando-se o montador proposto, há diversas sub-tarefas, cada qual com sua complexidade específica. Dividindo-se pelo objetivo central de cada parte do algoritmo, tem-se as seguintes etapas:

- Construção do grafo.
- Cálculo do emparelhamento máximo de peso máximo.
- Cruzamento de caminhos (*paths*).

Para compreender melhor os valores de complexidade de algoritmo, faz-se necessário o uso de um índice de variáveis $\times$ significado. Sendo assim, a seguinte nomenclatura é utilizada:

t Tamanho do arquivo.

c Cobertura do sequenciamento.

r Número de *reads*.

l Tamanho dos *reads*.

k Tamanho do k-mer.

v Número de vértices.

a Número de arestas.

u Número de *unipaths*.

u_2 Número de *unipaths* distintos dentro de um caminho.

p Número de caminhos.

x Número de arcos excluídos durante o cálculo de emparelhamento.

Durante a construção do grafo, a inserção de um vértice é feita apenas se o mesmo ainda não está presente. Logo, essa verificação é a parte mais executada. Para cada *read*, todos os seus k-mers são verificados no grafo. Sendo assim, o número de $k-$mers (incluindo repetições) gerados para verificação no grafo é dado, aproximadamente, por rlk, que será chamado de n para simplificação. Para as n consultas no grafo, o tempo de consulta na estrutura em tabela *hash* de vértices tende a ser constante, mas em casos raros, onde a chave de busca dos elementos apresentar conflitos, pode haver até v comparações dentro de uma determinada chave. O valor de v é determinado pelo número de k-mers possíveis dividido pelo valor da cobertura (c). Logo, $\frac{rlk}{c} = n$ (desconsiderando-se erros de sequenciamento). Assim, assintoticamente, a complexidade para a montagem do grafo é de $O(n^2)$ no pior caso.

O cálculo de emparelhamento máximo de peso máximo é um módulo de terceiros incorporado neste trabalho. A complexidade do código para emparelhamento máximo de peso máximo é de $O(v^3)$. Neste trabalho, porém, a complexidade no pior caso é $O(1)$, já que as componentes conexas são, no pior caso, um grafo completo K_4, ou seja, 4 vértices e 16 arestas. Sendo assim, a complexidade máxima é constante.

Na etapa de cruzamento de caminhos, cada arco excluído da fase de obtenção de *unipaths* é utilizado para a geração de novas combinações (x arcos excluídos). Para cada arco excluído, no pior caso, todos os caminhos possuiriam conexão no início e no fim deste arco, gerando $O(n^2)$ combinações de novos caminhos. Assim, a complexidade assintótica total de cada etapa de cruzamento de caminhos é $O(xn^2)$. O cruzamento de caminhos é realizado enquanto este retorne listas de caminho com

menor pontuação global do que a anterior, mas o número total de rodadas tende a ser pequeno, não tendo influência importante assintoticamente.

Quanto à complexidade de memória, a relação da memória aproximada necessária para execução do algoritmo depende fortemente do tamanho de k durante a montagem do grafo. Para um arquivo com r *reads*, a aplicação precisa de uma estrutura para armazenar cada um dos fragmentos. Sem entrar diretamente em números exatos de memória, o algoritmo demanda espaço para rl caracteres no pior caso, onde não há limpeza de dados em memória que não são mais necessários. Entretanto, o número de *reads* é muito maior do que o tamanho médio desses. Para cada um dos r *reads* lidos, produz-se $l-k+1$ vértices de tamanho k, ou $k(l-k+1)$ caracteres a partir de cada *read*. O valor de l, ou seja, o tamanho de um *read*, é sempre maior do que o valor de k. Para simplificação, pode-se considerar que são gerados l vértices de tamanho k, ou lk caracteres para cada *read*. Assim, durante a fase de construção do grafo, são gerados rlk caracteres de sequência para todo o arquivo. Como cada vértice tem um peso, há também a armazenagem de rl pesos (números inteiros). Considerando-se que cada vértice pode se conectar com até quatro vizinhos, cada vértice precisa armazenar ainda informação sobre as respectivas arestas. Cada aresta possui um peso correspondente. Levando-se em conta os rl vértices, há até $4rl$ pesos, um para cada aresta. Como resultado final, a montagem do grafo necessita de espaço suficiente para armazenagem de $rl + rlk$ caracteres e $5rl$ números inteiros.

Para o cálculo do emparelhamento máximo, o algoritmo precisa detectar as componentes conexas. Neste caso, subgrafos são criados, com replicação dos dados correspondentes a cada componente conexa do grafo original, em uma versão bipartida. O caso de maior gasto de memória ocorre quando se tem componentes conexas com quatro vértices em cada partição, em que os vértices de uma partição de conectam com os quatro da outra partição, originando 16 arestas. Como o grafo bipartido é a versão "dobro" e cada componente conexa possui 8 vértices, pode-se afirmar que o número de componentes conexas é a oitava parte do número de vértices. Logo, para rl vértices (chamaremos de v), há $\frac{2v}{8} = \frac{v}{4}$ componentes conexas. Para compor as componentes conexas, a ferramenta produz $2v$ vértices, armazenados nas novas estruturas. Como não há alteração no número de arestas na versão dobro do grafo, há a replicação do número de arestas do grafo original.

Quando o emparelhamento máximo de peso máximo é calculado e as arestas vizinhas são excluídas, um grafo com o resultado consolidado (união dos resultados de cada componente conexa) é formado, gerando um segundo grafo com todos os vértices, com gasto de memória próximo daquele produzido pelo grafo original.

Para a etapa de cruzamento de caminhos, o primeiro grupo de caminhos (formado pelos *unipaths*) possui aproximadamente o mesmo número de caracteres do primeiro grafo. Apenas trechos de baixa cobertura são excluídos. O menor tamanho que um *caminho* pode ter no início é igual ao tamanho de um vértice (k). O tamanho máximo é equivalente a $\frac{rl}{c}$, ou seja, o tamanho de todos os *reads* dividido pela cobertura das sequências fornecidas.

No pior caso, cada caminho inicial (*unipath* é equivalente ao tamanho de um vértice. Como cada caminho pode se conectar a até 4 vizinhos de um lado, é possível mensurar que há até 4^p possibilidades (deixando de lado a remoção de redundâncias). Assim, o impacto em memória é de $4^p k$ caracteres, combinado com 4^p pesos (um para cada caminho).

4.2 Execução dos Testes

Os testes foram iniciados no cluster da Divisão de Suporte ao Desenvolvimento Científico e Tecnológico da Universidade Federal de Viçosa (DCT/UFV), mas o sistema retornava erros de execução por quantidade insuficiente de memória. Por essa razão, transferiu-se os dados para um servidor de vários processadores e memória compartilhada do Laboratório de Bioinformática (LABINFO), no Laboratório Nacional de Computação Científica (LNCC)[1].

O servidor utilizado no LABINFO conta com 32 processadores Intel® Xeon® CPU E7-8837, com *clock* de 2.67GHz. Cada um dos processadores tem 8 núcleos de processamento. A memória compartilhada do equipamento é de 252,13 GB. Nos parâmetros de execução do programa (seção A.2, página 73), foi definido que seriam utilizadas 15 *threads* para execução dos testes.

Na Tabela 4.1, pode-se analisar o tempo de execução e a memória gasta durante os testes.

Como pode-se perceber, o tempo não possui uma relação exata com a quantidade de *reads* (cobertura 30 e 80), o que pode ser ocorrência do módulo heurístico, que pode achar a solução possível em tempo variado a cada rodada. A cobertura representa o número de cópias que foram feitas do genoma antes de fragmentá-lo e sequenciar seus fragmentos. Como base de comparação, a Tabela 4.2 apresenta o custo computacional de algumas ferramentas semelhantes.

É possível perceber que a quantidade de memória utilizada pelo protótipo desenvolvido neste trabalho é o principal ponto a ser melhorado. Proporcionalmente,

[1] http://www.lncc.br

Tabela 4.1. Tempo e Memória Gastos nos Testes

	Espécie	Tempo (ms)	Memória (GB)
Cobertura 30	Carsonella	153063	34,42
	Nasuia	127397	33,28
	Sulcia	203603	34,79
	Tremblaya	164155	34,79
Cobertura 80	Carsonella	138397	32,02
	Nasuia	70681	25,47
	Sulcia	164004	35,05
	Tremblaya	275077	33,28

O gasto de memória é percentual do total disponível (252,13 GB).

Tabela 4.2. Custos Computacionais de Alguns Montadores

Espécie	Genoma	Montador	Memória	Tempo
Caenorhabditis elegans[a]	100 mbp	SGA	4,5 GB	-
		AbySS	14,1 GB	-
		Velvet	23 GB	-
		SOAPdenovo	38,8 GB	-
Homo sapiens[a]	2,9 gbp	SGA	54 GB	140
		SOAPdenovo	118 GB	121
Campylobacter jejuni[b]	1,8 mbp	ALLPATHS	32 GB[c]	0,6
Yarrowia lipolytica[b]	20,5 mbp	ALLPATHS	64 GB[c]	19,5

[a] Dados extraídos do trabalho de Simpson & Durbin [2012].
[b] Extraído do material suplementar do artigo de Butler et al. [2008].
[c] Memória total disponível no equipamento, já que a memória realmente utilizada nestes exemplos não é informada no trabalho original.
"Genoma" é o tamanho do genoma original, representado em milhões ou bilhões de pares de base (mbp ou gbp, respectivamente). O tempo é dado em horas.

os outros montadores apresentam exigência menor de memória. O uso de outras estruturas de dados para representar as sequências (como cadeias de bits) pode ser uma boa alternativa.

4.3 Resultados Obtidos dos Testes

A análise dos resultados foi feita através da ferramenta BLAST [Zhang et al., 2000], na sua versão web[2]. O módulo utilizado foi o *blastn* com otimização *megablast* (sequências de alta similaridade).

[2]http://blast.ncbi.nlm.nih.gov/Blast.cgi

A Tabela 4.3 apresenta os resultados obtidos no sequenciamento de quatro espécies, com cobertura de *reads* igual a 30. Só foi colocado o nome do gênero na tabela, mas o nome completo das espécies pode ser visto na seção 3.5.

Tabela 4.3. Resultado do Sequenciamento com Cobertura = 30

	Genoma[a]	Consenso[a]	Identidade[a]	*Gaps*[a]	Fita	*Matches*
Carsonella	159662	69831	69831/69831	0	→	1
		89837	89837/89837	0	→	1
		159659	159659/159659	0	←	1
Nasuia	112091	157421	94824/94828	0	←	3[b]
		189086	94813/94817	0	→	5[b]
Sulcia	190733	79357	79357/79381	24	→	1
		112841	76476/76529	53	→	6[b]
		190634	178074/178178	103	←	4[b]
Tremblaya	138931	614894	75289/75289	0	→	86[b]
		656212	67012/67012	0	→	88[b]

[a] Valores apresentados em número de bases. "Genoma" representa o tamanho original do genoma da espécie testada. *Gaps* são lacunas (regiões não cobertas) e *matches* são trechos diferentes cobertos pelo consenso no genoma. Seta para a direita indica fita no mesmo sentido do genoma; seta para a esquerda, fita reversa.

[b] Nos casos em que há mais de um *match*, as colunas de Identidade, *Gaps* e Fita apresentam os valores do primeiro (e maior) *match*.

Como pode-se observar na Tabela 4.3, a amostra *Candidatus Carsonella ruddii pv* retornou correspondências (*matches*) únicas em cada consenso retornado, sendo que os dois primeiros são da mesma fita de DNA (e de trechos diferentes). O terceiro, por sua vez, representa a sequência reversa, tendo uma representação de toda a molécula (com algumas bases a menos nos extremos). Nas outras espécies, porém, foi retornado um grupo de sequências com múltiplos alinhamentos internos, com sobreposições entre os trechos (na tabela, os valores são do maior *match* em cada teste). A exceção ocorre no primeiro consenso da *Candidatus Sulcia muelleri*, que apresentou um trecho com 79357 bases de tamanho e apenas uma ocorrência (sequência da mesma área). Quanto ao número de lacunas (*gaps*), novamente a primeira espécie da tabela apresenta o melhor resultado, sem bases faltosas dentro do consenso. Apesar de algumas espécies terem apresentado mais de um *match*, o resultado é sempre composto de um *match* de maior extensão e outros de valor bem inferior ao tamanho do genoma.

Na Tabela 4.4, por sua vez, tem-se os resultados com cobertura de 80, onde percebe-se uma melhora nos resultados. Tanto *C. rudii* quanto *N. deltocephalinicola*

apresentaram montagens com um único *match* cada, sendo uma para a fita direta e a outra para a reversa. Isto acontece porque os dados de entrada possuem *reads* sequenciados de ambas as fitas de DNA. As duas últimas espécies, porém, apresentam ainda múltiplos *matches*, mas *S. muelleri* já passa a apresentar apenas dois consensos.

Tabela 4.4. Resultados de Teste com Cobertura de 80

	Genoma[a]	Consenso[a]	Identidade[a]	*Gaps*[a]	Fita	*Matches*
Carsonella	159662	159652	159652/159652	0	←	1
		159661	159661/159661	0	→	1
Nasuia	112091	112088	112081/112088	0	→	1
		112088	112081/112088	0	←	1
Sulcia	190733	191143	114702/114750	48	→	5[b]
		191155	114711/114750	39	←	5[b]
Tremblaya	138931	684627	69511/69511	0	→	89[b]
		656294	76389/76390	1	←	78[b]

[a] Valores apresentados em número de bases. "Genoma" representa o tamanho original do genoma da espécie testada. *Gaps* são lacunas (regiões não cobertas) e *matches* são trechos diferentes cobertos pelo consenso no genoma. Seta para a direita indica fita no mesmo sentido do genoma; seta para a esquerda, fita reversa.

[b] Nos casos em que há mais de um *match*, as colunas de Identidade, *Gaps* e Fita apresentam os valores do primeiro (e maior) *match*.

Tanto para a cobertura de 30 quanto para a cobertura de 80 há casos em que se encontra alguns *gaps*. Analisando-se a identidade destas espécies, percebe-se um número próximo na razão de identidade. Isto indica que os *gaps* eram geralmente de poucas bases, com valores próximos de 1 ou 2 bases. O tamanho do numerador e do denominador na coluna de identidade retrata as razões de identidade do maior *match* em cada caso. Em certos casos em que o valor de consenso retornou mais de um *match*, o consenso é maior do que o tamanho do genoma, ou seja, não o representa perfeitamente. Isto ocorre porque há ligações errôneas na montagem, resultando em sequências maiores. Nestes casos, há sempre um *match* que representa grande parte do genoma, com outros trechos menores conectados em seguida. Essa estrutura é o que causa o tamanho maior do que o genoma.

O número de *unipaths* e o de arcos excluídos interferem na complexidade da montagem. A Tabela 4.5 apresenta estas informações.

O número de *unipaths* e de arcos excluídos na Tabela 4.5 aumenta na cobertura de 80, se comparado ao resultado na cobertura inferior (30). O simulador de *reads* tenta reproduzir situações reais de sequenciamento, então pode acontecer,

Tabela 4.5. Número Total de *Unipaths* e Arcos Excluídos nos Testes

	Espécie	Unipaths	Arcos Excluídos
Cobertura 30	Carsonella	761	1.517
	Nasuia	488	962
	Sulcia	919	1.800
	Tremblaya	710	1.393
Cobertura 80	Carsonella	2029	4.059
	Nasuia	1.322	2.633
	Sulcia	2.473	4.915
	Tremblaya	1.741	3.458

por exemplo, de uma cobertura maior aumentar também o número total de erros presentes e outros complicadores, que provocam o surgimento de mais *unipaths* e arestas excluídas. Este aumento, porém, não prejudicou os resultados. Com mais informação na cobertura maior, os resultados foram melhores.

Houve tentativa de teste com outras espécies de genoma mais extenso, mas a estrutura computacional não comportou a memória demandada pelo algoritmo.

Em busca de melhores resultados nos casos em que a montagem retornou consensos com ligações errôneas, testes com valores maiores de k foram realizados, todos com cobertura igual a 80. As Tabelas 4.6, 4.7 e 4.8 apresentam os resultados obtidos.

Tabela 4.6. Resultados de Teste com Cobertura de 80 e $k = 41$

	Genoma[a]	Consenso[a]	Identidade[a]	*Gaps*[a]	Fita	Matches
Carsonella	159662	159662	159652/159652	0	←	1
		159661	159661/159661	0	→	1
Nasuia	112091	112088	112081/112088	0	←	1
		112088	112081/112088	0	→	1
Sulcia	190733	190656	190635/190714	79	→	3[b]
		190678	178150/178181	30	←	4[b]
Tremblaya	138931	581170	61228/61228	0	←	74[b]
		617686	45581/45581	0	→	84[b]

[a] Valores apresentados em número de bases. "Genoma" representa o tamanho original do genoma da espécie testada. *Gaps* são lacunas (regiões não cobertas) e *matches* são trechos diferentes cobertos pelo consenso no genoma. Seta para a direita indica fita no mesmo sentido do genoma; seta para a esquerda, fita reversa.

[b] Nos casos em que há mais de um *match*, as colunas de Identidade, *Gaps* e Fita apresentam os valores do primeiro (e maior) *match*.

É possível perceber nos testes que a montagem do genoma de *Sulcia* apresenta

Tabela 4.7. Resultados de Teste com Cobertura de 80 e $k = 51$

	Genoma[a]	Consenso[a]	Identidade[a]	*Gaps*[a]	Fita	Matches
Carsonella	159662	159652	159652/159652	0	←	1
		159661	159661/159661	0	→	1
Nasuia	112091	112088	112081/112088	0	→	1
		112088	112081/112088	0	←	1
Sulcia	190733	190769	90790/90797	3	←	5[b]
		190680	190674/190709	35	→	3[b]
Tremblaya	138931	281162	82398/82398	0	→	42[b]
		237056	69507/69507	0	←	28[b]

[a] Valores apresentados em número de bases. "Genoma" representa o tamanho original do genoma da espécie testada. *Gaps* são lacunas (regiões não cobertas) e *matches* são trechos diferentes cobertos pelo consenso no genoma. Seta para a direita indica fita no mesmo sentido do genoma; seta para a esquerda, fita reversa.

[b] Nos casos em que há mais de um *match*, as colunas de Identidade, *Gaps* e Fita apresentam os valores do primeiro (e maior) *match*.

Tabela 4.8. Resultados de Teste com Cobertura de 80 e $k = 61$

	Genoma[a]	Consenso[a]	Identidade[a]	*Gaps*[a]	Fita	Matches
Carsonella	159662	159661	159661/159661	0	→	1
		159617	159617/159617	0	←	1
Nasuia	112091	112088	112081/112088	0	→	1
		112088	112081/112088	0	←	1
Sulcia	190733	190731	190726/190731	5	←	3[b]
		190703	190703/190713	10	→	3[b]
Tremblaya	138931	485815	64863/64864	1	←	73[b]
		630508	64863/64864	1	←	105[b]
		310865	102579/102580	1	←	49[b]
		715075	102579/102580	1	←	109[b]

[a] Valores apresentados em número de bases. "Genoma" representa o tamanho original do genoma da espécie testada. *Gaps* são lacunas (regiões não cobertas) e *matches* são trechos diferentes cobertos pelo consenso no genoma. Seta para a direita indica fita no mesmo sentido do genoma; seta para a esquerda, fita reversa.

[b] Nos casos em que há mais de um *match*, as colunas de Identidade, *Gaps* e Fita apresentam os valores do primeiro (e maior) *match*.

melhora em valores maiores de k, já com $k = 41$. No valor anterior, com k-mer de 31 pares de base, o consenso retornado era maior do que o genoma original, o que não ocorre com valores maiores. Para $k = 61$, o genoma apresenta o valor mais próximo do tamanho real do genoma. O genoma de *Tremblaya* apresenta

significativa melhora na montagem com $k = 51$, mas o resultado piora para $k = 61$. Este genoma possui trechos de *repeat* com mais de 5 mil nucleotídeos, tornando o genoma significativamente mais complexo para montagem.

As ligações entre diferentes fitas do DNA poderia ser evitada, em tese, com soluções de controle da sequência de cada fita. Apesar do valor de k influenciar diretamente na qualidade da montagem, esse não é suficiente. Novos testes com o controle das sequências de cada fita e com a calibração de valores de parâmetro são indicados.

4.4 Participação em Eventos e Publicação

Durante todo o período de pesquisa, houve participações em alguns eventos e a publicação de um artigo. Primeiramente, ocorreu a apresentação de um pôster no I Encontro Mineiro de Modelagem Computacional (EMMCOMP), em 2011, realizado na Universidade Federal de Juiz de Fora. Intitulado "Proposta de solução para o problema de montagem de fragmentos de DNA para sequenciadores de nova geração", o pôster se baseava em uma das primeiras propostas discutidas, que envolvia a aplicação de uma metaheurística no lugar do método de Munkres para cálculo do emparelhamento máximo de peso máximo.

Foi apresentado um poster também no X-Meeting 2011, evento organizado pela Associação Brasileira de Bioinformática e Biologia Computacional (AB3C) em Florianópolis. Intitulado "*Resolving Problematic Topologies in k-mer graphs: a proposal of a new method for DNA fragment assembly*", o trabalho também abordava a estratégia de uso de uma metaheurística para solucionar as topologias (emparelhamento máximo de peso máximo).

Por último, foi apresentado um artigo na *XXXI International Conference of the Chilean Computer Science Society* (SCCC 2012), intitulado "*Theoretical basis of a new method for DNA fragment assembly in k-mer graphs*" [Couto et al., 2012]. Este artigo, por sua vez (disponível no Apêndice B), já aborda uma proposta parecida com a deste trabalho, diferenciando-se na construção de caminhos após a obtenção dos *unipaths*.

Capítulo 5

Discussão

5.1 Resultados *versus* Objetivos

O objetivo geral deste trabalho era o desenvolvimento de uma nova abordagem de montagem, que evitasse ao máximo as etapas de pré e pós-processamento (seção 1.2, página 3). Após executar os testes com as quatro espécies, pôde-se perceber que a ferramenta consegue efetuar a montagem completa em metade dos casos com cobertura igual a 80 e $k = 31$. Para k a partir de 41, *Candidatus Sulcia muelleri* apresenta uma montagem adequada, já que os múltiplos *matches* não interfere na qualidade do maior *match*, que obteve tamanho próximo do original. Apesar de não ter atingido resultados completos nas última espécie, este resultado já permite construir uma validação inicial da ferramenta.

Nos resultados em que se obteve consensos de menor qualidade, percebe-se que a ambiguidade é um dos complicadores. A presença de dados de ambas as fitas do DNA, por exemplo, podem criar casos em que se obtém consensos são maiores do que o genoma original. Trechos de fitas diferentes que possuam trechos similares podem se conectar durante a construção do grafo, criando arcos (excluídos durante o cálculo do emparelhamento) que gerarão combinações de caminho que não representam a realidade. A Figura 5.1 apresenta um exemplo deste caso.

Na Figura 5.1, vê-se que alguns fragmentos de ambas as fitas produzem k-mers em comum. Isso provoca a conexão errônea entre as duas fitas de DNA, prejudicando o resultado final. Apesar de não acontecer em todos os casos, é interessante analisar possibilidades de filtragem de dados de entrada, se possível de forma unificada com a montagem do grafo, evitando a presença de dados de ambas as fitas. No exemplo da figura, os consensos possuem sequência de tamanho 10, maior do que o genoma original (8 nucleotídeos).

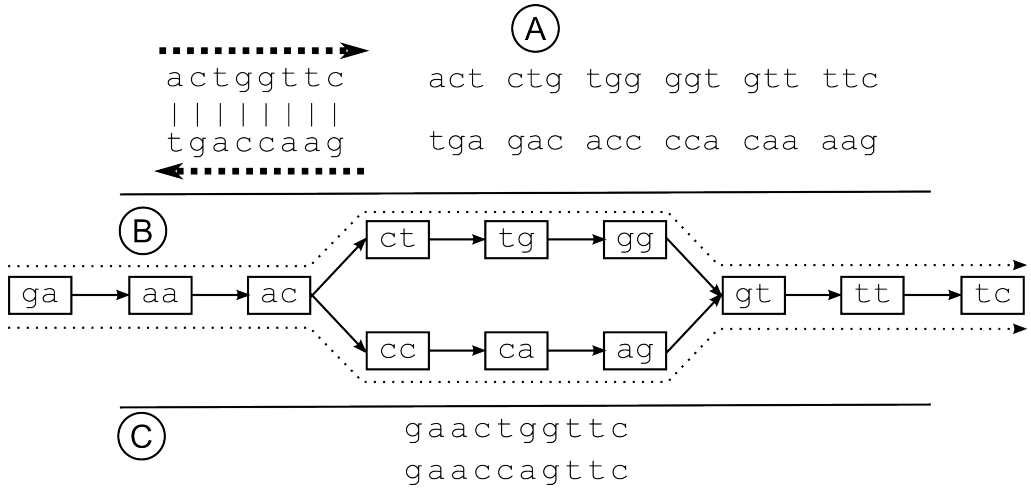

Figura 5.1. Conexões entre sequências de fitas diferentes. Em (A), vê-se um genoma fictício, com o sentido de cada fita destacado por setas pontilhadas. Os fragmentos de tamanho 3 gerados a partir dos fragmentos estão à direita. Em (B), o grafo k-mer, com $k = 2$, é apresentado. Os consensos obtidos são representados por setas pontilhadas. Em (C), os consensos em forma textual.

O montador Velvet [Zerbino & Birney, 2008] (seção 2.4.4.2), por exemplo, utiliza uma forma adaptada do grafo k-mer, em que todo k-mer carregado no grafo tem imediatamente seu complemento calculado e conectado, formando um "nó". Isso unifica os dados, evitando que a fita direta e a reversa sejam representadas de forma separada. Assim, não há a possibilidade da sequência de uma fita se conectar com a sequência da outra. O SHARCGS [Dohm et al., 2007] (seção 2.4.1.3), por sua vez, também possui um controle dos *reads*, buscando evitar que a sequência de um *read* esteja presente mais de uma vez no grafo e, para cada *read*, gera seu complemento reverso automaticamente.

O valor de k, ou seja, o comprimento da sequência de cada vértice, também influencia no resultado da montagem. Para *Candidatus Sulcia muelleri*, um valor de k igual a 41 já foi suficiente para parar de retornar um consenso com o tamanho maior do que o genoma original, eliminando ligações errôneas. No caso dos testes com *Candidatus Tremblaya princeps*, o valor de $k = 51$ reduziu bastante o tamanho do consenso, que era muito maior do que o genoma. Porém, esse ainda manteve ligações errôneas. Para $k = 61$, o resultado piorou. É compreensível que este genoma apresente mais problemas do que os outros três testados, já que possui áreas de *repeat* com cerca de 5 mil pares de base, o que dificulta bastante a montagem.

É interessante que se implemente o controle das sequências de cada fita em próximas versões desta ferramenta. Este controle e a calibragem de valores de k para cada genoma podem trazer melhorias importantes nos resultados.

A complexidade e a quantidade de memória, se comparados com o tamanho total dos genomas, são altas. Entretanto, por se tratar de um trabalho inicial, é aceitável que este apresente problemas e alguns ajustes necessários. A diminuição da complexidade e a utilização de *threads* em mais partes do algoritmo podem contribuir para a melhoria da eficiência do algoritmo.

Apesar de apresentar problemas em alguns reasultados, há que se concordar que a ferramenta mostrou ser possível efetuar a montagem em menos passos, já que alguns genomas foram montados. A diminuição da memória gasta possibilitará o teste do montador em outros cenários.

O fato do processo de cruzamento de caminhos ser uma heurística torna difícil prever com exatidão o tempo total gasto de acordo com o tamanho dos dados, como pode ser observado na Tabela 4.1 (página 55). Nesse contexto, o tempo de processamento do algoritmo não é completamente influenciado pelo tamanho dos dados de entrada. O tamanho da entrada influencia principalmente os passos executados de forma exata, até a geração de *unipaths*. O uso de delimitadores de tempo máximo ou número máximo de ciclos de cruzamento de caminhos representa uma boa alternativa caso os tempos de execução se mostrem maiores em outros casos experimentais.

Embora a solução tenha sido inicialmente validada, é necessário que se promova testes com outras espécies (mais complexas e de genoma maior) e também comparativos com montadores de contexto similar para que se tenha uma ferramenta mais robusta e competitiva. Testes de montagem híbrida (em que se emprega mais de uma ferramenta durante a montagem) também podem ser positivos, mas fogem da proposta deste protótipo.

Outra alternativa para diminuir os custos computacionais da ferramenta pode ser o uso de uma linguagem que não necessite de máquina virtual, ou seja, que trabalhe de maneira independente. Estas linguagens tendem a ser de mais leve processamento e algumas delas são muito fortes na área de Bioinformática, como Perl[1] (com seus pacotes para a área, no projeto BioPerl[2]), Python[3] e outras. Estas linguagens não foram utilizadas neste trabalho por causa da curva de aprendizado necessária para utilizá-las. Além disso, a linguagem Java foi inicialmente selecionada por conta de algumas bibliotecas já desenvolvidas por terceiros que poderiam ser incorporadas. Após alterações no protótipo, porém, apenas o código de emparelhamento máximo de peso máximo foi incorporado.

[1] http://www.perl.org/
[2] http://www.bioperl.org/wiki/Main_Page
[3] http://www.python.org/

O desenvolvimento deste protótipo demonstra a real aplicabilidade da técnica e indica a necessidade de aprimoramentos para que esse possa se tornar uma ferramenta para uso da comunidade de genômica em geral.

Capítulo 6

Conclusões e Perspectivas

6.1 Conclusões

A solução desenvolvida conseguiu efetuar a montagem de alguns genomas de forma satisfatória. Para outros, alguns problemas não puderam ser totalmente resolvidos. Apesar disso, pode-se concluir que a solução foi validada. O algoritmo evitou as correções de erros em k-mers e correções para cada topologia problemática, pois foram realizadas implicitamente na execução do algoritmo, além dos pós-processamentos, diminuindo o número de passos para a montagem.

Apesar de ter mostrado que a proposta é promissora, porém, percebe-se que há algumas melhorias a serem realizadas. Adaptações na estruturas de dados empregadas atualmente, bem como o corte na complexidade de algumas operações realizadas e na memória gasta podem proporcionar o refinamento que a ferramenta precisa, se transformando em uma ferramenta mais madura.

Percebe-se que a cobertura influi diretamente na qualidade das montagens, tendo em vista a mudança na qualidade da montagem do genoma de uma das espécies com a mudança da cobertura de 30 para 80. Os erros de sequenciamento, como se tornou possível perceber, podem ser contornados pelo emparelhamento máximo de peso máximo e a aplicação do emparelhamento apenas nas componentes conexas oferece diminuição na complexidade real do código.

Por fim, conclui-se que este trabalho atingiu os objetivos gerais e específicos, pois a abordagem foi desenvolvida, aplicada em espécies selecionadas e teve seus resultados analisados. Como uma ferramenta validada inicialmente, porém, há algumas melhorias e alterações possíveis em trabalhos futuros.

6.2 Trabalhos Futuros

Esta pesquisa abre espaço para diversas etapas de continuidade, englobando melhorias, correções e ampliações. No atual estágio, os trabalhos futuros identificados são:

- Diminuir a complexidade geral do algoritmo;
- Alterar as estruturas de dados atuais, buscando corte nos gastos de memória;
- Alterar a maneira como *strings* são representadas na ferramenta, diminuindo o gasto de memória;
- Incorporar análise da qualidade das bases sequenciadas durante a determinação do peso de vértices e arestas do grafo;
- Filtrar dados para controle de *reads* de fitas diferentes (evitar duplicidade).
- Testar o algoritmo em espécies com genomas maiores;
- Efetuar testes comparativos com outras ferramentas.

Referências Bibliográficas

Baker, M. (2012). De novo genome assembly: what every biologist should know. *Nat Meth*, 9(4):333–337.

Bennett, G. M. & Moran, N. A. (2013). Small, smaller, smallest: The origins and evolution of ancient dual symbioses in a phloem-feeding insect. *Genome Biology and Evolution*, 5(9):1675–1688.

BLAST (2013). Web blast page options. http://www.ncbi.nlm.nih.gov/BLAST/blastcgihelp.shtml. Acesso em: 03 dez. 2013.

Boisvert, S.; Laviolette, F. & Corbeil, J. (2010). Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies. *Journal of Computational Biology*, 17(11):1519–1533.

Butler, J.; MacCallum, I.; Kleber, M.; Shlyakhter, I. A.; Belmonte, M. K.; Lander, E. S.; Nusbaum, C. & Jaffe, D. B. (2008). Allpaths: De novo assembly of whole-genome shotgun microreads. *Genome Research*, 18(5):810–820.

Cerqueira, F. R. & Meidanis, J. a. (2001). Algorithms for Large-Scale DNA Sequencing. In *SEMISH 2001, proceedings of the Brazilian Computer Society Congress*.

Chaisson, M. J. & Pevzner, P. A. (2008). Short read fragment assembly of bacterial genomes. *Genome Research*, 18(2):324–330.

Chevreux, B. (2005). *MIRA: An Automated Genome and EST Assembler*. PhD thesis, Ruprecht-Karls University, Heidelberg, Germany.

Chevreux, B.; Wetter, T. & Suhai, S. (1999). Genome sequence assembly using trace signals and additional sequence information. In *Computer Science and Biology: Proceedings of the German Conference on Bioinformatics (GCB) 99*, pp. 45–56.

Compeau, P. E. C.; Pevzner, P. a. & Tesler, G. (2011). How to apply de Bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–91.

Conway, T. C. & Bromage, A. J. (2011). Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486.

Couto, A. D.; Cerqueira, F. R.; Guerra, R. L.; Goncalves, L. B.; Goulart, C. d. C.; Siqueira-Batista, R.; Ferreira, R. d. S. & Oliveira, A. d. P. (2012). Theoretical basis of a new method for dna fragment assembly in k-mer graphs. In *Chilean Computer Science Society (SCCC), 2012 31st International Conference of the*, pp. 69–77.

de la Bastide, M. & McCombie, W. R. (2007). *Assembling Genomic DNA Sequences with PHRAP*, chapter 11: Unit 11.4. John Wiley & Sons, Inc.

Dohm, J. C.; Lottaz, C.; Borodina, T. & Himmelbauer, H. (2007). Sharcgs, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Research*, 17(11):000.

Gnerre, S.; MacCallum, I.; Przybylski, D.; Ribeiro, F. J.; Burton, J. N.; Walker, B. J.; Sharpe, T.; Hall, G.; Shea, T. P.; Sykes, S.; Berlin, A. M.; Aird, D.; Costello, M.; Daza, R.; Williams, L.; Nicol, R.; Gnirke, A.; Nusbaum, C.; Lander, E. S. & Jaffe, D. B. (2011). High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518.

Green, P. (1994). Documentation for phrap and cross_match. http://www.phrap.org/phredphrap/phrap.html. Acesso em: 22 dez. 2013.

Jeck, W. R.; Reinhardt, J. A.; Baltrus, D. A.; Hickenbotham, M. T.; Magrini, V.; Mardis, E. R.; Dangl, J. L. & Jones, C. D. (2007). Extending assembly of short dna sequences to handle error. *Bioinformatics*, 23(21):2942–2944.

Kircher, M. & Kelso, J. (2010). High-throughput dna sequencing ? concepts and limitations. *BioEssays*, 32(6):524–536.

Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97.

Li, R.; Zhu, H.; Ruan, J.; Qian, W.; Fang, X.; Shi, Z.; Li, Y.; Li, S. et al. (2010). De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20(2):265–272.

López-Madrigal, S.; Latorre, A.; Porcar, M.; Moya, A. & Gil, R. (2011). Complete genome sequence of "candidatus tremblaya princeps" strain pcval, an intriguing translational machine below the living-cell status. *Journal of Bacteriology*, 193(19):5587–5588.

Luo, R.; Liu, B.; Xie, Y.; Li, Z.; Huang, W.; Yuan, J.; He, G.; Chen, Y. et al. (2012). Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. *GigaScience*, 1(1):18.

Men, A. E.; Wilson, P.; Siemering, K. & Forrest, S. (2008). Sanger dna sequencing. In Janitz, M., editor, *Genome Sequencing: Towards Personalized Medicine*, chapter Sanger DNA Sequencing, pp. 3–11. Wiley-VCH Verlag GmbH & Co. KGaA, Weinheim, Germany.

Metzker, M. L. (2010). Sequencing technologies — the next generation. *Nat Rev Genet*, 11(1):31–46.

Miller, J. R.; Koren, S. & Sutton, G. (2010). Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327.

Molaei, S.; Vahdani, B. & Molaei, S. (2013). A molecular algorithm for an operation-based job shop scheduling problem. *Arabian Journal for Science and Engineering*, 38(11):2993–3003.

Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38.

Myers, E. W. (1995). Toward simplifying and accurately formulating fragment assembly. *J. Comput. Biol.*, 2(2):275–290.

Myers, E. W. (2005). The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85.

Myers, E. W.; Sutton, G. G.; Delcher, A. L.; Dew, I. M.; Fasulo, D. P.; Flanigan, M. J.; Kravitz, S. A.; Mobarry, C. M. et al. (2000). A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204.

Nagarajan, N. & Pop, M. (2013). Sequence assembly demystified. *Nat Rev Genet*, 14(3):157–167.

Nakabachi, A.; Yamashita, A.; Toh, H.; Ishikawa, H.; Dunbar, H. E.; Moran, N. A. & Hattori, M. (2006). The 160-kilobase genome of the bacterial endosymbiont carsonella. *Science*, 314(5797):267.

National Human Genome Research Institute (2011). A brief guide to genomics. http://www.genome.gov/18016863. Acesso em: 02 dez. 2013.

Paszkiewicz, K. & Studholme, D. J. (2010). De novo assembly of short sequence reads. *Briefings in Bioinformatics*, 11(5):457–472.

Pevzner, P. A.; Tang, H. & Waterman, M. S. (2001). An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753.

Pop, M. (2009). Genome assembly reborn: recent computational challenges. *Briefings in Bioinformatics*, 10(4):354–366.

Salzberg, S. L.; Phillippy, A. M.; Zimin, A.; Puiu, D.; Magoc, T.; Koren, S.; Treangen, T. J.; Schatz, M. C. et al. (2011). Gage: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*.

Sanger, F.; Nicklen, S. & Coulson, A. R. (1977). Dna sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467.

Schatz, M. C.; Delcher, A. L. & Salzberg, S. L. (2010). Assembly of large genomes using second-generation sequencing. *Genome Research*, 20(9):1165–1173.

Scitable (2013). Dna is a structure that encodes biological information. http://www.nature.com/scitable/topicpage/dna-is-a-structure-that-encodes-biological-6493050. Acesso em: 25 dez. 2013.

Simpson, J. T. & Durbin, R. (2012). Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556.

Simpson, J. T.; Wong, K.; Jackman, S. D.; Schein, J. E.; Jones, S. J. & Birol, n. (2009). Abyss: A parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123.

Sommer, D.; Delcher, A.; Salzberg, S. & Pop, M. (2007). Minimus: a fast, lightweight genome assembler. *BMC Bioinformatics*, 8(1):64.

Staden, R. (1979). A strategy of dna sequencing employing computer programs. *Nucleic Acids Research*, 6(7):2601–2610.

Thorburn, W. M. (1918). The myth of occam's razor. *Mind*, 27(107):345–353.

Venter, J. C.; Adams, M. D.; Myers, E. W.; Li, P. W.; Mural, R. J.; Sutton, G. G.; Smith, H. O.; Yandell, M. et al. (2001). The sequence of the human genome. *Science*, 291(5507):1304–1351.

Warren, R. L.; Sutton, G. G.; Jones, S. J. M. & Holt, R. A. (2007). Assembling millions of short dna sequences using ssake. *Bioinformatics*, 23(4):500–501.

Zerbino, D. R. & Birney, E. (2008). Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829.

Zhang, W.; Chen, J.; Yang, Y.; Tang, Y.; Shang, J. & Shen, B. (2011). A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies. *PLoS ONE*, 6(3):e17915.

Zhang, Z.; Schwartz, S.; Wagner, L. & Miller, W. (2000). A greedy algorithm for aligning dna sequences. *J. Comput. Biol.*, 7(1-2):203–214.

Apêndice A

Informações Técnicas e de Uso do Montador

O protótipo desenvolvido neste trabalho segue um formato padrão para dados de entrada, chamado FASTA. Além do formato do arquivo de entrada, é importante observar a estrutura de comando utilizada para executar o protótipo. Essas informações são abordadas neste apêndice. Além dessas informações, dados específicos da linguagem utilizada são citados.

A.1 Arquivos FASTA

Arquivos FASTA são utilizados para armazenar sequências de DNA (ou outros tipos de informações genéticas) em formato texto. Nesses arquivos, cada sequência é representada por uma linha de descrição da sequência, que começa com o símbolo "$>$" (sinal de maior). Na linha seguinte, a sequência de DNA é apresentada, com as letras A, C, G, T e algumas letras com significado específico, como a letra N, que representa uma base que não foi identificada e, como resultado, pode ser qualquer uma das bases possíveis [BLAST, 2013]. Na Figura A.1, é mostrado um arquivo FASTA hipotético.

Figura A.1. Exemplo de um arquivo FASTA com apenas uma sequência. Em casos reais, o formato do cabeçalho descritivo é específico para cada aplicação e as sequências podem ser mais extensas.

A.2 Execução da Ferramenta

Para executar o montador, que é desenvolvido em Java, deve-se executar o seguinte comando:

`java -jar ‘‘montador.jar’’ tamanho_k numero_threads arquivo_entrada cobertura num_vertices`

Na linha de comando apresentada, os campos devem ser apresentados exatamente na ordem mostrada, tendo cada um a seguinte função:

- `java -jar`: Execução de programa em Java com a extensão “.jar”;
- `‘‘montador.jar’’`: representa o arquivo compilado do montador, que neste caso está no mesmo diretório de onde se executa o comando. Deve ser escrito da forma apresentada, ou seja, entre aspas duplas;
- `tamanho_k`: tamanho de cada k-mer no grafo (número inteiro positivo);
- `numero_threads`: número de núcleos de processamento disponíveis para o programa (número inteiro positivo);
- `arquivo_entrada`: referência para o arquivo FASTA, não sendo compatível arquivos com qualidade das bases ou outros formatos;
- `cobertura`: cobertura dos *reads* fornecidos (utilizado para cálculo de baixa cobertura);
- `tamanho_genoma`: tamanho aproximado do genoma (número de bases);

O tamanho do genoma é utilizado para alocação de memória na estrutura *hash* interna do grafo k-mer, evitando realocações grandes de memória das estruturas de

dados do algoritmo no decorrer de sua execução. Os valores de qualidade para cada base sequenciada, presentes em arquivos similares ao padrão FASTA, não são aceitos pelo protótipo desenvolvido, já que o objetivo é desenvolver um processo unificado, sem correções antes ou após a montagem. Considera-se que o nível de cobertura de cada parte pode, a princípio, ser bom indício da confiabilidade dos dados de entrada.

A.3 Desenvolvimento da Ferramenta

O montador foi desenvolvido em linguagem Java, com kit de desenvolvimento JDK[1] versão 6. A escolha por esta versão do JDK (ao invés da mais recente, versão 7) ocorreu por conta de restrições do *cluster* da Divisão de Suporte ao Desenvolvimento Científico e Tecnológico da Universidade Federal de Viçosa (DCT/UFV). O cálculo de emparelhamento máximo de peso máximo é realizado com uma implementação[2] do algoritmo de Kuhn disponível na Internet. A entrada de dados é feita através de arquivo FASTA sem dados de qualidade de bases. A ferramenta não possui interface gráfica de interação, sendo manipulada via terminal.

[1] http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html

[2] http://fypwcoupon.googlecode.com/svn/trunk/Mobivacy/src/edu/mit/csail/BipartiteMatcher.java

Apêndice B

Artigo Publicado na 31st International Conference of the Chilean Computer Science Society (2012)

2012 31st International Conference of the Chilean Computer Science Society

Theoretical Basis of a New Method for DNA Fragment Assembly in *k*-mer Graphs

Adriano Donato Couto[1], Fabio Ribeiro Cerqueira[1], Rafael Luciano Guerra[1],
Luciana Brugiolo Gonçalves[1], Carlos de Castro Goulart[1],
Rodrigo Siqueira-Batista[2], Ricardo dos Santos Ferreira[1], Alcione de Paiva Oliveira[1]
Postgraduate Program in Computer Science
[1]*Department of Informatics and* [2]*Department of Medicine and Nursing*
Federal University of Viçosa (UFV), Minas Gerais, Brazil
{*adriano.couto, fabio.cerqueira, rafael.luciano, lbrugiolo, goulart, ricardo, alcione*}*@ufv.br*
rsiqueirabatista@yahoo.com.br
* *Couto and Cerqueira contributed equally to this work.*

***Abstract*—The reduction of cost and running time provided by new generation sequencing technologies made possible the emergence of thousands of genome projects in the last few years. On the other hand, those technologies posed important computational challenges, pushing the advance of many research fields in computer science. Particularly, the *de novo* DNA fragment assembly, which is a fundamental stage in genome sequencing, is a complex problem that demands complex algorithms to solve it. Here, we provide a theoretical basis for the construction of a new method for *de novo* fragment assembly based on *k*-mer graphs. Our proposal encompasses many difficulties found in such problems using a unique procedure, in contrast with current methods that use several high-cost procedures to overcome the same issues. Furthermore, our approach is highly scalable since it allows the use of parallelism, being very suitable for solutions with graphics processing unit (GPU).**

***Keywords*-DNA fragment assembly; k-mer graphs; maximum matching;**

I. Introduction

DNA sequencing methods made possible a striking revolution on molecular biology, where the completion of the Human Genome Project [1] is probably the most important example. DNA sequencing is the first, and fundamental, stage of a more complex study of a genome. Since the proposal of Frederick Sanger's method for DNA sequencing [2], several genomes could be sequenced, resulting in important outcomes for biology, medicine, and agriculture, just to mention a few. On the other hand, Sanger's method is very expensive, which motivated the emergence of the so-called new generation sequencing (NGS) methods (examples can be found in [3]–[11]), where the cost decreased from USD 1,000/Mb (Megabases) to USD 22.50/Mb. Another significant innovation of NGS was regarding the decrease of running time needed for sequencing a complete genome, making possible to get gigabases of sequences in just a week [12], [13]. Using Sanger method, one would take 6 months to sequencing a fungus genome, whereas with NGS the same task could be accomplished in only 24 hours. For this reason, NGS promoted an exponential increase in the number of genome projects, pushing the improvement/development of important computational tools involved in the sequencing pipeline. This can make possible a huge revolution in biological and biomedical research [14].

In this paper, we are interested in the particular problem of assembling short DNA sequences. Figure 1 shows a general view of DNA sequencing and what is the input data to DNA assemblers. Notice that all known technologies for sequencing have a limitation concerning the size of a DNA molecule being sequenced. Considering that a chromosome has typically millions of bases, it has to be broken in several short pieces with size values obeying the limitation of the technology being applied. It means that, in order to have the complete (consensus) sequence of the original molecule, the sequenced fragments have to be assembled in the same order that they appear in the molecule where they came from. This particular problem is termed *de novo* DNA fragment assembly, which is known to be NP-hard [15]. This issue was satisfactorily solved for the Sanger method, where fragments have around 1000 nucleotides [16], [17]. However, since 2005, when NGS technologies came into notice, the assembly problem arose once more as an interesting research topic. The reason for this is that reads (fragment sequences) produced by NGS technologies are much smaller than reads coming from the Sanger method. The SOLiD technology, for instance, gives rise to reads of 50 bp. Table I shows the length limitation for each platform. Such small sizes mean a much higher input to fragment assembly algorithms as well as a significant augmentation of the problem complexity since the possibilities of ambiguity (many putative positions for a read in the assembly) are considerably increased [18].

Some DNA fragment assemblers are string-based and work using a greedy extension of the consensus sequences being produced. This kind of assembler is usually focused on assembling small genomes [19]–[21]. For genomes of general sizes, assembly algorithms normally use a graph approach in their strategies. Some of them apply overlap graphs [15], [22], where each node represents a read, and

1522-4902/13 $31.00 © 2013 IEEE
DOI 10.1109/SCCC.2012.16

Table I
READ LENGTH RETURNED FOR EACH PLATFORM.

Platform[a]	Read Length (Bases)
Roche/454's GS FLX Titanium	330[b]
Illumina/Solexa's GAII	75 or 100
Life/APG's SOLiD 3	50
Polonator G.007	26
Helicos BioSciences HeliScope	32[b]
Helicos Pacific Biosciences (target release: 2010)	964[b]

[a] Data extracted from Table 1 of Metzker review [8].
[b] On average.

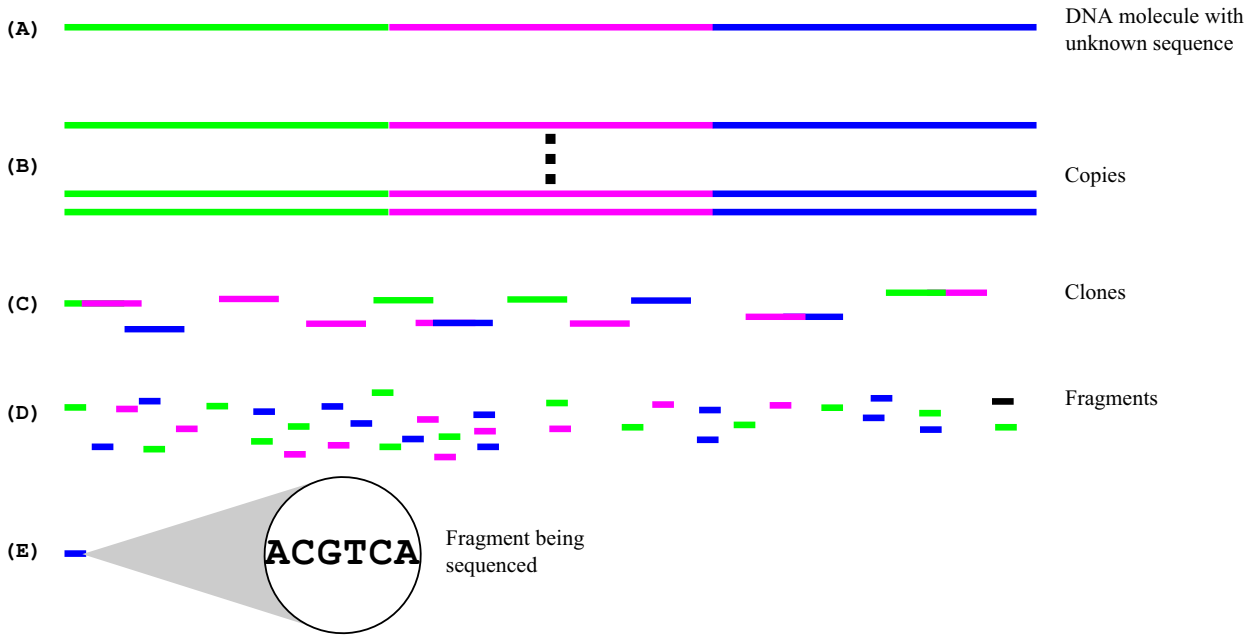

Figure 1. Sequencing process of an hypothetic DNA molecule. (A) The original DNA molecule. (B) The molecule is amplified. (C) A first level of fragmentation that generates big fragments. (D) A second level of fragmentation to produce pieces with appropriate lengths for the sequencing platform being used. (E) Fragments can then be sequenced. After these steps, the reads are given as input to the DNA assembler, which is responsible to join the obtained sequences in the correct order aiming to achieve the original DNA sequence.

an arc (i, j) is added if and only if a suffix of the read represented by i matches a prefix of the read represented by j. Figure 2 has an example of an overlap graph. Hence, to build and overlap graph, comparisons between all pair of reads have to be performed, which may be prohibitive considering the plethora of reads resultant from genomes of complex organisms. For this reason, many recent assembly algorithms model reads and their relationships using a *k*-mer graph [15], [22], where no explicit sequence comparison is necessary. In both strategies, the objective is to find paths (ideally one single path) in the graph that will determine the sought order of the reads. In the case of *k*-mer graphs, walks might be more appropriate than paths when the graph has cycles due to the presence of repeated regions of the DNA molecule.

In this work, we explore the *de novo* DNA fragment assembly problem modeled as a *k*-mer graph. In the following sections, we show the meaning of a *k*-mer graph as well as the main difficulties found by algorithms using this approach, and we provide a theoretical basis of a new method for finding walks in this sort of graphs in such a way that the typical issues do not have to be treated as distinct problems, as seen in previous approaches. Instead, our proposed method encompasses all difficulties in a single procedure based on maximum matching on bipartite graphs. It is also shown that our approach can make use of graphics processing unit (GPU) solutions to improve the turn around time of assembly.

II. BACKGROUND

A *k*-mer graph is a directed graph, where the nodes represent all subsequences with a fixed length *k* derived from a larger sequence, and the arcs denote overlaps of size $k-1$ between the subsequences [15], [22], [23]. Figure 3 shows a

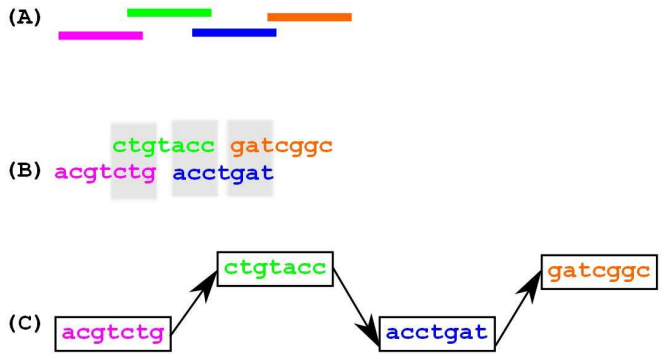

Figure 2. An example of overlap graph obtained from 4 sequenced fragments. (A) The DNA fragments. (B) Their correspondent sequences with overlap areas highlighted with gray rectangles. (C) The overlap graph obtained after comparing each pair of sequences.

read (a) and the corresponding k-mer graph (b) using $k = 4$ (in real data, higher values of k are used).

In a real assembly problem, a k-mer graph represents all reads given as input. Notice that each read results in a path, and reads having perfect overlap lead to a common path in the graph. Therefore, overlaps are drawn without the necessity of performing pairwise alignments. Figure 4 demonstrates the implicit overlap between two reads by means of the induced path in the resulting k-mer graph.

However, base-call errors may arise, i.e., bases that were erroneously reported by the sequencer machine. Sequencing errors have a stronger effect on k-mer graphs when compared to overlap graphs. It is so because the latter graph type is constructed based on approximate sequence alignments, whereas the former represents only perfect alignments. On the other hand, NGS provides a very high coverage of the molecule being sequenced, that is, there are many copies of the same read representing each region of the target molecule. As a result, a high degree of redundancy is observed in NGS data, which means that erroneous sequences are compensated by all the other correct copies of the same sequence. According to Miller [22], [23], k-mer graphs are also very sensitive to repeats, i.e., repeated sequences in the genome. Notice that this feature is the main source of ambiguity in the process of assembly (even for overlap graph approaches).

Miller and coauthors innumerate a series of problematic topologies of k-mer graphs constructed for real data:

- **Spur:** Short dead-end divergence (Figure 5(a)). This topology is caused by base-call errors present in one end of a read;
- **Bubble:** Divergence of a path followed by a convergence (Figure 5(b)). Also caused by base-call errors but in the middle of a read;
- **Frayed rope:** Convergence of paths followed by divergence (Figure 5(c)). This topology arises as a consequence of the presence of repeats in the genome;
- **Cycle:** Also a consequence of repeats.

In order to cope with the difficulties presented in Figure 5, many important computational tools, such as Velvet [24], AllPaths [25], and Euler [26], perform intensive pre/post-processing procedures to handle spur, bubble, frayed rope, and cycle patterns. Each such procedure is performed separately and has a high computational cost. Here, we propose a method that can be applied directly to the graph, meaning a unique assembly procedure encompassing all illustrated situations.

III. Proposed Method

A. Using maximum matching to obtain simpler components of the k-mer graphs

Remember that the ideal situation (in acyclic graphs) is finding a unique path representing the target molecule. Considering that the existence of such a path is unlikely due to the problems already cited, we seek for long paths representing contigs, i.e., contiguous regions in the original molecule. More precisely, we seek for a minimum path cover (i.e., the set with the least number of paths, such that every vertex in the graph belongs to at least one path), where paths have initial/final vertices with in/outdegree $= 0$. The minimization is due to our parsimony criterion, or Occam's razor, i.e., "the simplest possible explanation is the best" [27]. Notice that all possible paths might be a huge number to deal with. Besides, minimizing the number of paths leads to a tendency to produce longer paths. The restriction initial/final vertices with in/outdegree $= 0$ is to avoid interrupted (small) contigs.

We propose a method based on maximum matching on a bipartite version of the k-mer graph being analyzed. Notice that it was proposed in 2001 for simple overlap graphs [28], [29]. Here, we show the applicability of the method also for

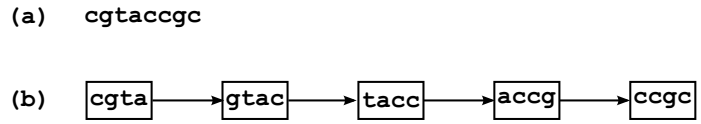

Figure 3. A read and its k-mer graph. (a) The read. (b) The k-mer graph for the read in (a) using $k = 4$.

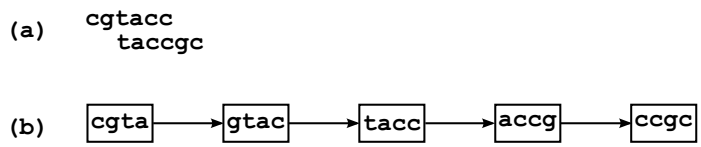

Figure 4. A k-mer graph representing the overlap between two reads. (a) Overlap of 4 bases between two reads. (b) A k-mer representing the reads for $k = 4$. The alignment is naturally drawn from the graph.

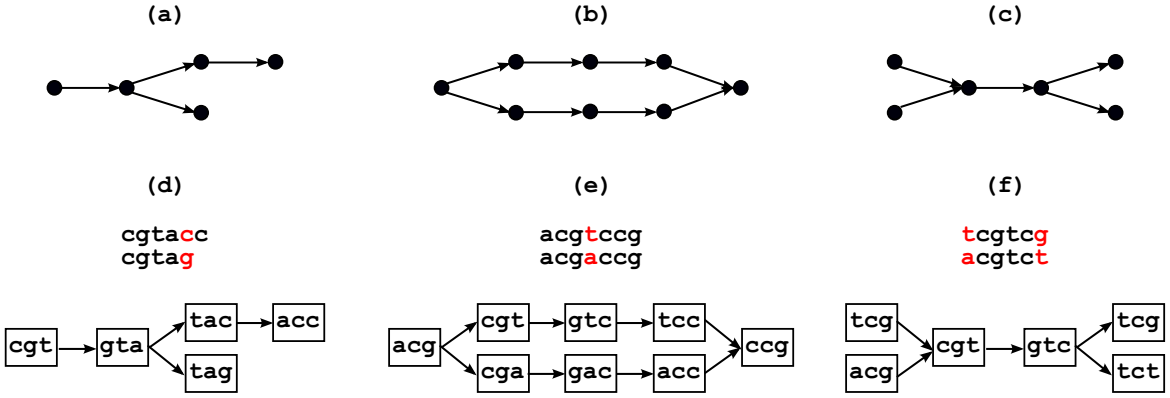

Figure 5. Problematic topologies in k-mer graphs. (a) Spur. (b) Bubble. (c) Frayed rope. (d) Example to illustrate the spur pattern. (e) Illustration of bubble pattern. (f) Example for the frayed rope pattern.

k-mer graphs and further explore it to each particular case illustrated in Figure 5.

Let $G = (V, A)$ be a directed graph. The *double* of G is an undirected bipartite graph $G = (V1, V2, E)$, where $V1 = V2 = V$ and E is composed as follows. For each arc $(u, v) \in A$, we have an edge $uv \in E$, where $u \in V1$ and $v \in V2$. Therefore, $|A| = |E|$. See Figure 6 for a simple example. Based on this definition, we pose the following theorem (see complete proof in the work of Cerqueira and Meidanis [28]).

Theorem 1: Let G be a directed graph and G' its double. Every matching M in G' corresponds to a set P of disjoint paths and a set of cycles that cover all the vertices in G such that $|M| + |P| = |V|$.

The equation of Theorem 1 makes it clear that the matching maximization means the minimization of the number of paths, since the number of vertices is constant. Therefore, we can produce the double of a k-mer graph and find a maximum matching on the resulting graph to obtain the smallest number of disjoint paths in the original graph. The idea of getting the smallest number of disjoint paths covering all vertices is for obtaining long paths to use them as a decomposition of the original graph in components for which the described problematic topologies can be easily recognized.

Let us illustrate the application of the above theorem to each case reported in Figure 5. Figure 7 shows the graph G corresponding to the spur topology, its double G', and a set P of disjoint paths obtained from the maximum matching on G'.

Notice in Figure 7 that the edge $2_x 4_y$ could be taken instead of edge $2_x 3_y$, leading to paths 1-2-4 and 3-5, which makes sense since a graph can have more than one maximum matching. The best maximum matching is one that isolates the path with k-mers corresponding to wrong reads, so that this path can be easily recognized and eliminated. A solution to find such a matching is shown throughout the text.

Figure 8 depicts the application of our approach to the bubble topology. In this case, there are four possible solutions. One is shown in the figure. Another putative solution is composed by paths 1-6-7-8-5 and 2-3-4. The best alternative here is also the one that isolates the path containing sequences with base-call errors. Such sequences can be detected using the fact that they are probably repre-

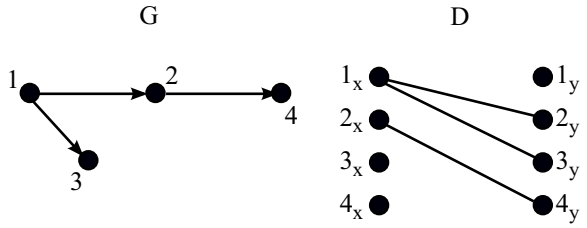

Figure 6. A directed graph G and its double G'. Nodes in G' have labels x and y for distinguishing partitions.

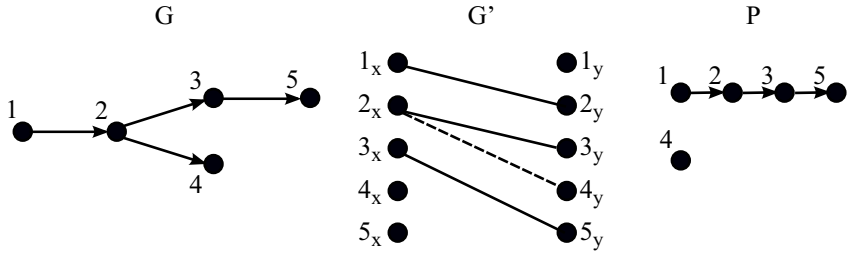

Figure 7. Applying maximum matching for the spur topology. Solid lines in G' denote the edges forming a maximum matching. Paths P are easily obtained following selected edges in G'.

sented by *k*-mers of low coverage, i.e., nodes representing subsequences of only few fragments.

Finally, we show in Figure 9 the application of our procedure for the frayed-rope topology. In this case, there are also four possible solutions. This situation cannot be avoided in the case of a repeat as it necessarily causes such kind of ambiguities. The combination of obtained components to solve the problem is presented in the following paragraphs. The important lesson of Figures 7, 8, and 9 is that the matching procedure breaks the complex k-mer graph into simpler components (paths and eventually cycles) for which elimination and combination can be performed to solve such topologies.

B. Solving spurs and bubbles generated by sequencing errors

For the case of spurs and bubbles caused by base-calls errors, the question is: How could we identify the best maximum matching, i.e., the one that isolates the path representing the sequence with errors, so that it can be easily eliminated? The answer turns out to be by moving the maximum cardinality matching problem to the maximum weighted maximum cardinality matching problem (MWMCMP). The weight of each arc (and, therefore, of each edge in the double graph) is measured based on its occurrence during the graph construction. For each time a connection between a pair of *k*-mers is found once more in another read, the occurrence of the corresponding arc is increased by one. Hence, erroneous connections tend to have low frequency, e.g., 1. As a result, the solution for the MWMCMP leads to the isolation of paths of low weight, which can be easily detected and eliminated from the final solution (vertex 4 of Figure 7 and the path 6-7-8 of Figure 8 could be such erroneous isolated paths).

One could argue that, even though the MWMCMP is tractable (solvable by Kuhn-Munkres algorithm, which is $O(n^3)$ for $K_{n,n}$ [30], [31]), the huge size of a *k*-mer graph would lead to a very large instance of this problem, resulting in a very low procedure. Notice, however, that the highest value of the in/outdegree of any vertice is limited to the size of the alphabet for DNA sequences. In the worst case, a connected component of a double graph is a bipartite subgraph of the form $K_{|\Sigma|,|\Sigma|}$, where Σ is the alphabet. Thereby, instead of trying to solve a huge instance (the entire double graph) of the MWMCMP, we propose to solve multiple small instances, corresponding to each connected component of the double graph. The merge procedure of local solutions into a global solution is trivial (the union of all selected edges). This divide-and-conquer possibility has the additional advantage of providing room for speeding up the whole process by the use of parallelism, such as GPU technologies.

C. Combining components for cases with no sequencing errors

It is important to notice that bubbles and spurs might be genuine. i.e., they might arise due to natural biological divergence such as single nucleotide polymorphisms (SNPs), rather than base-call errors. In such cases, no path from the matching procedure is of low weight, which implies that none of them can be removed. Instead, the resulting paths have to be combined to generate a solution. In the case of bubbles, spurs, and also frayed rope, such a solution is a path cover. This is illustrated in Figures 10, 11, and

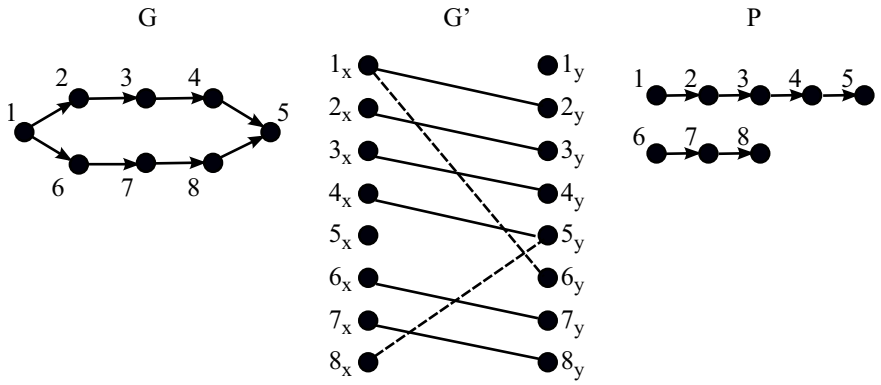

Figure 8. Applying maximum matching for the bubble topology. Solid lines in G' denote the edges forming a maximum matching. Paths P are obtained following selected edges in G'.

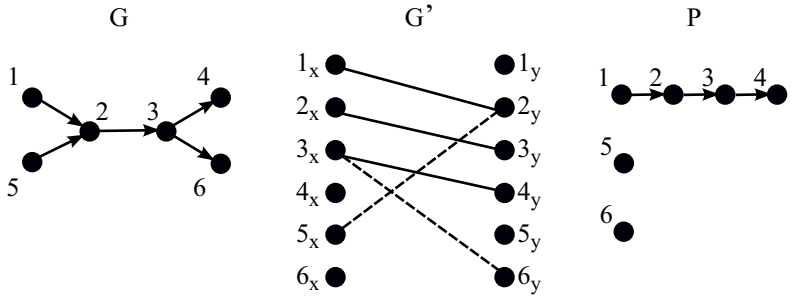

Figure 9. Applying maximum matching for the frayed-rope topology. Solid lines in G' denote the edges forming a maximum matching. Paths P are obtained following selected edges in G'.

12. In all cases, we can see that the matching procedure decomposed the given topology into components that can be easily recognized as part of such a structure, facilitating also the way we can merge such components to generate the ideal paths. The combination of paths is accomplished by the use of arcs that were not selected by the matching, and also by the duplication of the common parts.

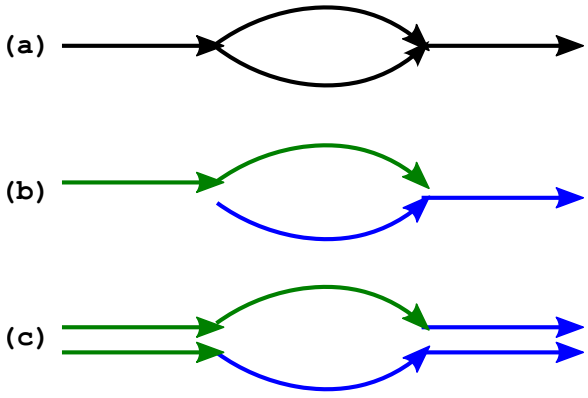

Figure 10. An example of how to solve genuine bubbles. (a) The original topology. (b) Paths derived from the matching. (c) The topology solved. Each arrow represents a path (a group of k-mers).

In the case of frayed rope, the ambiguity caused by repeats is not an error but an inherent issue in the assembly problem. In this way, we just run the matching procedure and try to combine the resulting paths into the set forming the minimum path cover (Figure 12). If no additional information is available, then we take any possible combination. On the other hand, the ideal combination could be achieved when the sequence process is based on paired-end reads. In this approach, the distance between many pair of reads in the target molecule is known a priori. The distances can then be used to infer the ideal path cover (a description of the paired-end sequencing technique can be found in [15], [22]).

For cycles, there are no errors either, and the idea is also to combine the resulting paths, but, in this case, to generate a walk (Figure 13). In the cycle case, a certain sequence of vertices (corresponding to a repeat) has to be covered more than once to compose the part of the graph representing a contig with repeats. Therefore, the term walk is more appropriate than path. As a result, and following the parsimony criterion, we seek for a combination of the paths (and possibly cycles produced by the matching) that leads to the shortest walk covering all vertices of these components. Figure 13 shows two paths (blue and green) that can be generated by the matching on a cycle topology and how

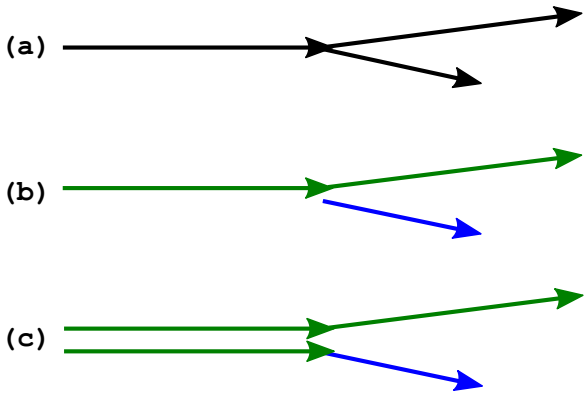

Figure 11. An example of how to solve genuine spurs. (a) The original topology. (b) Paths derived from the matching. (c) The topology solved. Each arrow represents a path (a group of k-mers).

these paths can be combined. The blue path is incorporated into the green path by duplicating the part corresponding to a repeat (dashed lines), resulting in the sought walk. This repeat part is easily identified because it is bounded by the arcs (not selected by the matching) that connect the green path with the blue one. Note that, when available, information about paired-end reads can be also used in this case to find the ideal walk.

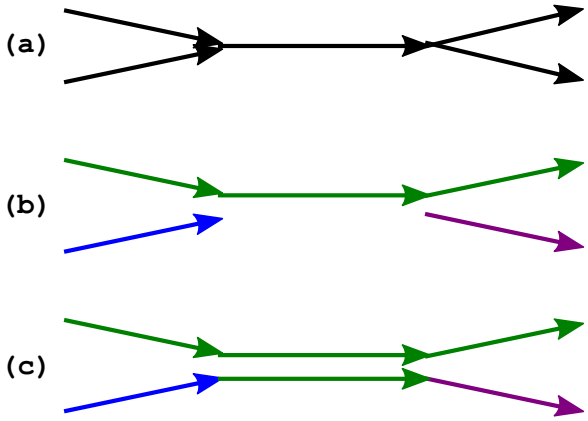

Figure 12. An example of how to solve frayed rope. (a) The original topology. (b) Paths derived from the matching. (c) The topology solved. Each arrow represents a path (a group of k-mers).

It is important to observe that for all cases of topologies the matching procedure can produce a different outcome from the ones shown as examples in the figures. However, the other possibilities are either symmetric to what we have shown or distinct but treated in the same fashion.

IV. Final Remarks

In this work we provide a theoretical basis for the construction of a new method for *de novo* sequence assembly

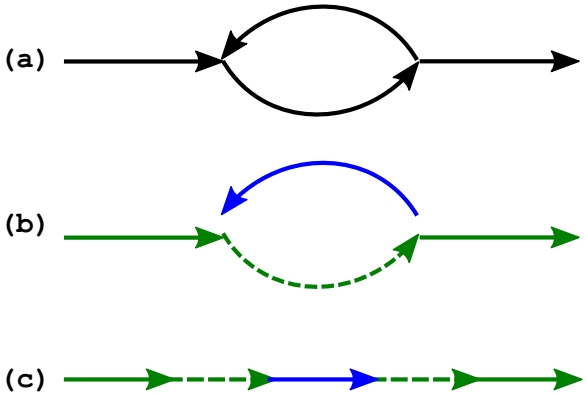

Figure 13. An example of how to solve cycles. (a) The original topology. (b) Paths derived from the matching. (c) The topology solved. Each arrow represents a path (a group of k-mers). Dashed arrow highlights the repeat area.

using k-mer graphs. Instead of having a distinct high-cost procedure to each problematic case in such kind of graphs, we overcome the explained difficulties using a unique strategy based on maximum matching on an equivalent bipartite graph (which is known to be a tractable problem) to find the smallest number of disjoint paths covering all nodes. These paths simplify the problem as they represent simpler components of the k-mer graph, allowing an easy identification and treatment of each problematic topology in such kind of graphs. In addition, we show that the proposed approach is very suitable for the application of GPU solutions.

This is important to observe two aspects of our method. First, alternative approaches to combine the obtained components, other than by identify the described topologies, are possible. Second, it is not necessary to explicitly represent the bipartite version of the k-mer graph to employ the presented solution. Notice that the double graph has twice as many vertices as the k-mer graph, which would mean a higher usage of memory.

We are currently implementing the described procedure using an alternative approach to combine the components obtained by the matching procedure. Our intention is to produce both a conventional Java program and a GPU solution.

Acknowledgment

This work is supported by FAPEMIG, CAPES, and CNPq.

References

[1] J. C. Venter *et al.*, "The sequence of the human genome," *Science*, vol. 291, no. 5507, pp. 1304–1351, 2001. [Online]. Available: http://www.sciencemag.org/content/291/5507/1304.abstract

[2] F. Sanger, S. Nicklen, and A. R. Coulson, "Dna sequencing with chain-terminating inhibitors," *Proceedings of the National Academy of Sciences*, vol. 74, no. 12, pp. 5463–5467, 1977. [Online]. Available: http://www.pnas.org/content/74/12/5463.abstract

[3] S. Fox, S. Filichkin, and T. Mockler, "Applications of ultra-high-throughput sequencing," in *Plant Systems Biology*, ser. Methods in Molecular Biology, D. A. Belostotsky, Ed. Humana Press, 2009, vol. 553, pp. 79–108. [Online]. Available: http://dx.doi.org/10.1007/978-1-60327-563-7_5

[4] R. A. Holt and S. J. Jones, "The new paradigm of flow cell sequencing," *Genome Research*, vol. 18, no. 6, pp. 839–846, 2008. [Online]. Available: http://genome.cshlp.org/content/18/6/839.abstract

[5] D. MacLean, J. D. G. Jones, and D. J. Studholme, "Application of 'next-generation' sequencing technologies to microbial genetics," *Nat Rev Micro*, vol. 7, no. 4, pp. 287–296, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1038/nrmicro2088

[6] E. R. Mardis, "Next-generation dna sequencing methods," *Annual Review of Genomics and Human Genetics*, vol. 9, no. 1, pp. 387–402, 2008, pMID: 18576944. [Online]. Available: http://www.annualreviews.org/doi/abs/10.1146/annurev.genom.9.081307.164359

[7] O. Morozova and M. A. Marra, "Applications of next-generation sequencing technologies in functional genomics," *Genomics*, vol. 92, no. 5, pp. 255 – 264, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0888754308001651

[8] M. L. Metzker, "Sequencing technologies [mdash] the next generation," *Nat Rev Genet*, vol. 11, no. 1, pp. 31–46, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1038/nrg2626

[9] J. Shendure, R. D. Mitra, C. Varma, and G. M. Church, "Advanced sequencing technologies: methods and goals," *Nat Rev Genet*, vol. 5, no. 5, pp. 335–344, May 2004. [Online]. Available: http://dx.doi.org/10.1038/nrg1325

[10] R. Li *et al.*, "De novo assembly of human genomes with massively parallel short read sequencing," *Genome Research*, vol. 20, no. 2, pp. 265–272, 2010. [Online]. Available: http://genome.cshlp.org/content/20/2/265.abstract

[11] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and n. Birol, "Abyss: A parallel assembler for short read sequence data," *Genome Research*, vol. 19, no. 6, pp. 1117–1123, 2009. [Online]. Available: http://genome.cshlp.org/content/19/6/1117.abstract

[12] M. C. Schatz, A. L. Delcher, and S. L. Salzberg, "Assembly of large genomes using second-generation sequencing," *Genome Research*, vol. 20, no. 9, pp. 1165–1173, 2010. [Online]. Available: http://genome.cshlp.org/content/20/9/1165.abstract

[13] K. Paszkiewicz and D. J. Studholme, "De novo assembly of short sequence reads," *Briefings in Bioinformatics*, vol. 11, no. 5, pp. 457–472, 2010. [Online]. Available: http://bib.oxfordjournals.org/content/11/5/457.abstract

[14] J. Shendure and H. Ji, "Next-generation dna sequencing," *Nat Biotech*, vol. 26, no. 10, pp. 1135–1145, Oct. 2008. [Online]. Available: http://dx.doi.org/10.1038/nbt1486

[15] M. Pop, "Genome assembly reborn: recent computational challenges," *Briefings in Bioinformatics*, vol. 10, no. 4, pp. 354–366, 2009. [Online]. Available: http://bib.oxfordjournals.org/content/10/4/354.abstract

[16] C. Kingsford, M. Schatz, and M. Pop, "Assembly complexity of prokaryotic genomes using short reads," *BMC Bioinformatics*, vol. 11, no. 1, p. 21, 2010. [Online]. Available: http://www.biomedcentral.com/1471-2105/11/21

[17] M. Baker, "De novo genome assembly: what every biologist should know," *Nat Meth*, vol. 9, no. 4, pp. 333–337, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1038/nmeth.1935

[18] W. Zhang, J. Chen, Y. Yang, Y. Tang, J. Shang, and B. Shen, "A practical comparison of *De Novo* genome assembly software tools for next-generation sequencing technologies," *PLoS ONE*, vol. 6, no. 3, p. e17915, 03 2011. [Online]. Available: http://dx.doi.org/10.1371%2Fjournal.pone.0017915

[19] D. Bryant, W.-K. Wong, and T. Mockler, "Qsra - a quality-value guided de novo short read assembler," *BMC Bioinformatics*, vol. 10, no. 1, p. 69, 2009. [Online]. Available: http://www.biomedcentral.com/1471-2105/10/69

[20] J. C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer, "Sharcgs, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing," *Genome Research*, vol. 17, no. 11, p. 000, 2007. [Online]. Available: http://genome.cshlp.org/content/early/2007/10/01/gr.6435207.abstract

[21] R. L. Warren, G. G. Sutton, S. J. M. Jones, and R. A. Holt, "Assembling millions of short dna sequences using ssake," *Bioinformatics*, vol. 23, no. 4, pp. 500–501, 2007. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/23/4/500.abstract

[22] J. R. Miller, S. Koren, and G. Sutton, "Assembly algorithms for next-generation sequencing data," *Genomics*, vol. 95, no. 6, pp. 315 – 327, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0888754310000492

[23] P. E. C. Compeau, P. A. Pevzner, and G. Tesler, "How to apply de Bruijn graphs to genome assembly." *Nature Biotechnology*, vol. 29, no. 11, pp. 987–91, Nov. 2011. [Online]. Available: http://www.nature.com/nbt/journal/v29/n11/full/nbt.2023.html

[24] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de bruijn graphs," *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008. [Online]. Available: http://genome.cshlp.org/content/18/5/821.abstract

[25] J. Butler, I. MacCallum, M. Kleber, I. A. Shlyakhter, M. K. Belmonte, E. S. Lander, C. Nusbaum, and D. B. Jaffe, "Allpaths: De novo assembly of whole-genome shotgun microreads," *Genome Research*, vol. 18, no. 5, pp. 810–820, 2008. [Online]. Available: http://genome.cshlp.org/content/18/5/810.abstract

[26] P. A. Pevzner, H. Tang, and M. S. Waterman, “An eulerian path approach to dna fragment assembly,” *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001. [Online]. Available: http://www.pnas.org/content/98/17/9748.abstract

[27] W. M. Thorburn, “The myth of occam’s razor,” *Mind*, vol. 27, no. 107, pp. pp. 345–353, 1918. [Online]. Available: http://www.jstor.org/stable/2248928

[28] F. R. Cerqueira, “Montagem de Fragmentos de DNA,” Master’s thesis, Instituto de Computação, Universidade Estadual de Campinas, 2000. [Online]. Available: http://cutter.unicamp.br/document/?code=vtls000195846

[29] F. R. Cerqueira and J. Meidanis, “Algorithms for Large-Scale DNA Sequencing,” in *SEMISH 2001, proceedings of the Brazilian Computer Society Congress*, 2001. [Online]. Available: http://200.169.53.89/download/CDcongressos/2001/SBC2001/pdf/arq0180.pdf

[30] H. W. Kuhn, “The hungarian method for the assignment problem,” *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955. [Online]. Available: http://dx.doi.org/10.1002/nav.3800020109

[31] J. Munkres, “Algorithms for the assignment and transportation problems,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/0105003