JAMERSON FELIPE PEREIRA LIMA

REPRESENTAÇÕES CACHE EFICIENTES PARA MONTAGEM DE FRAGMENTOS BASEADA EM GRAFOS DE *DE BRUIJN* DE SEQUÊNCIAS BIOLÓGICAS

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2017

Jamerson Felipe Pereira Lima

Representações cache eficientes para montagem de fragmentos baseada em grafos de *de Bruijn* de sequências biológicas

Este trabalho foi apresentado à Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: **Prof. Paulo Gustavo Soares da Fonseca**

RECIFE
2017

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

L732r Lima, Jamerson Felipe Pereira
Representações cache eficientes para montagem de fragmentos baseada em grafos de de Bruijn de sequências biológicas / Jamerson Felipe Pereira Lima. – 2017.
93 f.: il., fig., tab.

Orientador: Paulo Gustavo Soares da Fonseca.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências.

1. Ciência da computação. 2. Biologia computacional. I. Fonseca, Paulo Gustavo Soares (orientador). II. Título.

004 CDD (23. ed.) UFPE- MEI 2017-144

Jamerson Felipe Pereira Lima

Representações cache eficientes para montagem de fragmentos baseada em grafos de de Bruijn de sequências biológicas

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 20/02/2017.

BANCA EXAMINADORA

_____

Profa. Dra. Katia Silva Guimarães
Centro de Informática / UFPE

_____

Profa. Dra. Jeane Cecília Bezerra de Melo
Departamento de Estatística e Informática/UFRPE

_____

Prof. Dr. Paulo Gustavo Soares da Fonseca
Centro de Informática / UFPE
(**Orientador**)

À minha mãe, o farol desta nau que sonha com as águas dos mares mais distantes.

Agradecimentos

Agradeço ao meu país, o Brasil, por esta nova oportunidade de crescimento e acesso à educação. À UFPE, por ter sido a minha segunda casa durante estes dois anos, e ao Centro de Informática, pela possibilidade de uma pós-graduação de altíssimo nível, com uma estrutura excepcional. À FACEPE, pelo apoio financeiro que viabilizou minha dedicação à elaboração deste trabalho, sem o que ele não teria sido possível.

Ao meu orientador, Paulo, um ser humano excepcional, exemplo de inteligência, sabedoria, paciência, capacidade e responsabilidade. Eu não consigo expressar em palavras a minha gratidão e honra por tê-lo tido como parceiro nesta empreitada.

Às professoras Jeane Melo e Katia Guimarães, por terem aceito o convite para formar a banca da minha defesa, tendo feito contribuições de grande valia, e pelas quais nutro imenso respeito.

Aos meus amigos. Jeronimo e Leon, completando o time dos forasteiros do CIn, que acompanharam de perto as dores de cada dificuldade que enfrentei. Amaro, com quem compartilho as visões dos horizontes mais distantes. Alex, Joana, Jorge, Diego, Marcos, por existirem e me proporcionarem tantos momentos gratificantes. Roberto, uma pessoa iluminada com quem tenho a honra de conhecer tantas coisas diferentes. Aos tantos outros que eu tenho a sorte de compartilhar um suspiro nesta imensidão de tempo e espaço e que tornam melhores os meus dias. Vocês não imaginam o quanto cada um de vocês foi importante para que eu pudesse chegar ao fim dessa jornada.

À minha família. Aos meus irmãos Jonathan, Larissa, Letícia Marcelly, Geovanna, Letícia Helena, Renan, Guilherme, Raquel e Maria Cecília. Às minha avós Maria de Fátima e Iraci. Ao meu pai, Ribamar. Não imaginam quanto amo cada um de vocês e o quanto são importantes na minha vida, todos os dias.

Dois anos encarados à frente parecem duas décadas. Dois anos encarados às costas parecem dois meses. Embora agora a impressão seja de tudo ter passado rapidamente, não me falha à memória quanta coragem foi necessária para enfrentar cada novo desafio que se punha. Antes de Mestre, torno-me um ser humano mais maduro, alguém capaz de compreender e encarar melhor o futuro, sejam os próximos dois meses ou duas décadas.

Meu coração enche-se de orgulho por um resultado pelo qual tanto trabalhei. Novamente, sou imensamente grato a todos que acreditam no meu potencial e nos quais pude me apoiar durante esta árdua tarefa. Chegamos lá!

Este trabalho foi desenvolvido no âmbito do Projeto *Algoritmos e estruturas de dados cache-oblivious e aplicações à Biologia Computacional* (MCTI/CNPq/Universal 449842/2014-2, FACEPE APQ-0587-1.03/14).

Ubi dubium ibi libertas
(Onde há dúvida, há liberdade)
—PROVÉRBIO LATINO

Resumo

O estudo dos genomas dos seres vivos têm sido impulsionado pelos avanços na biotecnologia ocorridos desde a segunda metade do Séc. XX. Particularmente, o desenvolvimento de novas plataformas de sequenciamento de alto desempenho ocasionou a proliferação de dados brutos de fragmentos de sequências nucleicas. Todavia, a montagem dos fragmentos de DNA continua a ser uma das etapas computacionais mais desafiadoras, visto que a abordagem tradicional desse problema envolve a solução de problemas intratáveis sobre grafos obtidos a partir dos fragmentos, como, por exemplo, a determinação de caminhos hamiltonianos. Mais recentemente, soluções baseadas nos grafos de de Bruijn (gdB), também obtidos a partir dos fragmentos sequenciados, têm sido adotadas. Nesse caso, o problema da montagem relaciona-se com o de encontrar caminhos eulerianos, o qual possui soluções polinomiais conhecidas. Embora apresentem custo computacional teórico mais baixo, ainda demandam, na prática, grande poder computacional, face ao volume de dados envolvido. Por exemplo, a representação empregada por algumas ferramentas para o gdB do genoma humano pode alcançar centenas de *gigabytes*. Faz-se necessário, portanto, o emprego de técnicas algorítmicas para manipulação eficiente de dados em memória interna e externa. Nas arquiteturas computacionais modernas, a memória é organizada de forma hierárquica em camadas: cache, memória RAM, disco, rede, etc. À medida que o nível aumenta, cresce a capacidade de armazenagem, porém também o tempo de acesso. O ideal, portanto, seria manter a informação limitada o mais possível aos níveis inferiores, diminuindo a troca de dados entre níveis adjacentes. Para tal, uma das abordagens são os chamados algoritmos *cache-oblivious*, que têm por objetivo reduzir o número de trocas de dados entre a memória cache e a memória principal sem que seja necessário para tanto introduzir parâmetros relativos à configuração da memória ou instruções para a movimentação explícita de blocos de memória. Uma outra alternativa que vêm ganhando ímpeto mais recentemente é o emprego de estruturas de dados ditas sucintas, ou seja, estruturas que representam a informação usando uma quantidade ótima de bits do ponto de vista da teoria da informação. Neste trabalho, foram implementadas três representações para os gdB, com objetivo de avaliar seus

desempenhos em termos da utilização eficiente da memória cache. A primeira corresponde a uma implementação tradicional com listas de adjacências, usada como referência, a segunda é baseada em estruturas de dados *cache-oblivious*, originalmente descritas para percursos em grafos genéricos, e a terceira corresponde a uma representação sucinta específica para os gdB, com otimizações voltadas ao melhor uso da cache. O comportamento dessas representações foi avaliado quanto à quantidade de acessos à memória em dois algoritmos, nomeadamente o percurso em profundidade (DFS) e o *tour* euleriano. Os resultados experimentais indicam que as versões tradicional e *cache-oblivious* genérica apresentam, nessa ordem, os menores números absolutos de *cache misses* e menores tempos de execução para dados pouco volumosos. Entretanto, a versão sucinta apresenta melhor desempenho em termos relativos, considerando-se a proporção entre o número de *cache misses* e a quantidade de acessos à memória, sugerindo melhor desempenho geral em situações extremas de utilização de memória.

Palavras-chave: Grafos de de Bruijn. Algoritmos cache-oblivious. Estruturas de dados sucintas. Montagem de fragmentos. Tour euleriano.

Abstract

The study of genomes was boosted by advancements in biotechnology that took place since the second half of 20th century. In particular, the development of new high-throughput sequencing platforms induced the proliferation of nucleic sequences raw data. Although, DNA assembly, i.e., reconstitution of original DNA sequence from its fragments, is still one of the most computational challenging steps. Traditional approach to this problem concerns the solution of intractable problems over graphs that are built over the fragments, as the determination of Hamiltonian paths. More recently, new solutions based in the so called de Bruijn graphs, also built over the sequenced fragments, have been adopted. In this case, the assembly problem relates to finding Eulerian paths, for what polynomial solutions are known. However, those solutions, in spite of having a smaller computational cost, still demand a huge computational power in practice, given the big amount of data involved. For example, the representation employed by some assembly tools for a gdB of human genome may reach hundreds of gigabytes. Therefore, it is necessary to apply algorithmic techniques to efficiently manipulate data in internal and external memory. In modern computer architectures, memory is organized in hierarchical layers: cache, RAM, disc, network, etc. As the level grows, the storage capacity is also bigger, as is the access time (latency). That is, the speed of access is smaller. The aim is to keep information limited as much as possible in the highest levels of memory and reduce the need for block exchange between adjacent levels. For that, an approach are cache-oblivious algorithms, that try to reduce the the exchange of blocks between cache and main memory without knowing explicitly the physical parameters of the cache. Another alternative is the use of succinct data structures, that store an amount of data in space close to the minimum information-theoretical. In this work, three representations of the de Bruijn graph were implemented, aiming to assess their performances in terms of cache memory efficiency. The first implementation is based in a traditional traversal algorithm and representation for the de Bruijn graph using adjacency lists and is used as a reference. The second implementation is based in cache-oblivious algorithms originally described for traversal in general graphs. The third implementation is based in a

succinct representation of the de Bruijn graph, with optimization for cache memory usage. Those implementations were assessed in terms of number of accesses to cache memory in the execution of two algorithms, namely depth-first search (DFS) and Eulerian tour. Experimental results indicate that traditional and generic cache-oblivious representations show, in this order, the least absolute values in terms of number of cache misses and least times for small amount of data. However, the succinct representation shows a better performance in relative terms, when the proportion between number of cache misses and total number of access to memory is taken into account. This suggests that this representation could reach better performances in case of extreme usage of memory.

Keywords: De Bruijn graphs. Cache-oblivious algorithms. Succinct data structures. Fragment assembly. Eulerian tour.

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

BAC *Bacterial Artificial Chromosome* 22

BFS *Breadth First Search* 42

BPT *Buffered Priority Tree* 43

BRT *Buffered Repository Tree* 43

bp pares de bases 19

DFS *Depth First Search* 42

DNA Ácido Desoxirribonucleico 18

LRU *Least Recent Used*, ou Menos Recentemente Utilizado 28

MST *Minimum Spanning Tree* 42

NGS *Next-generation Sequencing* 21

OLC *overlap-layout-consensus*

PCR *Polymerase Chain Reaction* 21

RR *Random Replacement* 28

SSSP *Single-source Shortest Path* 42

vEB *van Emde Boas* 16

Lista de Algoritmos

Sumário

1

Introdução

O processamento massivo de dados em crescente disponibilidade, motivado pelo impulso no desenvolvimento de métodos computacionais, trouxe novos horizontes à pesquisa biológica. Embora o avanço tenha sido significativo, diversos problemas ainda representam um desafio, não somente sob o ponto de vista biológico, mas também sob o computacional.

Um dos avanços mais significativos foi obtido através do estudo do Ácido Desoxirribonucleico (DNA), que é uma molécula presente em todas as células dos seres vivos e que é responsável pela estrutura e funcionamento dos organismos, além da transmissão dos caracteres hereditários. O conhecimento da composição dessas moléculas, que coletivamente constituem o chamado *genoma* do indivíduo, possibilitou um estudo mais amplo e aprofundado do comportamento dos sistemas biológicos (PRIMROSE; TWYMAN, 2003).

A elucidação da composição do DNA dá-se através do processo de *sequenciamento*. Esse processo, contudo, pode demandar uma grande estrutura computacional, sobretudo face ao grande volume de dados produzido por algumas tecnologias de sequenciamento mais recentes. Em particular, uma das últimas etapas do processo consiste na solução de problemas computacionais complexos envolvendo estruturas de dados que podem facilmente exceder a capacidade de memória dos computadores disponíveis. O desenvolvimento de algoritmos e ferramentas computacionais neste campo é fundamental ao seu desenvolvimento.

O objetivo geral deste trabalho é explorar as arquiteturas de memória dos computadores modernos de modo a melhorar o desempenho de algoritmos utilizados no processo de sequencia-

mento. Atualmente, a maior parte das abordagens considera que o acesso à memória principal é direto, o que tipicamente não ocorre. Somente a memória *cache* é acessada diretamente pelo processador e a transferência de memória é feita por blocos. O uso eficaz deste modelo pode beneficiar o processamento de uma grande quantidade de dados, o que é comum na manipulação de dados biológicos.

1.1 Estrutura e composição do DNA

O código genético, formado pelo DNA, está presente no núcleo das células dos eucariontes e espalhado pelo citoplasma nos procariontes, bem como está presente em alguns vírus. O DNA é uma longa cadeia em formato de dupla hélice, formada por ligações de quatro diferentes moléculas, ou bases nitrogenadas, que são: Adenina (A), Citosina (C), Guanina (G) e Timina (T) (ALBERTS et al., 2014). A informação genética contida no DNA é traduzida em proteínas, as quais, por sua vez, determinam a estrutura e funcionamento dos organismos (NELSON; COX, 2008). Assim, seu estudo é fundamental para a compreensão dos processos biológicos que ocorrem nestes seres. O conjunto de todo o código genético de um ser é denominado **genoma**. Os trechos de DNA que codificam proteínas são denominados **genes**. Além destes, o DNA também é composto por trechos que não codificam proteínas, os quais têm função estrutural, de regulação ou não tem função conhecida.

A dupla hélice é formada por duas cadeias de DNA ligadas entre si, também chamadas fitas, cada uma composta por uma sequência linear de bases nitrogenadas. Devido a uma propriedade denominada paridade de bases, que ocorre por diferenças nas ligações químicas entre as bases. A Adenina de uma fita opõe-se à Timina da outra fita do DNA, assim como a Citosina opõe-se à Guanina, e vice-versa. Assim, as duas fitas do DNA são complementares, de modo que é possível determinar a partir de uma fita simples a sequência da fita oposta. Também por esse motivo, a unidade de medida de uma cadeia de DNA é pares de bases (bp), ou *base pair*, em Inglês.

As aplicações do estudo do DNA são inúmeras e podem envolver desenvolvimento de produtos na área de engenharia genética, como fármacos ou organismos geneticamente modificados, investigação forense, bioinformática, na extração, interpretação e manipulação massiva de

informação biológica, estudo da evolução, dado que a informação genética é transmitida dos ancestrais aos seus descendentes, entre diversas outras possibilidades.

1.2 Sequenciamento de DNA

O estudo do DNA, iniciado na primeira metade do Século XX, experimentou grandes avanços desde então. O processo de obtenção desse DNA envolve algumas etapas, entre as quais a obtenção da amostra biológica, o sequenciamento do DNA coletado e sua aplicação a fins biológicos.

O sequenciamento consiste na determinação exata da sequência de bases nitrogenadas em uma cadeia de DNA. Isso é necessário já que a composição do DNA está diretamente ligada às proteínas que são codificadas. Além disso, no estudo da genômica comparativa, por exemplo, é possível observar diferenças funcionais e evolutivas a partir da diferença entre as cadeias de DNA.

Com o desenvolvimento das ferramentas de manipulação de genes *in vitro*, a comunidade científica passou a questionar-se qual seria o custo, o retorno e a melhor maneira de realizar o sequenciamento de um genoma completo de um ser vivo. Concluiu-se que a melhor abordagem seria quebrá-lo em pedaços menores e depois reconstruí-los. Portanto, após o sequenciamento dos fragmentos individualmente, deve haver uma etapa na qual estes trechos são recombinados de modo a reconstruir a cadeia de DNA original, i.e., uma etapa de **montagem**.

1.2.1 Tecnologias de Sequenciamento

Um dos primeiros passos para a obtenção das cadeias de DNA dos seres vivos foi a utilização do sequenciamento Sanger, o qual é baseado em técnicas de eletroforese (SHENDURE; JI, 2008). Desde o primeiro genoma sequenciado, de um bacteriófago (SANGER; NICKLEN; COULSON, 1977), diversas tecnologias de sequenciamento foram exploradas. As primeiras tecnologias, conhecidas como *First-generation Sequencing*, baseavam-se em uma técnica denominada *chain termination*, ou terminação de cadeia. Entre estes, o método Sanger popularizou-se por sua eficiência e foi adotado como uma tecnologia padrão para sequenciamento durante cerca

de 25 anos. Implementações modernas dessa técnica produzem fragmentos de entre 1000 e 5000 pares de bases.

Após anos de aperfeiçoamento, diversos avanços aumentaram a capacidade de sequenciamento e a qualidade dos resultados. Uma das principais melhoras foi a automação do sequenciamento baseado em capilares, o que aumentou o paralelismo do método (LIU et al., 2012). Desse modo, o método Sanger tornou-se uma das principais ferramentas para a conclusão do Projeto Genoma Humano.

No século XXI, com a maior disponibilidade de poder computacional e necessidade de mais informação de modo a aumentar a acurácia dos métodos de sequenciamento, uma série de tecnologias mais avançadas de sequenciamento, denominadas *Next-generation Sequencing* (NGS), também denominadas tecnologias de segunda geração, emergiram (MARDIS, 2008). A principal vantagem em relação aos sistemas de sequenciamento mais antigos é a capacidade de paralelismo, com alto *throughput* [1], e o custo muito mais baixo para o sequenciamento de cada base. Embora, assim como ocorre no sequenciamento Sanger, o DNA seja dividido em trechos, nas NGSs a quantidade de *reads* [2], i.e., fragmentos do DNA, gerada é muito maior à medida que o tamanho de cada *read* pode ser de entre 35 e 700 pares de bases, a depender da tecnologia (LIU et al., 2012).

As tecnologias de sequenciamento NGS podem variar entre si também quanto ao tamanho de cada *read* gerado e ao tipo de erro mais comum associado, como substituição, eliminação ou inserção. Uma desvantagem das NGSs em relação ao sequenciamento Sanger, é o tamanho reduzido dos fragmentos obtidos, o que pode diminuir a qualidade do resultado obtido na montagem e aumentar a taxa de erro. Uma outra desvantagem é utilização da *Polymerase Chain Reaction* (PCR) para amplificação das amostras a serem sequenciadas, que pode ser demorada e tecnicamente complexa. A velocidade de sequenciamento, entretanto, é muito maior, dado o alto grau de paralelismo. A tendência é que, com seu amadurecimento, as desvantagens sejam superadas, assim como ocorreu com o sequenciamento Sanger, que passou por avanços durante 25 anos para alcançar a qualidade técnica que tem na atualidade.

A grande quantidade de informação gerada pelos métodos de sequenciamento de segunda

[1] Capacidade de transferência.
[2] O termo *read* é utilizado como equivalente a um fragmento de DNA.

geração trouxe novos desafios, haja vista o número de *reads* que podem ser gerados durante o processo e o número de possibilidades de comparação para a procura de trechos coincidentes, que é uma abordagem comum utilizada para a montagem dos fragmentos. Além disso, ainda são necessárias outras fases, como a correção de erros nos fragmentos, causados por falhas inerentes à coleta do DNA, a obtenção da cadeia consenso, de modo a resolver trechos de ambiguidade, entre outros. Assim, o processo de sequenciamento pode demandar grande capacidade de armazenamento e processamento.

1.2.2 Estratégias de sequenciamento

Existem duas estratégias de sequenciamento de um genoma: as *clone libraries*, ou "bibliotecas" de clones, e o *whole genome shotgun*. A primeira foi a estratégia de escolha durante vários anos, tendo sido o método inicialmente escolhido no projeto do consórcio internacional para o sequenciamento do genoma humano (LI et al., 2012). A estratégia para a constituição das *clone libraries* consiste em dividir o genoma em fragmentos de um tamanho grande, cerca de 1000 bp a 5000 bp, denominados clones. Essa divisão é realizada por uma enzima de restrição que, no momento em que divide os fragmentos de DNA, realiza uma marcação que torna possível o mapeamento desses trechos, ordenando-os quanto à sua posição original no genoma. Desse modo, o sequenciamento é feito clone a clone, o que dá nome ao método. Cada clone é então dividido em fragmentos menores, os quais são multiplicados com o uso de um vetor biológico, e.g., um *Bacterial Artificial Chromosome* (BAC), que é baseado num plasmídeo e é capaz de replicar-se. Os fragmentos são então isolados, sequenciados e montados. Os clones são reposicionados ordenadamente de modo a reconstituir o genoma. Essa estratégia diminui o impacto de trechos repetitivos no genoma e é bastante útil quando o objetivo é realizar anotação genômica. Entretanto, as desvantagens desta abordagem são o custo e tempo necessários, visto que é preciso manipular cada clone individualmente.

A estratégia *whole genome shotgun* (STADEN, 1979), por sua vez não possui uma etapa de mapeamento de trechos do DNA, realizando a quebra e montagem de trechos de todo o genoma de uma só vez. Essa abordagem funciona bem para genomas microbianos, os quais são tipicamente menores e possuem menos repetições. A perda de informação no nível do

genoma, visto que os fragmentos normalmente têm um tamanho muito menor, entre 35 e 500 bp (LIU et al., 2012), ainda inviabiliza o uso desta abordagem isoladamente para genomas com grande tamanho, como os mamíferos, por exemplo. Entretanto, com o aumento da capacidade de processamento computacional, o uso de uma quantidade crescente de informação reduz a influência dessas desvantagens.

1.2.3 Montagem do DNA

Visto que o resultado do sequenciamento são fragmentos do DNA, a etapa seguinte consiste no alinhamento desses *reads* entre si, avaliando a ocorrência de sobreposições, de modo a obter trechos maiores e, em última instância, o genoma original. Na Figura 1.1 é mostrado um exemplo de sobreposição entre dois fragmentos. Diz-se que ocorre um alinhamento entre dois fragmentos A e B, com tamanho a e b, respectivamente, se, dado um valor l que representa a tamanho mínimo de sobreposição, o sufixo de tamanho m de A é igual ao prefixo de tamanho m de B, tal que $m > l$. O alinhamento de dois fragmentos resulta na concatenação do sufixo de tamanho $a - m$ de A, da sobreposição de tamanho m e do sufixo de tamanho $b - m$ de B. A montagem de um conjunto de fragmentos consiste no mapeamento das sobreposições entre esses fragmentos para uma reconstrução putativa da sequência de objetivo. Dado que o alinhamento de uma quantidade grande de fragmentos de DNA, normalmente centenas de milhões ou bilhões, é um problema complexo, diversas abordagens para este problema já foram desenvolvidas (MILLER; KOREN; SUTTON, 2010). As mais utilizadas, a *overlap-layout-consensus* e os grafos de *de Bruijn*, são descritas a seguir.

Figura 1.1: Exemplo de sobreposição entre dois fragmentos.

1.2.3.1 *Overlap-layout-consensus*

O método de montagem *overlap-layout-consensus* (OLC) é composto por três etapas, como indica o nome, sendo a primeira, *overlap*, relativa à sobreposição dos fragmentos. Nesta

etapa, as busca de trechos de sobreposição entre dois *reads* é explícita, i.e., a abordagem é todos-contra-todos, ou seja, verifica-se a cada par de fragmentos a ocorrência de alinhamento entre prefixos e sufixos. O resultado é um grafo de sobreposições, no qual cada vértice corresponde a um dos fragmentos de DNA sequenciados e uma aresta dirigida entre dois vértices representa a ocorrência de um alinhamento do sufixo do vértice de origem com o prefixo do vértice de destino, caso o tamanho da sobreposição seja maior que um valor de limite, determinado previamente (LI et al., 2012).

Na etapa de *layout*, o objetivo é simplificação do grafo de sobreposições, removendo-se arestas redundantes do grafo e criar regiões de *contigs*, em que há percursos sem bifurcações. A etapa de *consensus* consiste na determinação dos caminhos mais prováveis em trechos do grafo de sobreposições em que um vértice possui arestas de saída para vizinhos distintos. A obtenção de um alinhamento entre todos os *reads* presentes no grafo de sobreposições é dada por um caminho que passa por cada vértice exatamente uma vez, i.e., um caminho hamiltoniano. Este problema, entretanto, é provado ser NP-completo, i.e., não possui uma solução conhecida de de tempo polinomial (KARP, 1972). Na etapa seguinte, de consenso, o objetivo é encontrar uma solução de consenso para a cadeia final, caso seja encontrada mais de um caminho hamiltoniano. Nas Figuras 1.2 e 1.3 são mostrados, respectivamente, um conjunto de fragmentos e o grafo OLC construído a partir desses fragmentos, com um $l = 6$. Na Figura 1.3 observa-se um possível caminho hamiltoniano que representa um alinhamento dos fragmentos.

AGCTAGCTAGCTGA AGCTGACCCGGATCGATAGC GGCTGTAGTCGAGCGTGAGGCTTT

AGCTGATCGTAGCAGCTAG GGCTTTGCGATGCCTAGCTGC

Figura 1.2: Exemplo de fragmentos de DNA.

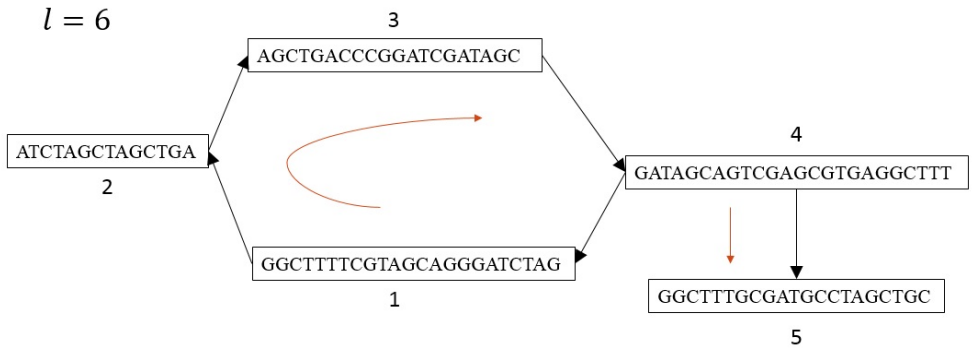


Figura 1.3: Grafo OLC obtido a partir dos fragmentos da Figura 1.2.

Uma vantagem deste método é que admite-se alguns erros no alinhamento, o que reduz o impacto de erros de sequenciamento. Assim, quando o grafo de alinhamentos é criado, não são criados caminhos artificiais devido a esses erros. Além disso, com a redução de custo das tecnologias de sequenciamento e o consequente aumento dos fragmentos lidos, esta abordagem pode beneficiar-se de *reads* maiores. A etapa de *overlap* pode ser bastante custosa do ponto de vista computacional, visto que são feitas comparações entre todos os fragmentos. Além disso, a informação sobre repetições pode se perder nesta etapa. Outro problema é o fato de que não existe solução exata para o problema do caminho hamiltoniano, o que demanda soluções aproximadas e pode não ser eficiente em alguns casos.

O *overlap-layout-consensus* foi o método de escolha até o surgimento de outra abordagem, com grafos de *de Bruijn*. Contudo, ainda nos dias atuais, com a melhora no tratamento de regiões de repetições, é um método frequentemente utilizado.

1.2.3.2 Grafos de *de Bruijn*

Uma das soluções computacionais utilizadas para a montagem de DNA são estruturas de dados denominadas grafos de *de Bruijn* (BRUIJN, 1946). Contrariando a intuição, os *reads* são reduzidos a fragmentos de um dado tamanho k, denominados *k-mers*. O valor de k deve ser menor que todos os fragmentos do conjunto e é determinado de acordo com o tipo de abordagem, situando-se normalmente entre 20 e 40. Os nós são compostos pelas *k-mers* e uma aresta existe entre os dois nós se ocorre um alinhamento de $k-1$ bases (PEVZNER; TANG; WATERMAN, 2001). Abordagens com esse tipo de estrutura normalmente utilizam algoritmos de busca de um *tour* euleriano, que possui solução em tempo polinomial, o que representa um avanço em relação

à abordagem *overlap-layout-consensus*.

Os grafos de *de Bruijn* são menos tolerantes a erros de sequenciamento que o *overlap-layout-consensus*, visto que um erro pode criar uma nova *k-mer* e, consequentemente, um caminho euleriano diferente do que ocorreria no genoma original, já que não ocorre o casamento dos fragmentos na região onde o erro acontece. A montagem de genomas com muitas regiões de repetição é também dificultada nesse caso, com resultados geralmente menos eficazes quando comparado com o *overlap-layout-consensus*. Além disso a cobertura, isto é, o número de fragmentos que representam uma base de determinada posição na sequência de referência, também influencia a escolha do método de montagem. Considerando o consumo de memória e processamento, o OLC é mais adequado para montagem de fragmentos maiores e baixa cobertura, à medida que os grafos de *de Bruijn* são mais adequados para fragmentos curtos e maior cobertura (LI et al., 2012).

O número de nós no grafo de *de Bruijn* é usualmente próximo ao tamanho do genoma, assim, na montagem do genoma humano, por exemplo, são necessários até 3×10^9 nós. Dado isto, comumente não é possível manter em memória principal toda a estrutura de dados em um dado momento. Isto pode ser agravado pelo fato de que, na arquitetura dos computadores modernos, o processador não endereça diretamente a memória principal, mas a memória *cache*, a qual, embora tenha um tempo de acesso menor que a memória principal, tipicamente tem um tamanho que pode ser várias ordens de grandeza menor que a memória principal e a memória secundária. Assim, a arquitetura de memória é um determinante no tempo necessário ao processamento do grafo de subsequências. Uma opção é o uso de abordagens que se utilizem dessas arquiteturas para reduzir o tempo de processamento. Existem duas abordagens para um maior aproveitamento da memória: a primeira são algoritmos *cache-oblivious*, que utilizam uma arquitetura de memória cache de maneira ótima, sem conhecimento sobre a arquitetura de memória, e a segunda é o uso de estruturas de dados que ocupem menos espaço, como ocorre com as estruturas de dados sucintas. Na Seção 1.3 é descrita essa abordagem.

1.3 Algoritmos *Cache-oblivious*

Os computadores modernos tem uma estrutura normalmente baseada na arquitetura de *von Neumann*. Esta arquitetura é composta por três elementos: uma CPU, responsável pelo processamento dos dados e um dispositivo de armazenamento massivo de memória e mecanismos de entrada e saída. A comunicação entre a CPU e a memória ocorre através de um barramento. Em uma linguagem de programação compatível com esse modelo, uma memória teoricamente infinita é acessada aleatoriamente e em tempo constante, o que também é chamado modelo de memória com acesso aleatório, ou *random access memory* (RAM), mostrado na figura Figura 1.4.

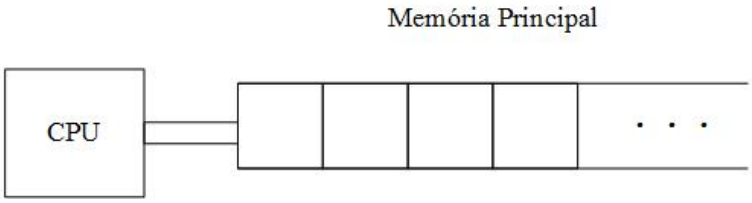


Figura 1.4: Modelo RAM.

Em uma arquitetura real, entretanto, ocorrem alguns problemas, entre os quais a diferença existente entre o tempo de acesso à memória e o tempo de processamento na CPU, que pode ser de algumas ordens de grandeza. Isto ocorre quando a CPU termina uma certa computação e necessita recuperar ou gravar informação na memória, fazendo com que fique ociosa e seja desperdiçado tempo de processamento.

Como solução, foram desenvolvidos modelos de memória hierárquicos com **cache**. Assim, o processador passa a comunicar-se com uma memória intermediária com maior velocidade de acesso e menor tamanho, armazenando os trechos da memória principal com maior probabilidade de serem acessados. Desse modo, é possível desenvolver sistemas com memória de grande capacidade, mas que pode ser acessada rapidamente, já que uma arquitetura com memória cache é capaz de reduzir o tempo de espera da CPU em até 50%. Numa arquitetura hierárquica, os espaços de memória são manipulados em blocos de tamanho B entre os níveis, isto é, regiões contíguas da memória compostas por várias palavras de memória. Isto ocorre porque o custo de manipulação de um bloco é aproximadamente o mesmo de manipular um único espaço de memória (ENGLANDER, 2009).

Ao requisitar um bloco à memória, a CPU inicialmente verifica a memória cache. Caso o bloco de memória encontre-se armazenado na cache, ocorre um *cache hit*, ou acerto de cache. Caso contrário, ocorre um *cache miss* e o bloco é recuperado na memória principal e armazenado na cache. Se a cache está cheia, então um dos blocos armazenados é escolhido por um algoritmo de substituição pré-determinado e substituído pelo novo. Uma cache com comportamento ótimo é onisciente, já que seria possível prever o comportamento da CPU e a ordem em que os blocos serão usados futuramente. Visto que esse comportamento não é possível na prática, a literatura dispõe de uma série de algoritmos de substituição, entre os quais o *Least Recent Used*, ou Menos Recentemente Utilizado (LRU) e o *Random Replacement* (RR) (TANENBAUM; BOS, 2014). A porcentagem de *cache hits* em relação ao total de consultas à memória é chamado *hit ratio* e funciona com uma importante medida de desempenho do sistema (ENGLANDER, 2009).

Embora essas hierarquias de memória tenham sido aplicadas, os programas desenvolvidos para essas máquinas ainda são baseados no modelo RAM, i.e., utilizam a memória como se fosse de acesso aleatório, o que pode reduzir o desempenho do programa, visto que a subutilização da memória cache e o acesso constante à memória principal aumentam o tempo ocioso da CPU. Esse fato motivou o desenvolvimento de modelos computacionais de memória externa, entre os quais, alguns que consideram a memória cache, também chamados cache-eficientes. Nesse modelo, a análise de um algoritmo é feita em termos de M, o tamanho da memória cache, e B, o tamanho do bloco de memória. Desse modo, a cache é capaz de conter M/B blocos de memória, como mostrado na Figura 1.5.

Uma abordagem cache-eficiente são os algoritmos *cache-oblivious*[3], ou cache-informados, pode ser dificultada pela diversidade do *hardware*, definidos por FRIGO et al. (1999), os quais são capazes de atingir um resultado ótimo quanto ao número de *cache misses*, sem que seja necessário fornecer informações sobre a estrutura física da arquitetura de memória na qual o algoritmo será executado. Assim, é possível que um algoritmo *cache-oblivious* tenha o mesmo desempenho apresentado por modelos que manipulam explicitamente a memória. Além disso, é possível que esse algoritmo execute essencialmente em qualquer arquitetura de memória com uma sobrecarga mínima, o que também melhora a portabilidade do algoritmo (FRIGO et al.,

[3]Os termos *cache-aware* e *cache-oblivious* são utilizados em lugar de suas traduções na Língua Portuguesa por falta de uma tradução consagrada.

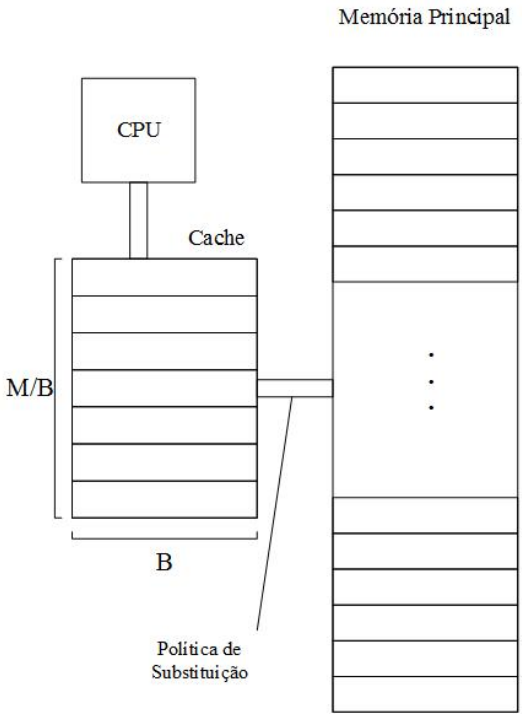


Figura 1.5: Arquitetura de memória com cache.

1999; ARGE et al., 2007).

Um exemplo simples de um algoritmo *cache-oblivious* é a inversão de um *array*. O primeiro passo para isto é garantir que a leitura de um *array* também é *cache-oblivious*. Numa leitura dos elementos desse *array* em ordem, caso o *array* tenha N elementos, manipulados em blocos com tamanho B, serão necessárias $\lceil N/B \rceil$ transferências de blocos entre dois níveis de memória. (DEMAINE, 2002).

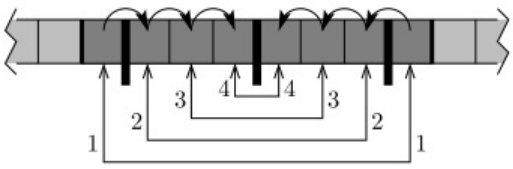


Figura 1.6: Algoritmo definido por BENTLEY (2000) para inversão *cache oblivious* de um array.

BENTLEY (2000) definiu uma estratégia *cache-oblivious* com um custo semelhante para inverter os elementos de um *array*, mostrada na figura Figura 1.6. A abordagem consiste em realizar a leitura simultânea a partir de ambas as extremidades do *array*, trocando as posições de cada par de elementos considerados a cada vez. Esse algoritmo incorre no mesmo número de leituras de memória que a simples leitura do *array* e tem uma complexidade semelhante a um

algoritmo equivalente em um modelo de memória externa.

O projeto de algoritmos *cache-oblivious* normalmente é baseado em técnicas e abordagens já definidas como *cache-oblivious*, como leituras de *array* e divisão e conquista. Esta abordagem de divisão e conquista, entretanto, nem sempre é a mais adequada, de maneira que técnicas mais sofisticadas são necessárias para alcançar um desempenho ótimo de cache. Na literatura estão estabelecidos uma série de exemplos, como algoritmos de ordenação, multiplicação de matrizes, busca binária, algoritmos para grafos, entre outros.

Um algoritmo cache-eficiente para ter um desempenho ótimo no uso da cache, pode ser desenvolvido tomando como entrada parâmetros das características físicas da memória cache, como o algoritmo de substituição utilizado, M, B, entre outros. Com essa informação, o programa tenta otimizar o uso dos blocos da memória, de modo a reduzir a quantidade de *cache misses*. Entretanto, a implementação desses algoritmos, também chamados *cache-aware* encontrada no mercado e a necessidade de fornecer na entrada do programa essa informação, que também pode ser de difícil obtenção.

1.4 Objetivos da Dissertação

O objetivo geral desta dissertação é estudar métodos cache-eficientes para montagem de DNA em grafos de *de Bruijn*. Esse estudo será realizado através da implementação de três alternativas para a representação do grafo de *de Bruijn* e posterior avaliação experimental dessas versões quanto ao desempenho de memória. Essa avaliação ocorre tanto em termos absolutos, da quantidade de *cache misses* causada pela execução do algoritmo, como relativos, com o *miss ratio* de cada versão. O Capítulo 2 explora soluções existentes relativas ao sequenciamento e algoritmos ou estruturas de dados que possam ser utilizadas como ferramenta para reduzir a sobrecarga no processamento de um grafo de *de Bruijn* durante a montagem. O Capítulo 3 versa sobre o desenvolvimento da proposta feita neste trabalho e a metodologia e materiais utilizados. O Capítulo 4 discute os resultados obtidos com a proposta desenvolvida. O Capítulo 5 traz as conclusões em relação aos resultados e perspectivas possíveis, considerando o que foi desenvolvido e o estado da arte.

2

Fundamentação Teórica

Visto que as bases que formam o DNA podem ser representadas como caracteres num computador, o desafio posto pela montagem de fragmentos gerados no sequenciamento do DNA motivou a exploração de soluções no campo dos algoritmos de cadeias de caracteres. Tais soluções podem diferir substancialmente entre si, como a *overlap-layout-consensus* e os grafos de *de Bruijn*.

2.1 Planejamento da Revisão Sistemática

De maneira a avaliar do estado da arte em relação ao proposto no trabalho, foi realizada uma revisão sistemática da literatura. Para tanto, foi estabelecido um protocolo de revisão sistemática, no qual são definidas a pergunta de pesquisa, os termos-chave, o método de busca, as fontes de busca, os critérios de inclusão e exclusão, entre outros fatores. Estes elementos são descritos a seguir nesta seção.

2.1.1 Pergunta de Pesquisa

É possível melhorar o desempenho de memória no *tour* euleriano de um grafo de *de Bruijn* utilizando abordagens eficientes em termos de memória?

2.1.1.1 Subpergunta

Algoritmos e estruturas de dados *cache-oblivious* ou sucintas são capazes de melhorar o desempenho, em termos de memória, da execução de um algoritmo de percurso em um grafo de *de Bruijn*?

2.1.2 Termos-chave da Pesquisa

A partir da pergunta de pesquisa são derivados os termos-chave da busca realizada na revisão sistemática. Estes termos são:

- *cache-oblivious*
- estruturas de dados sucintas
- grafo de *de Bruijn*
- DFS
- euleriano
- montagem de dna

A *string* de busca desenvolvida a partir desses termos chave foi a seguinte:

("cache oblivious"AND "graph*"AND ("dfs"OR "euler")) OR ("succinct"AND "de bruijn graph*") OR ("cache oblivious"AND "de bruijn graph*") OR (("dna"OR "assembly") AND "de bruijn graph*"AND "cache oblivious") OR (("dna"OR "assembly") AND "de bruijn graph*"AND "succinct") OR ("succint"AND "graph*"AND ("dfs"OR "euler"))

O objetivo é identificar descrições teóricas ou implementações de versões cache-oblivious ou sucintas para percurso em grafos arbitrários ou de *de Bruijn*. As fontes de busca selecionadas foram:

- **ACM** <http://dl.acm.org/>
- **PubMed** <https://www.ncbi.nlm.nih.gov/pubmed>
- **SIAM** <https://www.siam.org/>

- **Springer** <https://link.springer.com/>

Nos motores de busca enumerados foram encontrados 107 resultados no total para a *string* de busca desenvolvida. A partir desses resultados, foi efetuada um etapa de seleção, de maneira a elucidar quais desses estudos são relevantes para a pesquisa. Os critérios de inclusão foram:

- trabalhos que respondam as questões de pesquisa;
- estudos que apresentem técnicas para o desenvolvimento de abordagens *cache-oblivious* ou sucinta para percurso em um grafo de *de Bruijn*;
- estudos que tenham descrições detalhadas para o desenvolvimento de abordagens *cache-oblivious* ou sucinta para a execução de um *tour* euleriano em um grafo de *de Bruijn*.

Os critérios de exclusão foram:

- trabalhos notadamente irrelevantes à pesquisa, de acordo com as perguntas de pesquisa elaboradas;
- trabalhos duplicados;
- trabalho incompletos;
- trabalhos que não estejam escritos em Língua Inglesa;
- trabalho não publicados entre Janeiro/1999 e Junho/2016;
- resumos;
- trabalhos incompletos.

Após a fase de aplicação dos critérios de inclusão e exclusão, alguns trabalhos foram selecionados como relevantes à pesquisa, como mostrado na Tabela 2.1.

Trabalho	Autor
An Optimal Cache-Oblivious Priority Queue and Its Application to Graph Algorithms	(ARGE et al., 2007)
Succinct de Bruijn Graphs	(BOWE et al., 2012)
Space-efficient and exact de Bruijn graph representation based on a Bloom filter	(CHIKHI; RIZK, 2012)
On the Representation of de Bruijn Graphs	(CHIKHI et al., 2014)
Efficient Construction of a Compressed de Bruijn Graph for Pan-Genome Analysis	(BELLER; OHLEBUSCH, 2015)
Bidirectional Variable-Order de Bruijn Graphs	(BELAZZOUGUI et al., 2016)
Using cascading Bloom filters to improve the memory usage for de Brujin graphs	(SALIKHOV; SACOMOTO; KUCHEROV, 2014)

Tabela 2.1: Trabalhos selecionados no mapeamento sistemático.

2.1.3 Extração dos Dados

Visto que o objetivo deste trabalho é avaliar o comportamento em memória de representações alternativas de grafos de *de Bruijn* para a montagem de DNA, alguns dos trabalhos obtidos no mapeamento sistemático foram selecionados para servirem como referência às implementações a serem feitas. A única representação *cache-oblivious* entre os trabalhos selecionadas é a definição do algoritmo de DFS encontrada em ARGE et al. (2007), pelo que foi esta a opção utilizada para avaliação da aplicação desse conceito aos grafos de *de Bruijn*. Em termos de estruturas que têm como objetivo reduzir o tamanho necessário ao armazenamento das estruturas de dados, foi escolhida a abordagem sucinta descrita por BOWE et al. (2012), em detrimento às demais representações baseadas em compressão, como em BELLER; OHLEBUSCH (2015), e filtros de *bloom*, como em CHIKHI; RIZK (2012), CHIKHI et al. (2014) e SALIKHOV; SACOMOTO; KUCHEROV (2014). A compressão das estruturas pode adicionar alguma complexidade ao problema, assim com ocorre com os filtros de *bloom*, visto que estas estruturas têm um elemento probabilístico. Esse tipo de estruturas podem gerar discussões que não são foco deste trabalho. Adicionalmente, embora também apresente uma abordagem sucinta, BELAZZOUGUI et al. (2016) representa grafos bidirecionais e com valor de k variável, o que não é objeto de estudo neste trabalho.

Em seguida são descritos os conceitos básicos necessários à compreensão da implementação realizada e as abordagens propriamente ditas, com detalhes sobre as estruturas utilizadas.

2.2 Grafos de *de Bruijn*

O primeiro método para montagem de sequências biológicas utilizando os grafos de de Bruijn foi apresentado por PEVZNER; TANG; WATERMAN (2001), que propuseram o EULER, um algoritmo que realiza o alinhamento dos fragmentos obtidos em um processo de sequenciamento, através de um *tour* euleriano sobre o gdB que os representa.

Definição 2.1. Um grafo de *de Bruijn* $G = (V,E)$ k-dimensional de uma *string* $S = s_1 s_2 \dots s_n$, com alfabeto Σ e tamanho σ, é um grafo tal que $V = \{s_i \dots s_{i+k-1}; 1 \leq i \leq n-k+1\}$ e $E = \{(s_i \dots s_{i+k-1}, s_{i+1} \dots s_{i+k}); 1 \leq i \leq n-k\}$.

Essa definição difere da original proposta por BRUIJN (1946), a qual representa nos vértices todo o conjunto $U = \Sigma^k$ k-mers. Assim, $V \subseteq U$, já que não necessariamente todas as *strings* de tamanho k ocorrem em S. Além disso, a definição de PEVZNER; TANG; WATERMAN (2001) admite arestas múltiplas.

Existe uma interpretação natural de um caminho num grafo de *de Bruijn*. Se dois vértices u e v associados a rótulos de tamanho k são adjacentes, uma aresta $(u,v) \in E$, que é um caminho de tamanho 1, consiste na sequência de tamanho $(k+1)$ resultante do alinhamento desses rótulos. Um caminho de tamanho m consiste na sequência de tamanho $(k+m)$, resultante do alinhamento dos rótulos dos vértices nesse caminho. Com um *tour* euleriano em G é possível obter a sequência S (IDURY; WATERMAN, 1995). Para a existência de *tour* euleriano no grafo, algumas condições devem ser atendidas.

Definição 2.2. Um *tour* euleriano em um grafo dirigido $G = (V,E)$ visita cada aresta de E exatamente uma vez.

Teorema 2.1. *Um multigrafo conexo dirigido $G = (V,E)$ é euleriano se, e somente se, todo vértice tem grau par.*

Corolário 2.1. *Um multigrafo conexo dirigido $G = (V,E)$ é semi-euleriano se, e somente se, existem exatamente dois vértices grau ímpar.*

Na Figura 2.1 é mostrado um exemplo de um grafo de *de Bruijn* 4-dimensional para a cadeia S =ACGACGACTGAC. S pode ser obtida com um *tour* euleriano ACGA $\rightarrow$ CGAC $\rightarrow$ GACG $\rightarrow$ ACGA $\rightarrow$ CGAC $\rightarrow$ GACT $\rightarrow$ ACTG $\rightarrow$ CTGA$\rightarrow$ TGAC.

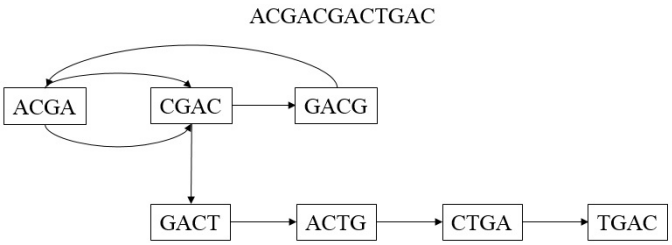


Figura 2.1: Um grafo de *de Bruijn* 4-dimensional para a cadeia ACGACGACTGAC.

Cada *tour* euleriano distinto no grafo corresponde a uma sequência distinta. Somente pode-se garantir que um *tour* euleriano em G é não-ambíguo se, e somente se, o número de *tours* eulerianos em G é exatamente um (IDURY; WATERMAN, 1995). Na Figura 2.1, é mostrado um exemplo. O vértice v = CGAC ocorre duas vezes no *tour* euleriano. No *tour* euleriano que devolve a sequência correta, a aresta CGAC$\rightarrow$ GACG é escolhida antes da aresta CGAC$\rightarrow$ GACT. Caso contrário, a sequência resultante é ACGACTGACGAC. Um valor de k maior reduziria a probabilidade de ocorrência de uma dada k-mer em S, entretanto, o impacto é pequeno ou nulo a partir de um certo limite (CHAISSON; BRINZA; PEVZNER, 2009).

2.2.1 Problemas na abordagem com grafos de *de Bruijn*

Regiões repetitivas são o problema de solução mais difícil num grafo de *de Bruijn*. Uma repetição é definida como um caminho $v_1 \ldots v_n$ tal que $grau_{entrada}(v_1) > 1$, $grau_{saida}(v_n) > 1$ e $grau_{entrada}(v_i) = grau_{saida}(v_i)$ para $1 \leq i \leq n-1$, de modo que uma repetição possui diversas entradas e saídas. A Figura 2.2 mostra um esquema de uma repetição. Devido à perda de informação que ocorre na fragmentação do DNA, não sabe-se a ordem em que essas entradas e saídas são visitadas, o que pode ter difícil solução e causar fragmentação do genoma sequenciado. Na Figura 2.3 as arestas codificam a sequência AAAAACAAAAAAAA, entretanto, com a fragmentação, não é possível saber a ordem das visitas ao vértice AAAA.

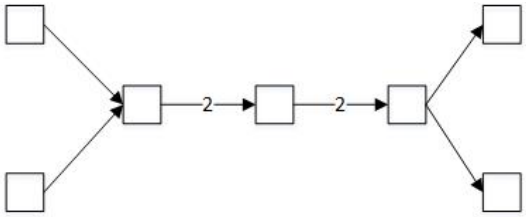


Figura 2.2: Esquema de uma repetição num grafo de *de Bruijn*.

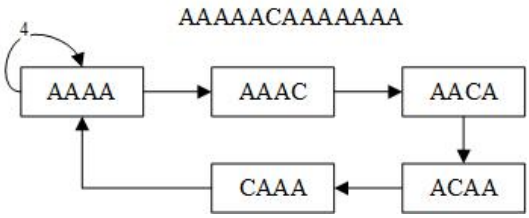


Figura 2.3: Um grafo de *de Bruijn* 4-dimensional para a cadeia AAAAACAAAAAAA. A aresta AAAA → AAAA tem multiplicidade 4.

Erros de sequenciamento também têm impacto negativo na montagem, visto que não existe mais consenso, o que altera a estrutura do grafo e o *tour* euleriano. O estado da arte descreve técnicas para o problema, como o sequenciamento *paired-end* e *mate-paired*. Alguns exemplos de ferramentas que levam em conta essas soluções são o Velvet (ZERBINO; BIRNEY, 2008), ABySS (SIMPSON et al., 2009) e SOAPdenovo (LI et al., 2010), entre outros, que usam diferentes abordagens para melhorar a qualidade da montagem utilizando o grafo de *de Bruijn* (MILLER; KOREN; SUTTON, 2010).

No genoma também pode ocorrer poliploidia, ou seja, a existência de cópias distintas de um mesmo gene. Embora possa ser compreendido como um erro no sequenciamento, é natural que isto ocorra. Neste caso, o grafo passa a ter dois caminhos distintos para um mesmo trecho da sequência, o que pode ser resolvido com um consenso feito pelo algoritmo, que agrega à sequência resultante informação sobre as bases poliplóides (CLAROS et al., 2012).

Para um genoma de tamanho G, o número de vértices é no máximo igual a $(G-k)$ e o número de arestas é igual a $(G-k)+1$, independentemente da profundidade de sequenciamento. Na prática, o número de vértices do grafo será muito maior que $G-k+1$, devido a muitos falsos *k-mers* causados por erros de sequenciamento. Um desses falsos *k-mers* cria muitos falsos caminhos no grafo, o que aumenta a complexidade do problema (LI et al., 2012). Um exemplo é mostrado na Figura 2.4. Neste caso, ocorre um erro de sequenciamento que impossibilita o

casamento entre sufixo e prefixo. Com $k = 3$, por exemplo, o erro de sequenciamento faz com que, caso sejam considerados somente alinhamentos exatos, não exista um percurso no grafo que alinhe os dois fragmentos, já que os *k-mers* `TCC` e `TCG` deveriam rotular o mesmo vértice.

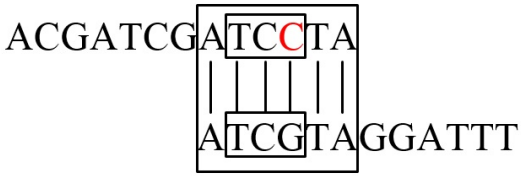


Figura 2.4: Exemplo de mismatch na construção do grafo de *de Bruijn*.

Uma implementação canônica para um grafo de *de Bruijn* pode representar os vértices como estruturas e as arestas como ponteiros entre os vértices. Se o *k-mer* que rotula cada vértice é representado como como um inteiro de 64 bits, assumindo-se $k \leq 32$, e as adjacências são representadas por dois inteiros de 32 bits, associados ao ponteiro para o vizinho e à multiplicidade da adjacência, considerando são quatro possíveis sucessores, cada vértice pode ser representado com 56 bytes. Para representar o genoma humano, incluindo o complemento reverso, seriam necessários 4.8 bilhões de vértices, totalizando 250 GB de armazenamento. Não obstante, ainda são necessárias estruturas adicionais para avaliar a existência das *k*-mers durante a construção do grafo. Caso seja utilizada uma tabela *hash*, considerando um valor de *hash* e o ponteiro correspondente, além um fator de carga igual a 1, seriam necessários mais 16 *bytes* por vértice, totalizando mais 70 GB de espaço necessário (CONWAY; BROMAGE, 2011). A grande quantidade de memória necessária para armazenar essas estruturas justifica a adoção de representações adaptadas à memória externa.

2.3 Modelos de Memória Externa

2.3.1 Modelo I/O

O modelo I/O (AGGARWAL; VITTER JEFFREY, 1988) é uma arquitetura de memória externa constituída por uma memória principal de tamanho M, acessada diretamente pela CPU, e uma memória externa de tamanho arbitrário. A memória é dividida em blocos de tamanho B e um programa nesta arquitetura carrega explicitamente blocos da memória secundária na

memória principal. Diversas abordagens de algoritmos de memória externa foram publicadas recentemente (CHIANG et al., 1995; ARGE, 2003; VITTER, 2007; ARGE; SITCHINAVA; GOODRICH, 2010).

A análise de um algoritmo desenvolvido no modelo I/O é feito em termos de blocos transferidos entre os dois níveis. Exemplos de algoritmos com complexidade conhecida são a leitura de um *array*, para o qual são necessárias $\Theta(\frac{N}{B})$ transferências de memória, também chamado *limite linear*. Também provou-se que são necessárias $\Theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ na ordenação, conhecido como *limite de ordenação* (AGGARWAL; VITTER JEFFREY, 1988).

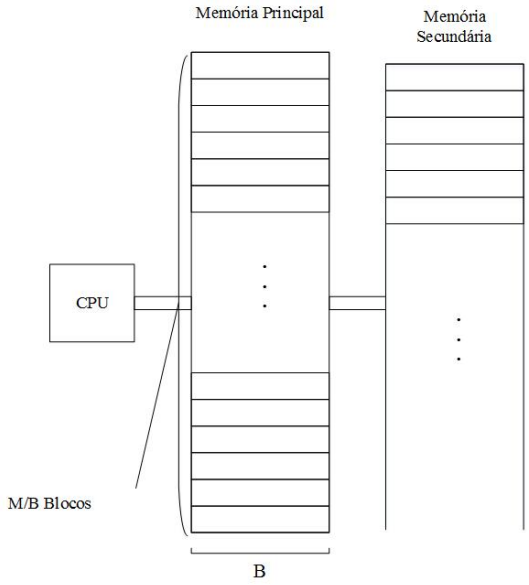


Figura 2.5: Modelo I/O.

2.4 Algoritmos Cache-eficientes

Os algoritmos **cache-eficientes** são algoritmos de memória externa que funcionam e são analisados sobre uma arquitetura com uma memória principal de tamanho arbitrário e uma memória cache de tamanho M. Ambas as memórias são divididas em blocos de tamanho B, de modo que a cache é composta por M/B blocos. Para a análise dos algoritmos *cache-oblivious*, FRIGO et al. (1999) propuseram o **modelo cache-ideal**. Nesse modelo, a análise de um algoritmo com entrada de tamanho n numa arquitetura de cache com tamanho M e bloco de tamanho B é feita através de duas medidas: a complexidade de trabalho $W(n)$, equivalente à análise feita no modelo RAM, e a complexidade de cache $Q(n;M,B)$, que representa o número

de cache misses causado pelo algoritmo. Se um algoritmo utiliza como entrada parâmetros físicos da cache de modo a otimizar o programa, este algoritmo é denominado *cache-aware*. Caso contrário, é denominado *cache-oblivious*.

O modelo *cache-ideal* possui as seguintes características:

- exatamente dois níveis de memória. Além da memória *cache*, somente existe um nível inferior, a memória principal.
- substituição ótima. Na ocorrência de um *cache miss* é substituído o bloco que será utilizado mais no futuro que os demais.
- associatividade total. Uma linha de memória pode ser armazenada em qualquer ponto da memória *cache*.
- substituição automática. Não é necessário manipular os blocos da cache explicitamente.

Adicionalmente, assume-se que a cache é **alta**, isto é, $M = \Theta(B^2)$. Um algoritmo projeto no modelo cache ideal é capaz de executar em uma arquitetura de cache multinível e com algoritmo de substituição LRU com sobrecarga dada por um fator constante.

Teorema 2.2. *Considere um algoritmo que causa $Q^*(n,M,B)$ cache misses em um problema com tamanho n usando uma cache ideal de tamanho (M,B). Então, o mesmo algoritmo incorre em $Q(n;M,B) \leq 2Q^*(n;M,B)$ cache misses em uma cache que com substituição LRU. (FRIGO et al., 1999)*

Corolário 2.2. *Um algoritmo qualquer com complexidade de cache igual a $Q(n;M,B)$ no modelo cache-ideal satizfaz a condição de regularidade*

$$Q(n;M,B) = O(Q(n;2M,B)), \tag{2.1}$$

e o número de cache misses com substituição LRU é $\Theta(Q(n;M,B))$. (FRIGO et al., 1999)

Teorema 2.3. *Um algoritmo cache-oblivious ótimo cuja complexidade de cache satisfaz a condição de regularidade (Corolário 2.2) pode ser implementado de maneira ótima em modelos de multinível com gerenciamento de memória explícito. (FRIGO et al., 1999)*

2.4.1 Relação entre implementações *cache-aware* e *cache-oblivious*

Um questionamento existente é a relação entre entre o desempenho real de um algoritmo *cache-oblivious* e o de seu equivalente *cache-aware* quando, em tese, suas complexidades são equivalentes ou similares. As abordagens *cache-oblivious* são normalmente baseadas em estratégias complexas que dividem o problema de modo a fazer com que os dados caibam na memória cache ou que fiquem localizados em uma mesma região de memória. Essa abordagem pode piorar o desempenho, considerando o custo adicional para sua implementação em relação às versões em que isso não é necessário (SACH; CLIFFORD, 2008). Contudo, os algoritmos *cache-aware* têm desvantagem quando considera-se a complexidade e a diversidade dos sistemas comerciais existentes, uma vez que seriam necessários diversos parâmetros que normalmente não são disponibilizados pelos fabricantes, como tamanho dos blocos de cada nível, latência, quantidade de níveis de memória cache, entre outros.

Quanto ao aspecto prático do desenvolvimento, os algoritmos *cache-oblivious* têm vantagem em relação às implementações *cache-aware*. O programador pode desenvolver um programa de mais alto nível no caso de um algoritmo *cache-oblivious*, já que não há necessidade de referência explícita a detalhes de baixo nível, como tamanho de bloco da memória, transferência de blocos em níveis com B distinto ou local de armazenamento de blocos. Além disso, se um algoritmo *cache-aware* utiliza subrotinas externas, estas subrotinas também devem utilizar algoritmos e estruturas de dados *cache-aware*, o que aumenta ainda mais a quantidade trabalho necessária ao desenvolvimento.

As versões de algoritmos *cache-aware* historicamente tiveram um desempenho melhor em relação às versões *cache-oblivious*, visto que possuem mais informação sobre a arquitetura em que são processados (LADNER; FORTNA; NGUYEN, 2002; SACH; CLIFFORD, 2008). Entretanto, a obtenção dessa informação necessária à execução de algoritmos *cache-aware*, feita em tempo de compilação ou execução, normalmente é difícil ou mesmo impossível.

Os algoritmos *cache-oblivious*, por sua vez, têm capacidade de alcançar resultados assintoticamente ótimos quanto ao número de *cache misses*, isto é, sem considerar fatores constantes. Assim sua complexidade é definida em termos de M e B sem que seja necessário que o algoritmo tenha conhecimento desses valores.

2.5 Algoritmos e Estruturas de Dados para Grafos *Cache-oblivious*

Na literatura estão disponíveis diversos exemplos de algoritmos, como *Depth First Search* (DFS), *Breadth First Search* (BFS), *Minimum Spanning Tree* (MST), *Single-source Shortest Path* (SSSP), e estruturas de dados, como filas de prioridades, árvores binárias de busca, B-trees, entre outras (BRODAL et al., 2004; BENDER; FARACH-COLTON; KUSZMAUL, 2006; ALLULLI, 2007; ARGE et al., 2007; BENDER; FARACH-COLTON, 2007). A seguir será descrito o algoritmo de DFS, desenvolvido por ARGE et al. (2007), uma vez que esta foi a opção escolhida para implementação neste trabalho.

2.5.1 DFS *Cache-oblivious*

Definição 2.3. A busca em profundidade em um grafo $G = (V, E)$ a partir de uma origem v é a exploração das arestas do vértice mais recentemente descoberto que ainda possua arestas não visitadas (CORMEN et al., 2009).

Uma implementação canônica deste algoritmo pode ser obtida utilizando-se uma pilha P, que armazena o nós v que ainda não foram visitados, mas que possuem uma aresta (w, v) incidente a um vértice já visitado w, e que representa o caminho entre o nó atual e nó raiz da busca, e um *array* A que contém um *bit* para cada nó v do grafo, indicando se v já foi visitado ou não. O vértice v no topo da pilha é considerado repetidamente. Quando um vértice não visitado v é alcançado pela busca, ele é marcado como visitado em A e todos os vértices adjacentes a v são empilhados em P.

ARGE et al. (2007) define uma versão *cache-oblivious* para o algoritmo de busca em profundidade primeiro baseada em um algoritmo equivalente para o modelo I/O descrito por BUCHSBAUM et al. (2000), com complexidade de memória igual a $O((V + \frac{E}{B}) \log_2 V + sort(E))$. Este algoritmo utiliza um conjunto de estruturas de dados composto por uma pilha, V filas de prioridades e uma *buffered repository tree*, descrita abaixo.

Uma pilha que é implementada utilizando-se um *array* dinâmico necessita de $O(1/B)$ *cache misses* amortizados para cada operação de *push* ou *pop*. Isto ocorre devido à estratégia de paginação ótima, a qual garante que o último bloco do *array*, utilizado no *push* e *pop*, estará

sempre presente na cache.

Uma *Buffered Repository Tree* (BRT) consiste de uma árvore binária estática com V folhas rotuladas por chaves no intervalo $[1..V]$, em ordem, como ilustrado na Figura 2.6. Na estrutura são armazenados $O(E)$ elementos. A cada nó interno está associado um *buffer* que armazena elementos a serem inseridos em lote na estrutura. O *buffer* associado a uma folha da BRT com chave v armazena somente elementos com chave v. São implementadas duas operações: $insert(v,T)$ e $extract(T)$. A operação $insert(v,T)$ insere um elemento com chave v no *buffer* da raiz. A operação $extract(v,T)$ extrai todos os elementos com chave v da BRT T. Para tanto, é percorrido o caminho entre a raiz e a folha com chave v. A cada nó interno visitado na árvore, são removidos e reportadas os elementos com chave igual a v. Os elementos restantes são distribuídos entre os *buffers* dos nós filhos. Assim, um elemento com chave w é inserido no *buffer* do nó que está no caminho entre a raiz e a folha com chave a w. Um conjunto de elementos adicionados simultaneamente a um dado *buffer* deve estar agrupado em um espaço de memória contíguo, mas não necessariamente no mesmo espaço de memória que os elementos já existentes, de modo que o *buffer* de um nó pode ser visto como o encadeamento de conjuntos de elementos armazenados em um mesmo espaço de memória. Visto que todas as inserções ocorrem na raiz da estrutura, este *buffer* é implementado como um *array* dinâmico, o que faz com que o custo amortizado de inserção seja $O(\frac{1}{B})$. A complexidade de memória da estrutura é $O(\frac{1}{B}\log_2 V)$ para inserção e $O(\log_2 V)$ para extração.

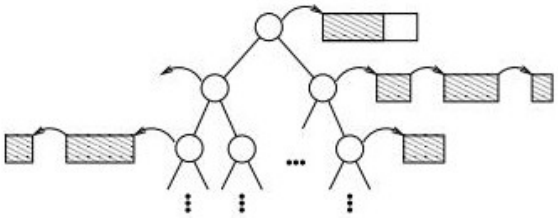


Figura 2.6: Esquema de uma *Buffered Repository Tree* descrita por ARGE et al. (2002).

Em lugar da fila de prioridades utilizadas no algoritmo definido por BUCHSBAUM et al. (2000), utiliza-se outra estrutura de dados na implementação do algoritmo: a *Buffered Priority Tree* (BPT). Construída a partir de E_v elementos com E_v chaves distintas, BPT tem a mesma estrutura que a BRT, de modo que os E_v elementos são armazenados nas folhas da raiz, em ordem

crescente das chaves. Em contrapartida, neste caso os *buffers* dos nós são utilizados para efetuar remoções em lote. Além disso, a cada nó μ da BPT é associado um contador que indica o número de elementos não removidos armazenados na subárvore da qual μ é raiz. Dados E' elementos, é possível efetuar uma operação de remoção em lote adicionando-se esses elementos ao *buffer* da raiz. Percorre-se, então, o caminho entre a raiz e o nó que possui a menor chave na árvore. A cada nó μ visitado neste caminho, os elementos contidos no *buffer* devem ser distribuídos entre os nós-filhos da esquerda, μ_e, e da direita, μ_d, assim como ocorre na BRT, atualizando-se os respectivos contadores. Se o nó μ_e tem contador maior que zero, visita-se recursivamente μ_e. Caso contrário, se μ_e tem contador maior que zero, visita-se μ_d recursivamente.

2.5.1.1 Algoritmo DFS *Cache-oblivious*

Inicialmente, o BRT T está vazio e cada BPT $P(v)$ é construída a partir das arestas E_v incidentes a v. A chave de uma aresta (u,w) é u em T e w em $P(u)$. O nó de origem da DFS é colocado na pilha S. O nó u no topo de S é considerado repetidamente. As arestas com chave u são extraídas de T e removidas em lote de $P(u)$. Se a operação de remoção devolve uma aresta (u,v), o vértice v é numerado e empilhado em S, e as arestas incidentes a v são inseridas em T. Caso contrário, $P(u)$ está vazio, então u é desempilhado de S. O procedimento deve ser repetido até que S esteja vazia. Este procedimento será discutido novamente na Seção 3.2.1.

2.5.2 Fila de Prioridades *Cache-oblivious*

Uma fila de prioridades com estrutura *cache-oblivious* foi definida por ARGE et al. (2007). Esta estrutura é caracterizada pela divisão de seus elementos N elementos em $\Theta(\log\log N_0)$ níveis, os quais têm tamanhos que variam entre $N_0 = \Theta(N)$ e c acima de um limiar constante c_t. O primeiro e mais baixo nível da estrutura é o nível c e cada nível i subsequente tem tamanho $N_0^{(2/3)^{i-1}}$ maior que o imediatamente precedente. A denominação de cada nível corresponde assintoticamente ao número de elementos que pode comportar, i.e., o nível $X^{2/3}$, é capaz de armazenar $X^{2/3}$ elementos.

A fila de prioridades possui duas operações: *push*, que insere $\lfloor X \rfloor$ elementos em um nível $X^{3/2}$; e *pull*, que remove e devolve $\lfloor X \rfloor$ elementos de um nível $X^{3/2}$, ou de níveis superiores.

Ambas as operações as quais são sempre efetuadas no nível c e podem propagar-se aos níveis mais altos. Estas duas operações são utilizadas para implementar a inserção e a remove-min de elementos na *priority queue*.

Para uma memória principal $O(M)$, ambas as operações *push* e *pull*, efetuadas sobre $\lfloor X \rfloor$ elementos um nível $X^{3/2}$ têm custo $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$ de transferências de memória e tempo amortizado $O(X/B)$, sem considerar o custo de chamadas recursivas.

2.5.2.1 Estrutura dos níveis

Cada nível é constituído por um conjunto de *buffers*, que podem servir para armazenar ou transferir elementos entre os níveis da *priority queue*. Um nível $X^{2/3}$ é composto por um *up buffer* $u^{X^{3/2}}$, formado por até $\lfloor X^{3/2} \rfloor$ elementos, e no máximo $\lceil X^{1/2} \rceil + 1$ *down buffers* $d_1^{X^{3/2}}, \ldots, d_{\lceil X^{1/2} \rceil + 1}^{X^{3/2}}$. O primeiro *down buffer* é composto por até $2\lfloor X \rfloor - 1$ elementos, e os demais podem armazenar entre $\lfloor X \rfloor$ e $2\lfloor X \rfloor - 1$ elementos.

Intuitivamente, os elementos em *up buffers* estão "subindo" e os elementos em *down buffers* estão "descendo" na *priority queue*. Algumas invariantes são mantidas em relação à manipulação dos elementos nos *buffers*:

- Os elementos estão ordenados entre os *down buffers* de um mesmo nível, isto é, os elementos de um buffer $d_i^{X^{3/2}}$ têm valores menores que os elementos contidos em $d_{i+1}^{X^{3/2}}$. Entretanto, os elementos em um *down buffer* $d_i^{X^{3/2}}$ estão desordenados entre si.

- Em um nível $X^{3/2}$, elementos nos *down buffers* têm valores menores que os elementos contidos no *up buffer* $u^{X^{3/2}}$.

- Os elementos dos *down buffers* de um nível $X^{3/2}$ têm valores menores que que os contidos nos *down buffers* de um nível superior $X^{9/4}$.

O elemento de maior valor em cada um *down buffer* $d_i^{X^{3/2}}$ é denominado *pivô*. Os pivôs determinam os limites dos intervalos de valores armazenados em cada *down buffer*.

Essas invariantes garantem que os valores dos elementos crescem à medida que os níveis da *priority queue* são percorridos do nível c ao nível N_0. Assim, o elemento com menor valor em

toda a estrutura está contido no nível c.

O *up buffer* e os *down buffers* de cada nível $X^{3/2}$, nesta ordem, são armazenados contiguamente na memória. Os *down buffers* são armazenados numa ordem ordem arbitrária na região de memória que ocupam, entretanto, cada um destes tem um apontador para o seguinte. O tamanho total ocupado pelo *array* que contém cada nível é $\sum_{i=0}^{\log_{3/2}\log_c N_0} O(N_0^{(2/3)^i}) = O(N_0)$.

2.5.2.2 Inserção e remove-min

As operações de *push* e *pull*, mais gerais, são utilizadas para implementar operações mais sofisticadas na fila de prioridades: inserção e remove-min. Ambas as operações são efetuadas sobre dois *buffers* armazenados na memória principal, um de inserção e outro de remoção, respectivamente, os quais contém no máximo $c^{2/3}$ elementos cada. Em uma operação de inserção, podem ocorrer duas situações:

- Se a chave do elemento é maior que a maior chave contida no *buffer* de remoção, então o elemento é inserido no *buffer* de inserção;
- Caso contrário, o elemento é inserido no *buffer* de remoção e o elemento com maior chave no buffer de remoção é inserido no *buffer* de inserção.

Uma vez que, intuitivamente, o *buffer* de remoção contém os elementos de menor chave entre todas as chaves da fila de prioridades, a operação de remove-min consiste em remover e devolver o elemento de menor chave contido no *buffer* de remoção.

Caso fique vazio em algum momento, o *buffer* de remoção é preenchido com os $\lfloor c^{2/3} \rfloor$ menores elementos entre os $\lfloor c^{2/3} \rfloor$ obtidos de uma operação *pull* efetuada sobre nível c da fila de prioridades e os elementos do *buffer* de inserção, o qual não tem seu número de elementos alterado. Se o *buffer* de inserção torna-se cheio em algum momento, seus $\lfloor c^{2/3} \rfloor$ elementos são inseridos no nível c da fila de prioridades com uma operação *push*. A não ser pelas chamadas recursivas das operações *push* e *pull*, o custo das operações é constante e não implica em transferências de memória. Quando as operações *push* e *pull* são consideradas, o custo amortizado é de $O(\frac{1}{B}\log_{M/B}\frac{1}{B})$ transferências de memória e $O(\log_2 N_0)$ em custo amortizado de tempo de processamento.

2.5.3 *List Ranking*

Dado um *array* não ordenado de elementos de uma uma lista ligada, cada qual aumentado com um ponteiro para o próximo elemento da lista. O *list ranking* consiste em, determinar a ordem de cada um dos elementos em relação ao fim da lista, isto é, seu *rank* (WYLLIE, 1979). As abordagens desenvolvidas com base em arquiteturas PRAM (Parallel Random Access Machine) podem ser utilizadas para desenvolver versões que calculam o *ranking* dos elementos da lista (REIF, 1993; VISHKIN, 1985). O algoritmo consiste no que segue: um conjunto de elementos independentes, isto é, que não possuem arestas entre si, é selecionado. As arestas de cada um dos nós deste conjunto são removidas, os nós restantes na lista são recursivamente ranqueados e os nós que haviam sido removidos têm seu *rank* computado e são reintegrados à lista.

O principal desafio para que o *list ranking* possa ser utilizado em um algoritmo cache-oblivious é a etapa de seleção do conjunto de nós independentes. As demais etapas do algoritmo podem ser efetuadas com leituras de memória e ordenação, que têm custo igual a $O(sort(V))$ transferências de memória (ARGE et al., 2007).

2.5.4 *Tour* Euleriano *Cache-oblivious*

Um caminho euleriano em um grafo dirigido pode ser obtido de diversas maneiras, entretanto, algumas premissas devem ser respeitadas. São estas:

1. Uma aresta (u,v) é selecionada somente se existe uma saída garantida através de uma aresta (v,w)

2. Todos os vértices têm grau par, ou existem, no máximo, dois vértices v e u tais que $grau\_entrada(v) - grau\_saida(v) = 1$ e $grau\_saida(v) - grau\_entrada(u) = 1$. Neste caso, o percurso deve começar em u e terminar em v.

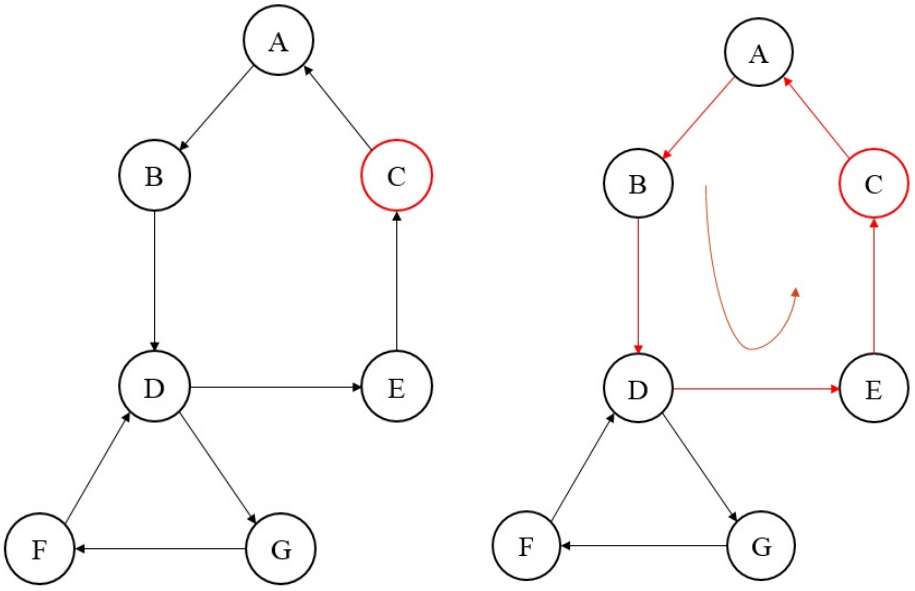
Além disso, para que exista um *tour* euleriano, é necessário que todos os vértices do grafo tenham grau par. Caso isso não ocorra, somente dois vértices podem ter grau ímpar, de tal maneira que um tenha maior grau de saída e o segundo tenha maior grau de entrada.

Umas das maneiras de obter o *tour* euleriano mantendo as premissas é iniciar o caminho no vértice com maior grau de saída, caso exista, e terminá-lo no que possui maior grau de entrada (HIERHOLZER; WIENER, 1873). Assim, podem ser selecionadas arestas de acordo com um critério arbitrário, e cada aresta selecionada é colocada em uma pilha. No instante em que é alcançado um vértice no qual não existe saída, as arestas são desempilhadas até que seja alcançado um vértice que contenha uma aresta de saída. Cada aresta desempilhada é inserida em uma lista de vértices. Caso o grafo seja euleriano, o caminho resultante é a lista de vértices invertida. Um exemplo é dado na Figura 2.7, que devolve o caminho `C-A-B-D-G-F-D-E-C`.

Utilizando-se de estratégias definidas em algoritmos PRAM, é possível obter um caminho de Euler em um dado grafo através de um algoritmo *cache-oblivious* (TARJAN; VISHKIN, 1984). Nem todo grafo necessariamente possui um caminho euleriano, entretanto, uma árvore em que cada aresta não-direcionada é substituída por duas arestas direcionadas, em direções opostas, possui um caminho euleriano (ARGE et al., 2007).

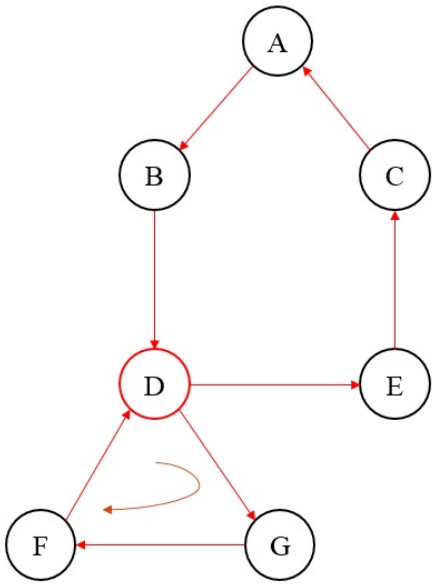
Para que seja calculado um caminho euleriano em uma árvore, suponha-se inicialmente que, para cada vértice v, é definida uma lista ordenada de visita aos seus vizinhos. Se nesta lista duas arestas (u,v) e (w,v) estão próximas, a aresta subsequente à (u,v) no caminho euleriano é a aresta (v,w). Esse processo é efetuado com uma leitura das listas de arestas e resulta em uma lista de todas as arestas do grafo e suas respectivas arestas sucessoras. O ranqueamento e ordenação das arestas de acordo com seu *ranking* devolve um caminho euleriano naquele grafo.

Com base nesse algoritmo de *tour* euleriano, é possível desenvolver algumas versões *cache-oblivious* para árvores dos algoritmos de BFS, DFS e decomposição de centróides em $O(sort(V))$ transferências de memória.



(a) Grafo.

(b) Primeiro ciclo.



(c) Segundo ciclo.

Figura 2.7: *Tour* euleriano.

2.6 Estruturas de Dados Sucintas

Estruturas de dados sucintas caracterizam-se por ocuparem uma quantidade de espaço próxima ao mínimo teórico que seria necessário para armazenar uma determinada quantidade de informação. Assim, se, teoricamente, uma quantidade de informação ocuparia n bits, uma

estrutura de dados sucinta para armazenar esta mesma informação ocupa, no máximo, $n+o(n)$ bits [1]. JACOBSON (1989) definiu-as com o objetivo de codificar *arrays* de bits, árvores com nós não rotulados e grafos planares.

O que distingue as estruturas de dados sucintas das demais estruturas de dados comprimidas é a possibilidade de serem efetuadas eficientemente operações sobre os dados comprimidos, o que também as torna boas escolhas em contrapartida aos algoritmos de compressão sem perda de dados, haja vista ser necessário realizar a descompressão dos dados antes de seu uso.

Uma estrutura de dados sucinta para representar uma cadeia S deve suportar ao menos três operações básicas: $access(S,i)$, $rank_c(S,i)$ e $select_c(S,i)$.

Definição 2.4. Seja uma *string* S de tamanho n com alfabeto Σ. $access(S,i)$ devolve o caractere da posição i em S, tal que $0 \leq i < n$.

Definição 2.5. Seja uma *string* S com alfabeto Σ. $rank_c(S,i)$ devolve o número de ocorrências de c em S até a posição i, onde $c \in \Sigma$ e $0 \leq i \leq |S|$. Além disso, para quaisquer c e S, $rank_c(S,0) = 0$.

Definição 2.6. Seja uma *string* S com alfabeto Σ. Define-se $select_c(S,i)$ como a posição do i-ésimo c em S, onde $0 \leq i \leq rank_c(S,n)$, $c \in \Sigma$ e $n = |S|$. Para quaisquer c e S, $select_c(S,0) = 0$ e para qualquer $j > rank_c(T,n)$, $select_c(S,j) = n+1$.

2.6.1 Representação Sucinta de um Grafo de *de Bruijn*

Embora a representação canônica de um grafo de *de Bruijn* seja, em termos de construção, vantajosa quando comparada à abordagem *overlap-layout-consensus*, o tamanho da estrutura tipicamente torna-se um problema (CHIKHI et al., 2014). Assim, algumas abordagens mais recentes tiveram como base estruturas de dados sucintas, utilizando representações alternativas do grafo de *de Bruijn* (CONWAY; BROMAGE, 2011; CONWAY et al., 2012; CHIKHI; RIZK, 2012). Uma vez que os dados estão representados em uma quantidade menor de espaço, em tese os acessos aos níveis mais baixos de memória sejam menos frequentes, o que, a princípio, reduziria o tempo de processamento, considerando que nível da memória é diretamente proporcional ao tempo de acesso.

[1] a notação o (o-pequeno) é mais estrita que a notação O

Seja um *string* S formada pelo alfabeto $\mathscr{A}$. Além disso, considere-se um conjunto $\mathscr{A}^-$, tal que $|\mathscr{A}| = |\mathscr{A}^-|$ e $\mathscr{A} \cap \mathscr{A}^- = \emptyset$. Seja também c^- um elemento de $\mathscr{A}^-$ correspondente a um elemento $c \in \mathscr{A}$ e uma função tal que $u(c^-) = c$, para qualquer $c^- \in \mathscr{A}^-$, e $u(c) = c$, para qualquer $c \in \mathscr{A}$. A representação sucinta de um grafo de *de Bruijn* $G = (V, E)$ k-dimensional de S descrita por BOWE et al. (2012) é constituída por:

- uma *string* W, formada por caracteres de $\mathscr{A}$ e $\mathscr{A}^-$;
- uma *string* *last*, formada pelo alfabeto binário $\{0, 1\}$;
- um *array* F, com tamanho $\sigma = |\mathscr{A}|$, também chamado *diretório*.

Na Figura 2.8 é mostrada a representação sucinta para um grafo de *de Bruijn* 3-dimensional que representa a cadeia `TACGACGTCGACT`. Na tabela Node, mostrada apenas para fins de ilustração, são exibidos os rótulos dos vértices de G ordenados de acordo com a ordem lexicográfica das respectivas cadeias reversas. Cada caractere $W[i]$ representa o rótulo de uma aresta (u, v), onde $rotulo(u) = Node[i]$. Por conveniência, são adicionados k caracteres \$ antes da cadeia e um depois, de modo que cada aresta de G corresponde a um caractere de S. Como consequência da ordenação dos nós, arestas com mesmo rótulo são posicionadas numa mesma região de W. O *array* F armazena as frequências cumulativas dos blocos de rótulos de nós terminados em `A`, `C`, `G` e `T`, indicando as posições em W e *last* que correspondem aos inícios desses blocos.

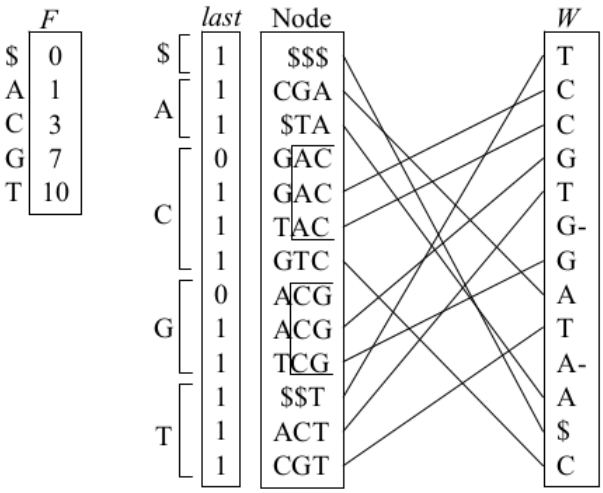


Figura 2.8: Representação sucinta de um grafo de *de Bruijn* descrita por BOWE et al. (2012).

Seja $Node[i]$ o rótulo do vértice u correspondente à linha i. O caractere $W[i]$ corresponde a uma aresta (u,v). O rótulo de v, nesse caso, será dado pelo sufixo de comprimento $k-1$ de $Node[i]$ concatenado com $W[i]$. A linha referente a v pode ser obtida a partir de i por $F[W[i]] + rank_{W[i]}(W,i)$. Por exemplo, considere a linha $i = 4$ na Figura 2.8, que representa a aresta $u =$ "GAC" $\xrightarrow{\mathtt{G}} v =$ "ACG". Nesse caso, a linha correspondente a v é dada por $F[\mathtt{G}] + rank_{\mathtt{G}}(W) = 7 + 1 = 8$. Repare que múltiplas linhas podem referir-se ao mesmo nó, porém apenas uma delas possui o valor correspondente $last[i'] = 1$. Assim, todo $Node[i]$ com $last[i] = 1$ é distinto, de modo que o conjunto desses $Node[i]$ tem uma correspondência um-para-um com o conjunto V. Portanto, um nó $v \in V$ pode ser rotulado pelo índice i correspondente em $Node$ e $last$, de modo que $last$ possui $|V|$ ocorrências de 1. Essas propriedades servem como base para que se possa "navegar" pelo grafo de maneira eficiente, conforme veremos no Capítulo 3.

3

Desenvolvimento

De maneira a analisar algoritmos *cache-oblivious* e estruturas de dados sucintas, o desenvolvimento do trabalho consistiu de duas fases principais: a implementação e comparação de versões canônicas dos algoritmos de DFS e *tour* euleriano em um grafo de *de Bruijn* canônico com suas correspondentes *cache-oblivious*, além do algoritmo de *tour* euleriano em uma representação sucinta do grafo. O objetivo é avaliar o desempenho de cada versão quanto à eficiência de uso da memória cache.

Uma vez que o interesse central deste trabalho são os grafos de *de Bruijn*, estes grafos foram usados como entrada para os algoritmos DFS, com a mesma estrutura utilizada nas implementações de *tour* euleriano presentes neste trabalho. As implementações aqui descritas foram realizadas de maneira fiel às referências utilizadas, obviamente utilizando dos artifícios necessários a uma implementação em linguagem de programação. Não foram utilizadas técnicas que não tenham sido explicitadas de maneira a melhorar o resultado de alguma estrutura ou algoritmo específico.

3.1 Implementação com Listas de Adjacências com *Arrays* Dinâmicos

A representação utilizada para a versão canônica de um grafo de *de Bruijn* $G = (V, E)$ k-dimensional para uma *string* S foi a de listas de adjacências, como mostrado na Figura 3.1. A estrutura é composta por um *array* dinâmico V que contém os $|V|$ vértices correspondentes às k-mers. Cada vértice v está associado com um *array* dinâmico $A[v]$ que contém suas adjacências. Um vértice presente em V é composto por um inteiro k_v, que indica a posição da k-mer que rotula v, uma chave v que indica sua posição ordinal em V e um ponteiro para o *array* de adjacências. Duas ocorrências de um mesmo k-mer em S estão associados a um mesmo vértice v, de modo que é guardado o primeiro desses valores. Uma vez que cada vértice v somente guarda a posição k_v, é necessário armazenar a sequência S no grafo. Abaixo é mostrada a estrutura do grafo em linguagem C. Como mostrado, também são armazenados os vértices *head* e *tail*, que indicam o início e o fim do percurso caso G tenha vértices com grau ímpar.

```
typedef struct DBG {
    int k;
    int number_edges;
    DynamicArray *nodes;
    DBGNode *head;
    DBGNode *tail;
    char *chain;
} DBG;
```

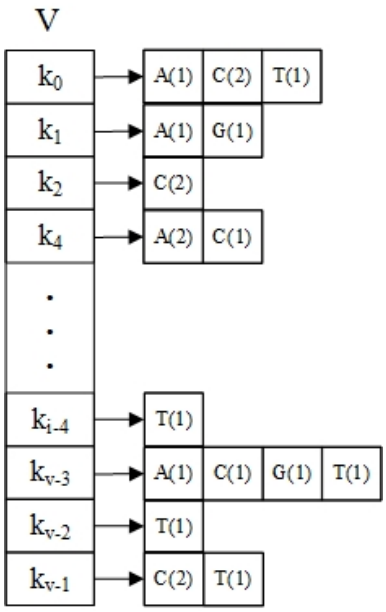


Figura 3.1: Representação por listas de adjacências.

No *array* de adjacências de u, cada posição representa a vizinhança a um nó distinto v e contém um ponteiro para o v e um inteiro que indica a multiplicidade da aresta. Se há em G uma aresta (u,v) e u e v são rotulados por k-mers $K(u) = s_u s_{u+1} \ldots s_{u+k-1}$ e $K(v) = s_v s_{v+1} \ldots s_{v+k-1}$, então $s_{u+1} \ldots s_{u+k-1} = s_v \ldots s_{v+k-2}$ e s_{v+k-1} é igual a um dos $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$. Assim, um vértice u tem, no máximo, m vizinhos distintos.

Uma etapa fundamental na construção do grafo é a verificação da existência um vértice rotulado por uma dada k-mer. Essa verificação foi feita utilizando-se uma tabela *hash*, na qual a chave é a própria k-mer e o valor é um ponteiro para a estrutura propriamente dita, caso exista. No Algoritmo 1 é mostrado o procedimento de construção do grafo.

Esse procedimento consiste em percorrer a cadeia S obtendo as k-mers correspondentes às posições i e $i+1$. Verifica-se a existência dessas k-mer na tabela *hash* com a função `ObterHash`($tabela$, $k-mer$). Caso exista, a estrutura correspondente é recuperada. Caso contrário, a estrutura do vértice para a k-mer é inicializada e armazenada na *hash* com a função `InsereHash`. De acordo com a propriedade do grafo de *de Bruijn*, é adicionada uma aresta dirigida entre os vértices correspondentes às k-mers correspondentes às posições i e $i+1$, o que é feito através da função `AdicionaAresta`(v,w).

Embora neste trabalho fale-se em grafo de *de Bruijn* k-dimensional de uma sequência S com alfabeto Σ, normalmente o obtido é um subgrafo de *de Bruijn*, visto que o grafo completo

```
Entrada : uma sequência S, um valor k e uma tabela hash que recebe uma string
          como chave
Saída   : um grafo de de Bruijn k-dimensional de S
1  início
2      grafo ← ∅;
3      para i ← 0 até Tamanho(cadeia) − k faça
4          vertice_esquerda ←ObterHash(tabela, Kmer(cadeia, k, i));
5          vertice_direita ←ObterHash(tabela, Kmer(cadeia, k, i + 1));
6          se vertice_direita é nulo então
7              InserirVertice(grafo, vertice_direita);
8              InserirHash(tabela, vertice_direita);
9          fim
10         se vertice_esquerda é nulo então
11             InserirVertice(grafo, vertice_esquerda);
12             InserirHash(tabela, vertice_esquerda);
13         fim
14         AdicionarAresta(vertice_esquerda, vertice_direita);
15     fim
16     devolva grafo
17 fim
```

Algoritmo 1: Construção do grafo.

é composto por todas as *strings* de tamanho k formadas pelo alfabeto Σ. Além disso, para fins de simplificação, o grafo de *de Bruijn* é formado a partir da cadeia S completa, e não a partir das *reads*, como ocorre tipicamente nos *softwares* de montagem de fragmentos. Isso é motivado principalmente pela necessidade de uso de outros conceitos que não são foco deste trabalho, como correção de erros nos fragmentos ou cobertura no sequenciamento do DNA.

3.1.1 Algoritmo DFS Canônico

A versão implementada do algoritmo de busca em profundidade pode ser vista no Algoritmo 2. Esta implementação é baseada em uma pilha em que os nós são empilhados explicitamente, em lugar de uma versão recursiva, em que as chamadas seriam empilhadas logicamente. A escolha foi feita com base em testes empíricos, em que a versão iterativa do algoritmo mostrou-se capaz de executar com uma entrada cerca de cinco vezes maior.

Teorema 3.1. *Uma implementação canônica do algoritmo de busca em profundidade incorre em $O(|V| + |E|)$ transferências de memória no pior caso.*

Demonstração. O algoritmo empilha/desempilha cada um dos vértices exatamente um vez. Pelo

```
   Entrada : um grafo G = (V,E)
   Saída   : numeração dos vértices de G por visita em profundidade
 1 início
 2     S ← ∅;
 3     faça
 4         v ← Desempilha(S);
 5         se v não está marcado como visitado então
 6             marcar v como visitado;
 7             para cada vizinho u de v faça
 8                 Empilha(u, S);
 9             fim
10         fim
11     enquanto S não é vazia;
12 fim
```

Algoritmo 2: Algoritmo padrão de busca em profundidade.

nó desempilhado, ele percorre sua lista de vizinhos e verifica se o vizinho já está marcado como visitado. Logo, num total de $2E$ consultas a *arrays*. Portanto, supondo, no pior caso, que cada acesso resulta em um cache miss, temos $O(V+E)$ cache misses no total. □

3.1.2 Algoritmo Canônico de *Tour* Euleriano

Haja vista haverem três abordagens distintas para o mesmo problema de *tour* euleriano, é necessário estabelecer alguns parâmetros de comparação. O primeiro, é verificar se o grafo é de fato euleriano, isto é, se todos os vértices possuem grau par, i.e., $|\{(u,v) : u \in V, v \in V\}| = |\{(v,w) : v \in V, w \in V\}|$. Em todas as implementações realizadas, esta é uma premissa. Em segundo, é importante estabelecer que o algoritmo será basicamente o mesmo, alterando-se somente as chamadas para os procedimentos específicos de cada implementação, mas com um retorno semelhante em todos os casos. O Algoritmo 3 descreve o procedimento básico para obtenção de um *tour* euleriano em G. Seu custo é $O(log_2 V)$, como mostrado no Teorema 3.2.

Teorema 3.2. *A implementação canônica do algoritmo de tour euleriano incorre em $O(E)$ transferências de memória.*

Demonstração. O Algoritmo 3 visita cada aresta $v \xrightarrow{r} w$ exatamente uma vez, empilhando/desempilhando os valores v e r nas pilhas S e R respectivamente. São portanto, necessários $O(E)$ acessos à pilha, cada um dos quais pode representar um *cache miss* no pior caso. O tour Euleriano

Entrada : um grafo $G = (V, E)$ e uma *raiz* do caminho
Saída : um *tour* euleriano em G

```
1  início
2      S ← ∅;
3      R ← ∅;
4      v ← raiz;
5      caminho ← ε;
6      faça
7          se existe uma aresta não visitada (v,w), para w ∈ V então
8              Empilha(S, v);
9              Empilha(R, Rotulo((v,w)));
10             v ← w;
11         fim
12         senão
13             r ← Desempilha(R);
14             Concatenar(caminho, r);
15             v ← Desempilha(S);
16         fim
17     enquanto a pilha S não é vazia;
       /* O caminho é obtido na ordem inversa, portanto,
          deve-se invertê-lo para obter o caminho correto
          */
18     devolva Inverte(caminho)
19 fim
```

Algoritmo 3: *Tour* euleriano.

é construído de trás para a frente, mas a inversão da variável *caminho* pode ser feita em $O(E/B)$ acessos. Portanto, temos $O(E)$ *cache misses* no total. □

3.2 Implementação com Algoritmos e Estruturas de Dados *Cache-oblivious*

3.2.1 Algoritmo de Busca em Profundidade *Cache-oblivious*

O algoritmo definido por ARGE et al. (2007) utiliza um conjunto de estruturas de dados, dentre as quais aqui são descritas as implementações da *Buffered Repository Tree* (BRT) e *Buffered Priority Tree* (BPT).

3.2.1.1 Implementação de *Buffered Repository Tree* (BRT)

A BRT é uma árvore binária estática com $|V|$ folhas ordenadas, cada qual associada a uma chave no intervalo $[1,|V|]$. No algoritmo de busca em profundidade definido por ARGE et al. (2007), sua função é o armazenamento de arestas incidentes a nós que já foram visitados no percurso, com a função `BRTInserir(`(v,w)`)`, onde (v,w) é um elemento a ser armazenado, e extração em lote de um conjunto de arestas com mesma chave, através da operação `BRTExtrair(`v`)`, onde v é um inteiro tal que $1 \leq i \leq |V|$. A cada nó da BRT, exceto a raiz e as folhas, está associado um *buffer* especial, composto por conjuntos encadeados de elementos armazenados em um espaço de memória contíguo. Ao nó raiz está associado um *buffer* implementado como um *array* dinâmico.

`BRTInserir(`v`)` insere um elemento diretamente no *buffer* da raiz. Para efetuar `BRTExtrair(`v`)`, ilustrado no Algoritmo 4, é necessário fazer uma busca binária na árvore desde a raiz até a folha com chave v. Durante o percurso, é necessário distribuir os elementos contidos no *buffer* de cada um dos nós visitados entre seus filhos. Um elemento m contido um *buffer* associado a um nó μ da BRT é colocado no *buffer* do filho da esquerda de μ, caso este nó esteja no caminho entre a raiz da árvore e a folha com chave μ. O mesmo ocorre com o filho da direita de μ.

O procedimento `BRTDistribuir(`μ`, v)` itera sobre os elementos do *buffer* associado a um nó μ da BRT e efetua a distribuição desses elementos entre os filhos de μ. Os elementos com chave igual a v são removidos e reportados. Os elementos a serem transferidos simultaneamente de um nó-pai μ a um nó-filho μ_f devem ser armazenados em um espaço de memória contíguo, o qual deve ser encadeado no *buffer* de μ_f. Durante o processo de distribuição dos elementos de μ, pode-se utilizar dois *arrays* dinâmicos, correspondentes a cada um de seus filhos, visto que não há informação *a priori* sobre a quantidade de elementos que será distribuída para cada um dos filhos, além não existe garantia de ordenação entre os elementos de um mesmo *buffer*. Após o esvaziamento do *buffer* de μ, os elementos são então transferidos para o *buffer* correspondente.

Quando o procedimento `BRTExtrair(`T`, v)` alcança a folha com chave v, os elementos armazenados no *buffer* desse nó são removidos e devolvidos como resultado, juntamente com os nós reportados durante a execução de `BRTDistribuir(`m`, v)`.

Entrada : um grafo dirigido $G = (V, E)$ e uma raiz da busca
Saída : a enumeração dos vértices de por profundidade G

```
1  início
2      extraidos ← ∅;
3      m ← Raiz(brt);
4      inicio ← 0;
5      fim ← |V| − 1;
6      enquanto inicio ≠ fim faça
7          Acrescentar(extraídos, BRTDistribuir(m, v));
8          se v < ((fim + inicio)/2) então
9              m ← FilhoEsquerda(m);
10             fim ← ((fim + inicio)/2);
11         fim
12         senão se v > ((fim + inicio)/2) então
13             m ← FilhoDireita(m);
14             inicio ← ((fim + inicio)/2);
15         fim
16     fim
17     Acrescentar(extraidos, BRTBuffer(v));
18     devolva extraidos;
19 fim
```

Algoritmo 4: `BRTExtrair` (ARGE et al., 2007).

Lema 3.1. *Uma buffered repository tree cache-oblivious usa $\Theta(B)$ espaço em memória principal e suporta operações de inserção e extração em $O(\frac{1}{B}log_2 V)$ e $O(log_2 V)$ transferências de memória*

amortizadas, respectivamente. (ARGE et al., 2007)

3.2.1.2 Implementação de *Buffered Priority Tree* (BPT)

A *Buffered Priority Tree* tem a mesma estrutura da BRT: uma árvore binária com $|V|$ folhas ordenadas, associadas às chaves no intervalo $[1..|V|]$. Existem três diferenças essenciais:

1. a árvore é construída a partir de um conjunto pré-determinado de elementos;
2. os *buffers* associados a cada nó m são utilizados para remoção;
3. cada nó m possui um inteiro que indica o número de elementos armazenados na subárvore enraizada em m e não programados para remoção, i.e., não estão presentes nos *buffers*.

Durante a construção da árvore, os elementos são inseridos nos *buffers* das folhas com chave correspondente. Além disso, são inicializados os contadores dos nós de acordo com a quantidade de elementos armazenados em cada folha. De maneira semelhante a `BRTExtract`, para uma BPT T procedimento `BPTRemove(`T`,` E'`)` insere E' elementos, presumidamente armazenados em T, no *buffer* da raiz, o qual também é implementado como um *array* dinâmico. Em seguida, `BPTRemove(`T`,` E'`)` percorre o caminho entre a raiz e a folha de menor chave que ainda possui nós não removidos. A cada nó m no caminho percorrido, os elementos contidos no *buffer* de m são distribuídos entre os *buffers* de seus filhos da esquerda e da direita, m_e e m_d, respectivamente, atualizando seus contadores.

Caso o contador de m_e seja maior que 0, o procedimento visita m_e recursivamente. Caso contrário, visita-se m_d. Ao alcançar uma folha u, o procedimento devolve o elemento armazenado em u e atualiza o contadores de todos os nós m contidos no caminho entre u e a raiz de T.

Além de remover um conjunto de elementos em *batch*, essa operação funciona como uma *remove-min* existente em uma fila de prioridades. Por esse motivo, a BPT é usada como substituta na versão *cache-oblivious* do algoritmo de DFS definida por ARGE et al. (2007).

Lema 3.2. *Sem o uso de memória principal permanente, uma buffered priority tree cache-oblivious suporta remoção em lote de E' elementos em $O((\frac{E'}{B}+1)log_2 V)$ transferências de*

memória amortizadas. A BPT pode ser construída em $O(sort(E_v))$ transferências de memória. (ARGE et al., 2007)

3.2.1.3 Implementação do Algoritmo de DFS *cache-oblivious*

No Algoritmo 5 é mostrada a versão *cache-oblivious* do procedimento para enumeração em profundidade dos nós de G. Além da BRT T e um conjunto P de *priority queues*, também é usada uma pilha S. Cada $P(v)$ é implementado como uma BPT. A chave de uma aresta (u,v) é u em T e v em $P(v)$.

Inicialmente, são construídas a BRT e as BPTs para cada $u \in V$ a partir das arestas incidentes em u, i.e., $\{(u,v) : v \in V\}$. O topo da pilha v é considerado repetidamente. A cada iteração, são extraídas de `bpt` as arestas com chave v, através da operação `BRTExtrair`(brt, v). Essas arestas são removidas da $bpt[v]$ com a operação `BPTRemove`($bpt[v]$, $arestas\_visitadas$). Se a operação `BPTRemove` também devolve a aresta (v,w) de menor chave entre as que ainda não foram removidas da estrutura, w é empilhado em S e numerado. As arestas incidentes a w $\{(u,w) \forall u \in V, u \neq w\}$ são, então, inseridas na BRT utilizando o procedimento `BRTInsere`($\{(u,w) \forall u \in V, u \neq v\}$). Se a operação `BPTRemove` não devolve nenhum valor, v é desempilhado de S.

Teorema 3.3. *A numeração DFS de uma grafo dirigido pode ser computada cache-oblivious em $O((V + \frac{E}{B}) log_2 V + sort(E))$ transferências de memória. (ARGE et al., 2007)*

3.2.2 Algoritmo de *Tour* Euleriano

Foi desenvolvida uma versão do *tour* euleriano que utiliza uma BRT T como repositório de arestas. O funcionamento é o mesmo da BRT utilizada no algoritmo de busca em profundidade. A diferença é que uma operação *BRTExtrair(T, u)* somente devolve um valor com chave u, em lugar de todos os valores com chave u. A chave de uma aresta (u,v) é u.

Partindo-se de um dado nó u, e uma BRT T, as arestas de G são inseridas na raiz de T. Se a chave de uma aresta (u,v) em T é u, selecionar a próxima aresta do caminho pode-se resumir a uma operação `BRTExtrair(`T`,` u`)`. Assim, a cada extração efetuada, é reportada a aresta que deve-se seguir. Portanto, se `BRTExtrair(`T`,` v`)` devolve (v,w), a próxima aresta

Entrada : um grafo dirigido $G = (V, E)$ e uma *raiz* do percurso
Saída : a enumeração por profundidade dos vértices de G

```
1  início
2      S ← ∅;
3      v ← raiz;
4      brt ← ConstruirBRT(|V|);
5      bpts ← ∅;
6      for v ∈ V do
7          bpts[v] ←ConstruirBPT({(v,w) : w ∈ V});
8      end
9      Empilha(S,v)
10     enquanto a pilha S não é vazia faça
11         v ← Topo(S);
12         arestas_visitadas ← BRTExtrair(brt, v);
13         proxima_aresta ← BPTRemove(bpts[v], arestas_visitadas);
14         se proxima_aresta (v,w) é nula então
15             Desempilha(S);
16         fim
17         senão
18             Empilha(S, w);
19             Numerar(w);
20             BRTInsere({(u,w)∀u ∈ V, u ≠ v});
21         fim
22     fim
23 fim
```

Algoritmo 5: Busca em profundidade *cache-oblivious* (ARGE et al., 2007).

a ser visitada pode ser dada por `BRTExtrair(`T`, `w`)`. Com isso, pode-se calcular o *tour* euleriano de um grafo G em $O(Elog_2(V))$ *cache misses*, como mostrado no Teorema 3.4.

```
Entrada : um grafo G = (V,E) e uma raiz do caminho
Saída   : um tour euleriano em G
1  início
2      T ← ConstruirBRT(|V|);
3      para cada aresta (u,v) de G faça
4          BRTInsere(T, (u,v));
5      fim
6      S ← ∅;
       // R empilha os caracteres do caminho
7      R ← ∅;
8      v ← raiz;
9      aresta_extraida ← ∅;
10     caminho ← ε;
11     faça
12         aresta_extraida ← BRTExtrair(T, v);
13         se aresta_extraida(v,w) não é nula então
14             Empilha(S, v);
15             Empilha(R, Rotulo((v,w)));
16             v ← w;
17         fim
18         senão
19             Concatenar(caminho, Desempilha(R));
20             v ← Desempilha(S);
21         fim
22     enquanto a pilha S não é vazia;
       /* O caminho é obtido na ordem inversa, portanto,
          deve-se invertê-lo para obter o caminho correto
          */
23     devolva Inverte(caminho)
24 fim
```

Algoritmo 6: *Tour* euleriano com BRT.

Teorema 3.4. *A implementação cache-oblivious do algoritmo de tour euleriano em um grafo dirigido $G = (V,E)$ causa $O(Elog_2(V))$ transferências de memória.*

Demonstração. De maneira semelhante à análise do Lema 3.1, os custos de inserção e remoção são $O(\frac{1}{B}log_2V)$ e log_2V, respectivamente. O *array* que armazena o *caminho* é acessado E vezes, o custo total de inserção é $O(\frac{E}{B}log_2V)$. Visto que a extração é efetuada para cada uma das arestas, o total de acessos à memória é $Elog_2V$. A reversão do caminho é feita em $O(\frac{E}{B})$, como discutido

na Seção 1.3. O algoritmo efetua $O(\frac{E}{B}log_2V + Elog_2V + \frac{E}{B})$ acessos de memória. Portanto, o custo total é $O(Elog_2V)$ transferências de memória. □

3.3 Implementação Sucinta

Retomando os conceitos de *rank* e *select*, discutidos na Seção 2.6, é possível definir os procedimentos predecessor e sucessor.

Definição 3.1. Seja uma sequência S de tamanho n e um índice i, tal que $0 \leq i < n$. $pred(c,S,i) = select_c(S, rank(S,i))$ é a primeira ocorrência de c quando a cadeia S é lida a partir da posição i até o começo. Assim, se $S[i]$ é a primeira ocorrência de c em S, então $pred(c,S,i) = 0$.

Definição 3.2. Seja uma sequência S de tamanho n e um índice i, tal que $0 \leq i < n$. $succ(c,S,i) = select_c(S, rank_c(S,i-1)+1)$ é a primeira ocorrência de c quando a cadeia S é lida a partir da posição i até o final. Portanto, se $S[i]$ é a última ocorrência de c em S, $succ(c,S,i) = (n+1)$.

Seja $G = (V,E)$ um grafo de *de Bruijn* k-dimensional de uma sequência S, com alfabeto Σ. Numa representação sucinta de G é possível desenvolver funções de alto nível sobre o grafo a partir das operações predecessor e sucessor, além das funções de rank e select (Definições 2.5 e 2.6) sobre as estruturas de baixo nível *last* e W. Por exemplo, o grau de saída de um nó v com $last[v] = 1$ pode ser computado como mostrado no Algoritmo 7, uma vez que o valor 1 que precede a posição v em *last* delimita o início de todas as arestas que saem do nó v.

Uma função essencial para a navegação em G é dada por `Child`(G, v, c), que devolve o vértice w tal que (v,w) é rotulada por c. Caso não exista tal aresta, o procedimento devolve um valor inválido. Como mostrado no Algoritmo 8, o intervalo (l,r) corresponde às linhas com rótulo igual ao da linha v. A aresta eferente de rótulo c desejada pode encontrar-se em qualquer dessas linhas, seja associada ao caractere c, seja ao caractere estendido $\bar{c}$. A sua posição exata p, se existir, é determinada com auxílio da função predecessor. Em seguida, a posição w é determinada a partir de p conforme explicado no final da Seção 2.6.1.

Entrada : um índice v tal que $last[v] = 1$ e $Node[v]$ é o rótulo do vértice
Saída : grau de saída do nó v

```
1 início
2 |  devolva v − Pred(last, v-1)
3 fim
```

Algoritmo 7: *cdeg*.

Entrada : um vértice v de G e um caractere $c \in \mathscr{A}$
Saída : o vértice w filho de v através da aresta c, ou um vértice inválido, caso esta aresta (v,w) não exista em G

```
1  início
2  |  (l,r) ← (Pred(1, last, v),v);
3  |  p ← Pred(c, W, v);
4  |  se l ≤ p ≤ r então
5  |  |  r ← Rank(c, W, p);
6  |  |  w ← Select(1, last, F[c]+r);
7  |  |  devolva w;
8  |  fim
9  |  p ← Pred(c̄,W,v);
10 |  se l ≤ p ≤ r então
11 |  |  p ← Pred(c, W, p);
12 |  |  r ←= Rank(c, W, p);
13 |  |  w ← Select(1, last, F[c]+r);
14 |  |  devolva w;
15 |  fim
16 |  devolva n+1
17 fim
```

Algoritmo 8: *child*.

3.3.1 Implementação de *W* e *last*

Uma vez *W* e *last* definidos como *strings*, é possível, então, utilizar estruturas sucintas que suportem as operações de *rank* e *select* necessárias à implementação das funções de mais alto nível. Visto que *last* é uma *string* formada somente pelo alfabeto binário $\{0,1\}$, é possível utilizar uma estrutura mais simples para sua implementação. Neste caso, foi utilizada a implementação do *bit vector* descrita por NAVARRO; PROVIDEL (2012).

Para a implementação de W, entretanto, é necessária uma estrutura de dados mais complexa, visto que o alfabeto da cadeia de entrada, em tese, é arbitrário, ainda que conheça-se *a priori* o alfabeto de entrada em um programa de montagem de fragmentos de DNA, seja o $\Sigma = \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$. Uma estrutura sucinta capaz de generalizar operações de *rank* e *select* de vetores binários para alfabetos arbitrários é a *wavelet tree* (GROSSI; GUPTA; VITTER, 2003), que foi então utilizada para implementar W.

Dado que a *wavelet tree* é uma árvore, optou-se também por utilizar o *layout* de *van Emde Boas* para armazenar essa árvore. Considerando que W é visitado sucessivas vezes, a localidade de memória pode ser um elemento importante na melhora do desempenho geral do algoritmo. O *layout* de *van Emde Boas* torna possível estabelecer um limite assintótico para o número de *cache misses* incorridos no acesso à estrutura.

3.3.2 *Wavelet Trees*

As *wavelet trees*, definidas por GROSSI; GUPTA; VITTER (2003), são estruturas de dados capazes de representar de maneira sucinta uma cadeia de caracteres $S = s_1, s_2 \ldots s_t$ com tamanho t e alfabeto arbitrário Σ. Uma *wavelet tree* para representar S é formada por um árvore binária balanceada de altura $\sigma = \log|\Sigma|$ que, a cada ramificação, particiona recursivamente o alfabeto Σ entre os nós filhos, até que, nas folhas, somente existam uma ou duas letras.

O objetivo é codificar a S como uma cadeias formadas pelo alfabeto binário $\{0,1\}$. Uma vez que a estrutura tem como objetivo representar um texto com um alfabeto arbitrário, é necessário desfazer a ambiguidade da representação binária. Assim, a cada ramificação, o alfabeto correspondente a um nó-pai é dividido ao meio entre os nós-filhos. No nó-pai, os

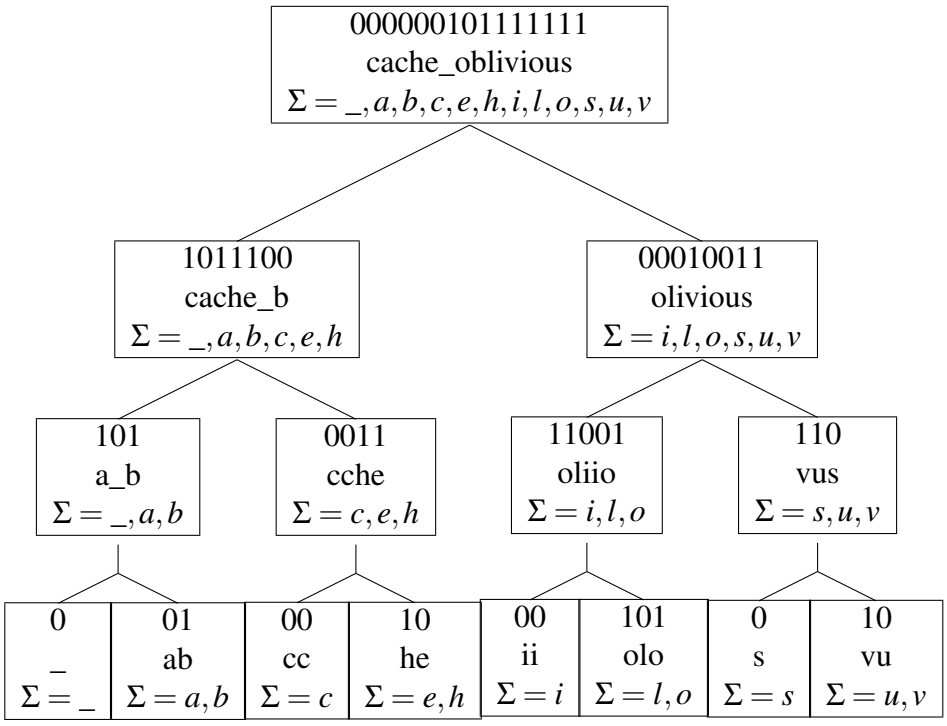


Figura 3.2: Uma wavelet tree.

caracteres correspondentes ao filho da esquerda são codificados como 0, e os da direita como 1. Em cada nó, somente são codificados os caracteres de S contidos no trecho do alfabeto correspondente àquele nó. Assim, a cadeia seja codificada como um vetor de bits, é possível saber a posição de um caractere arbitrário de Σ em S, já que a posição relativa dos caracteres é mantida. Isso torna possível a generalização das operações de *rank* e *select* aplicadas a vetores de bits a alfabetos arbitrários.

Na Figura 3.2 é mostrada a estrutura de uma *wavelet tree*. Na prática, o somente são armazenados os vetores de bits e o intervalo de Σ correspondente a cada nó.

3.3.3 Árvores de *van Emde Boas* (vEB)

Um dos problemas no uso de estruturas baseadas em ponteiros, como listas encadeadas, árvores ou grafos, é a falta de localidade de memória das unidades da estrutura. Assim, o percurso feito entre um nó e outro pode levar a um bloco de memória distinto, o que força uma nova consulta da memória, caso o bloco seguinte ainda não tenha sido carregado na memória cache.

A representação de *van Emde Boas* (van Emde Boas, 1975) é uma estratégia recursiva

para armazenamento de árvores. O objetivo é aumentar a localidade de memória, fazendo com que nós visitados em sequência na árvore sejam armazenados o mais possível em uma mesma região (SILVA, 2016). Assim, é possível estabelecer um limite superior para o número de cache misses ocorridos no uso dessa estrutura. Embora esta seja uma estratégia *cache-oblivious*, os algoritmos aplicados a árvores armazenadas de acordo com a representação de *van Emde Boas* não necessariamente devem ser *cache-oblivious*.

A estratégia consiste dividir a árvore T com altura h no nível intermediário das arestas. As subárvores resultantes são uma árvore superior e aproximadamente $\sqrt{N}$ árvores inferiores, cada qual tamanho próximo de $\sqrt{N}$. A subárvore superior deve ser armazenada recursivamente, seguida pelas $\sqrt{N}$ árvores inferiores, utilizando o mesmo layout, até que somente deva ser armazenado um único nó.

Em árvores que não têm altura que é uma potência de 2, essa altura deve ser $\lceil \lceil h/2 \rceil \rceil$. Neste caso, a árvore superior tem altura $h - \lceil \lceil h/2 \rceil \rceil$. Caso essa altura também não seja uma potência de 2, aplica-se o mesmo procedimento de arredondamento.

3.3.4 Algoritmo de DFS

A implementação do algoritmo de busca em profundidade desenvolvida para a versão sucinta do grafo é descrita no Algoritmo 9. O algoritmo é o mesmo utilizado na versão canônica. A diferença é dada pela maneira como é obtido um nó vizinho, o que é feito pelo procedimento `Child`.

3.3.5 Algoritmo de *Tour* Euleriano

A exemplo do que ocorreu com a implementação da DFS numa representação sucinta do grafo de *de Bruijn*, a implementação do *tour* euleriano para a versão sucinta do grafo de *de Bruijn*, mostrada no Algoritmo 10, tem a mesma estrutura do procedimento canônico. O procedimento `CdegLbl`(G, v, c) devolve o número de arestas de saída de v rotuladas pelo caractere c. *visitados* é um *array* que indica o número de vizinhos já visitados em cada vértice do grafo. Verifica-se se todos os vizinhos do nó v já foram visitados. Caso sim, desempilha-se um vértice de S e o o caractere correspondente em R é concatenado a *caminho*. Caso ainda haja

```
Entrada : um grafo G = (V,E) com representação sucinta e uma raiz da busca
Saída   : numeração dos vértices de G por visita em profundidade
1  início
2      visitado ← ∅;
3      S ← ∅;
4      Empilha(S, raiz);
5      enquanto S não é vazia faça
6          v ← Desempilha(S);
7          se visitado[v] é diferente de 1 então
8              visitado[v] ← 1;
9              para cada caractere c de Σ faça
10                 se u ←Child(G, v, c) é um vértice de V então
11                     Empilha(S, v);
12                     v ← u;
13                 fim
14             fim
15         fim
16     fim
17 fim
```

Algoritmo 9: Algoritmo de busca em profundidade na representação sucinta do grafo.

arestas incidentes a visitar, verifica-se, em ordem, se algum dos vizinhos `A`, `C`, `G` ou `T` pode ser visitado, considerando o número de visitas já realizadas, $visitas[v]$. O vizinho escolhido é empilhado em R e o vértice atual passa a ser `Child`(G, v, c).

Entrada : um grafo $G = (V, E)$ com representação sucinta e uma *raiz* do caminho
Saída : um *tour* euleriano em G

```
1  início
2      S ← ∅;
3      R ← ∅;
4      v ← raiz;
5      caminho ← ε;
6      enquanto a pilha S não é vazia faça
7          se visitas[v] == Cdeg(G, v) então
8              Concatenar(caminho, Desempilha(R));
9              v ← Desempilha(S);
10             continue;
11         fim
12         total ← 0;
13         para cada caractere c de Σ faça
14             u ←Child(G, v, c);
15             se u é um vértice de V então
16                 total ← total + CdegLbl(G, v, c);
17                 se visitas[v] < total então
18                     Empilha(S, v);
19                     Empilha(R, c);
20                     visitas[v] ← visitas[v] + 1;
21                     v ← u;
22                     pare;
23                 fim
24             fim
25         fim
26     fim
       /* O caminho é obtido na ordem inversa, portanto,
          deve-se invertê-lo para obter o caminho correto
          */
27     devolva Inverte(caminho)
28 fim
```

Algoritmo 10: *Tour* euleriano na versão sucinta do grafo de *de Bruijn*.

4

Resultados

Com o objetivo de avaliar empiricamente os algoritmos e estruturas de dados que foram implementados, foram efetuados experimentos envolvendo as três representações do grafo de *de Bruijn*, nomeadamente a versão canônica, a *cache-oblivious* e a sucinta, durante a execução dos algoritmos DFS e *tour* euleriano. Foram utilizados dados públicos e a eficiência da utilização da cache foi aferida com o auxílio de ferramenta específica, conforme o descrito a seguir.

4.1 Ambiente Experimental

4.1.1 Dados

Os dados foram coletados a partir do arquivo de sequências de DNA separadas por linha com 50MB obtido no site Pizza&Chilli (http://pizzachili.dcc.uchile.cl/texts/dna/dna.50MB.gz). O arquivo é composto de sequências de DNA obtidas nos arquivos 01hgp10 a 21hgp10 e 0xhgp10 a 0yhgp10 do Projeto Gutenberg (http://www.gutenberg.org/). Para o uso nos testes, foram removidos todos os caracteres de espaço do arquivo.

4.1.2 *Cachegrind*

O *cachegrind* é uma ferramenta do *software Valgrind* (SEWARD et al., 2016) que realiza simulação de uma hierarquia de memória com cache através de instrumentação de código. Assim,

são obtidos informações sobre o número de acessos de leitura e escrita das caches de dados e instruções, bem como o número de *cache misses* ocorridos no programa. Também é possível realizar a anotação da saída do *cachegrind*, de maneira a obter informações em um nível mais baixo, como a cada função ou mesmo a cada linha de código.

O *cachegrind* obtém as informações sobre a cache do computador onde o teste é efetuado e utiliza esses parâmetros na simulação, entretanto, somente é possível simular, no máximo, dois níveis de cache. Portanto, caso o computador tenha três ou mais níveis de cache, o *cachegrind* somente simula a cache do primeiro e do último níveis, chamadas L1 e LL, respectivamente.

4.1.3 Máquina

Os experimentos foram executados em um computador portátil com processador Intel®Core i5, com 6GB de memória RAM com em um sistema operacional Linux. A memória cache é formada por três níveis. O primeiro nível tem 32 kB, bloco de 64 B e é 8-*way* associativa e a de terceiro nível tem 3 MB, bloco de 64 B e é 12-*way* associativa. Estes foram também os parâmetros usados na simulação do *cachegrind*.

4.2 Experimentos

Foram efetuados experimentos para os algoritmos DFS e *tour* euleriano com representações canônica, *cache-oblivious* e sucinta do grafo de *de Bruijn*. Os casos compreendem as execuções com k igual a 5, 10, 15, 20, 25 e 30, com entradas da tamanho 10000 a 100000 em cada caso. Os tamanhos das sequências foram determinados devido à limitação de tempo com entradas maiores, principalmente na implementação sucinta. Cada uma das execuções foi feita três vezes, com três sequências distintas obtidas a partir das posições 0, 10000000 e 20000000 do arquivo de entrada, e foi obtida a média dessas três execuções. Todas as execuções foram feitas com o valgrind com a ferramenta *cachegrind*, configurando a exibição do número de acessos de leitura e escrita à memória (D), o número de cache misses de leitura e escrita na cache de nível 1 (D1) e o número de cache misses de leitura e escrita na cache do último nível (DL). A cache L1 é menor em relação às demais, de maneira que o número de cache misses naturalmente

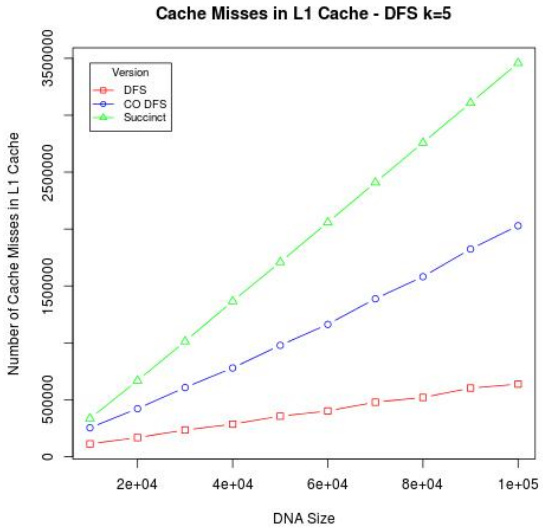
é maior. O arquivo de saída foi anotado, utilizado-se o comando *cg_annotate*. Neste arquivo, foram coletados os parâmetros Dr, Dw, D1mr, D1mw, Dlmr e Dlmw, correspondentes às leituras, escritas, cache misses de leitura em L1, cache misses de escrita em L1, cache misses de leitura em DL e cache misses de escrita em Dl, respectivamente. Também foi calculado o *miss ratio* nas caches L1 e LL, dado por $L1 = \frac{D1mr+D1mw}{Dr+Dw}$ e $LL = \frac{Dlmr+Dlmw}{Dr+Dw}$.

4.2.1 Cache Misses

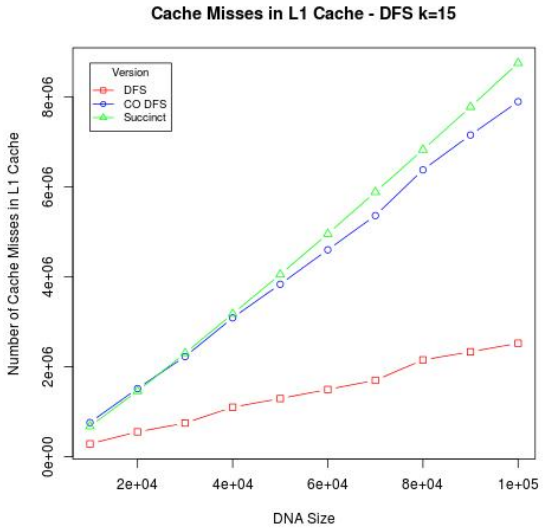
Nos primeiros experimentos realizados, foi contabilizado o número de cache misses causado pela execução de cada um dos programas. Os resultados são relativos à execução do programa inteiro em cada uma das versões, incluindo as etapas de construção das estruturas de dados utilizadas e a execução dos algoritmos propriamente dita. Isso ocorreu por dois motivos: o primeiro, que o *cachegrind* possibilita somente a obtenção do número de cache misses de cada função isoladamente, e o segundo, para tornar a comparação mais justa, já que as versões alternativas (*cache-oblivious* e sucinta) dependem de estruturas adicionais, o que aumenta a complexidade geral do problema.

4.2.1.1 Algoritmo DFS

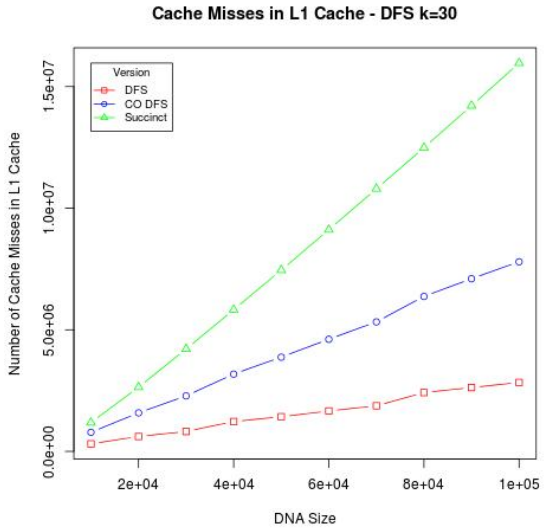
Nas Figuras 4.1 e na 4.2 são exibidos os resultados para as caches L1 e LL, respectivamente, para a execução da DFS com as três representações do grafo de *de Bruijn* para valores de k variando de 5 a 30. Em cada gráfico, os valores são exibidos para um k fixo e para uma sequência S de tamanho variando entre 10kb e 100kb.



(a) Grafo 5-dimensional.

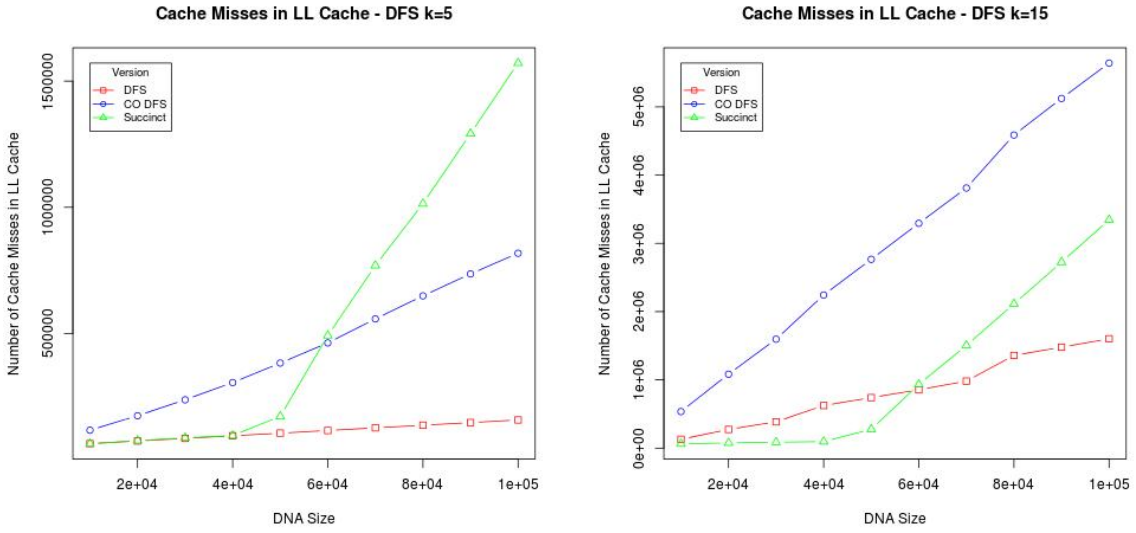


(b) Grafo 15-dimensional.



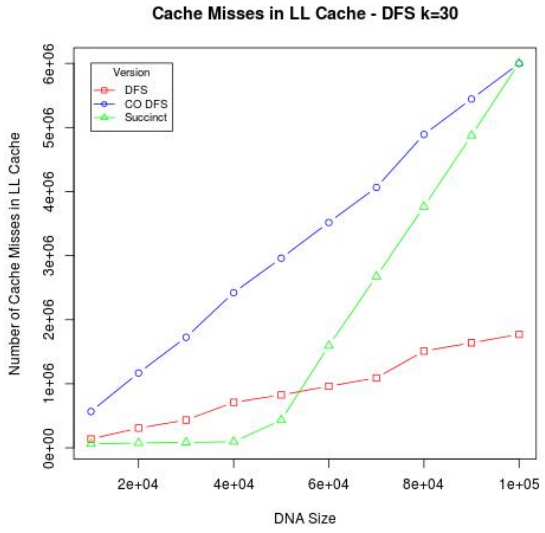
(c) Grafo 30-dimensional.

Figura 4.1: Cache misses na Cache L1.



(a) Grafo 5-dimensional.

(b) Grafo 15-dimensional.



(c) Grafo 30-dimensional.

Figura 4.2: Cache misses na Cache LL.

Observa-se quanto ao número de cache misses na cache L1 que o algoritmo de DFS canônico tem o melhor desempenho quando comparado com as demais abordagens. Além disso, observa-se que o o comportamento da curva em relação ao crescimento da sequência de entrada é aproximadamente o mesmo, independentemente do tamanho de k. Um comportamento semelhante foi observado na cache LL.

A versão *cache-oblivious* do algoritmo de DFS tem, em geral, um desempenho intermédio entre as demais. A curva tem um crescimento linear em relação ao tamanho da sequência de

entrada, o que pode ocorrer já que, como discutido anteriormente, um tamanho de k maior faz com que grande parte do grafo seja linear, já que a probabilidade de ocorrência de uma dada k-mer mais de uma vez diminui.

O comportamento da curva da versão cache-oblivious é semelhante na cache LL e o crescimento é aproximadamente linear. Entretanto, nota-se que o desempenho em termos de número absoluto de cache misses é pior em todos os casos com $k > 5$.

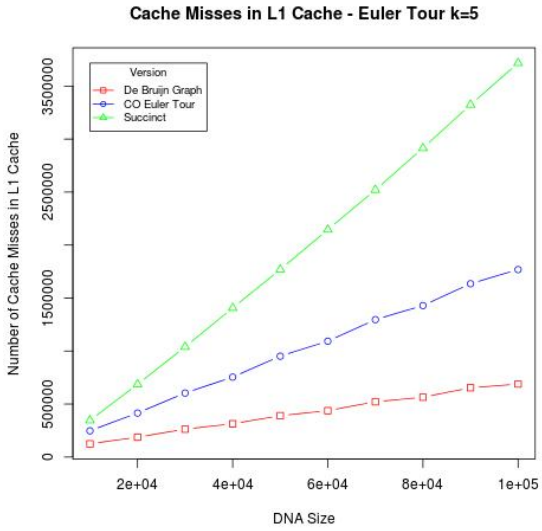
A versão sucinta tem um desempenho pior na maior parte dos casos. O comportamento da curva é linear em todos os casos em relação ao crescimento da sequência de entrada. Na Figura 4.1(b) observa-se que as versões *cache-oblivious* e sucinta têm um resultado semelhante, entretanto, os valores de cache misses crescem mais rapidamente para a versão sucinta com o aumento de k.

A versão sucinta foi a que apresentou a maior diferencça de comportamento entre as análises na cache L1 e LL. Diferentemente do primeiro caso, na cache LL a versão sucinta apresentou um comportamento melhor ou muito próximo da versão canônica com um sequência de entrada com tamanho ≤ 40000, com um crescimento mais lento do número de cache misses em relação ao tamanho da sequência de entrada. Com uma entrada de tamanho > 40000, o desempenho é semelhante ao observado na cache L1, com um crescimento rápido e linear.

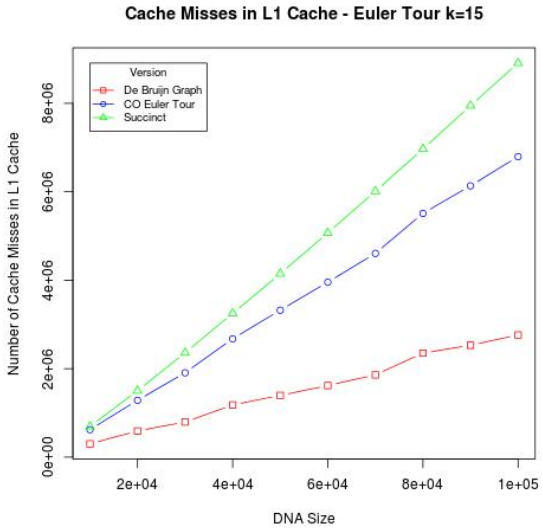
Isso pode ser resultado do menor espaço ocupado por essa representação, o que faz com que a maior parte do grafo possa ser acomodada na cache, à medida que esta aumenta.

4.2.1.2 Algoritmo de *Tour* Euleriano

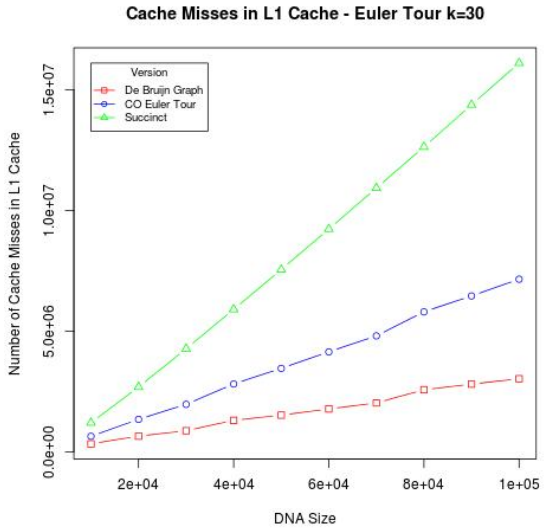
Nas Figuras 4.3 e 4.4 são exibidas os resultados para as caches L1 e LL, respectivamente, para a execução do *tour* euleriano com as três representações do grafo de *de Bruijn* para valores de k variando de 5 a 30. Em cada gráfico os valores são exibidos para um k fixo e para uma sequência S de tamanho variando entre 10kb e 100kb.



(a) Grafo 5-dimensional.

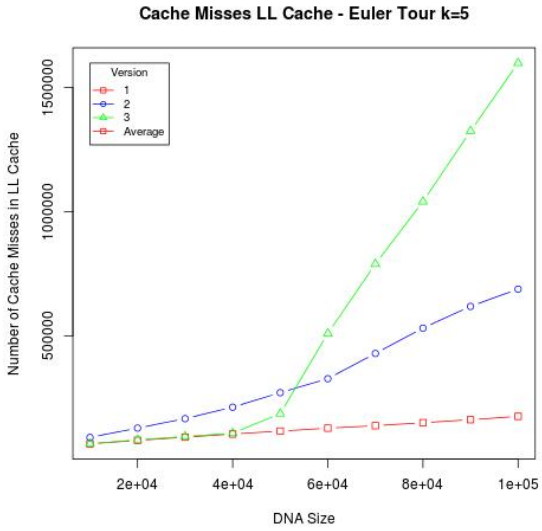


(b) Grafo 15-dimensional.

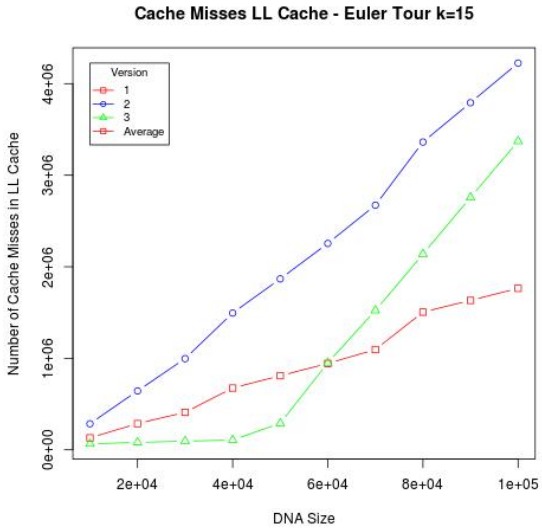


(c) Grafo 30-dimensional.

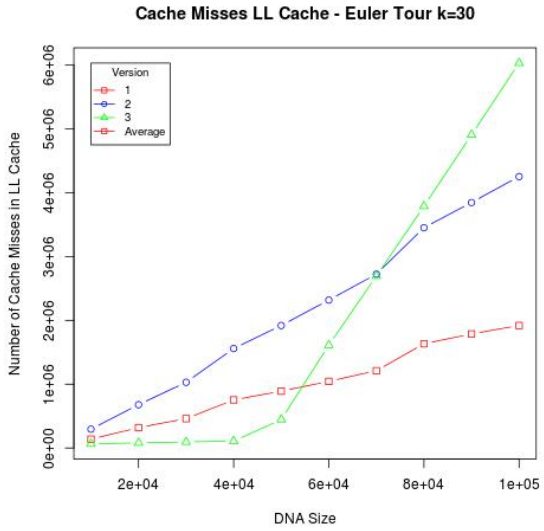
Figura 4.3: Cache misses na Cache L1.



(a) Grafo 5-dimensional.



(b) Grafo 15-dimensional.



(c) Grafo 30-dimensional.

Figura 4.4: Cache misses na Cache LL.

Como mostrado na Figura 4.3, o desempenho do *tour* euleriano quantos aos cache misses na L1 é semelhante ao que ocorre no algoritmo de DFS. Assim como ocorrido na DFS, o desempenho da versão *cache-oblivious* foi o pior entre todas as versões, contudo, com um comportamento linear neste caso. Outra diferença foram os resultados da implementação sucinta, que teve um resultado próximo à versão *cache-oblivious*, tendo crescido mais rapidamente em relação ao aumento da sequência de entrada do que o observado nos testes com a versão sucinta do DFS. Como observado na Figura 4.4, uma relação semelhante ocorre entre os cache misses

da cache LL para o DFS e o *tour* euleriano, apesar de a curva da versão sucinta crescer mais rapidamente.

A partir do observado, nota-se que, conforme esperado, o valor de k e o tamanho da sequência de entrada S têm impacto sobre o número de cache misses nas cache L1 e LL. Como discutido anteriormente, se um k é pequeno, a probabilidade de repetição de uma mesma k-mer na sequência S é maior, quanto mais S seja grande. Com $k = 5$, por exemplo, existem 1024 k-mers distintas. Assim, se S tem somente 10000 bases, cada k-mer ocorreria aproximadamente 10 vezes, supondo-se que todas as k-mers têm igual probabilidade de ocorrer.

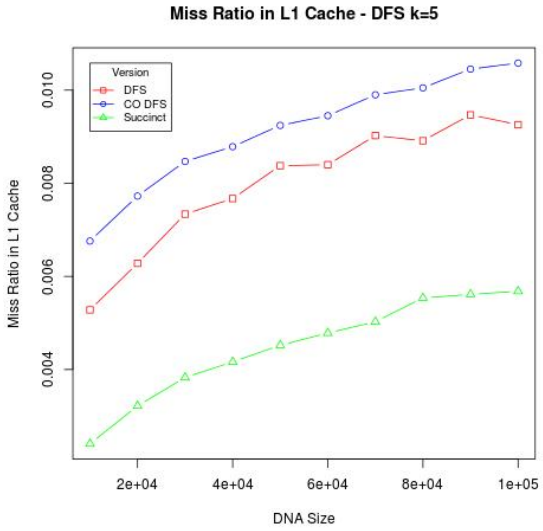
4.2.2 Experimento *Miss Ratio*

Embora o uso da cache em termos absolutos seja uma boa métrica, a comparação entre implementações distintas pode ser enviesada, já que em alguns casos são necessárias estruturas de dados auxiliares ou algoritmos com uma abordagem distinta da canônica. Assim, uma análise alternativa é a observação do aproveitamento da cache em relação ao número total de acessos à memória, o que pode ser dado pelo *miss ratio*.

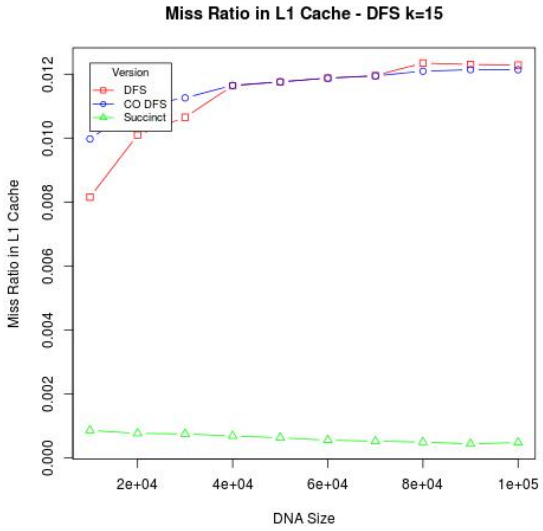
Assim sendo, foram realizados testes para analisar o aproveitamento da memória cache pelas estruturas de dados utilizadas em cada versão dos algoritmos de DFS e *tour* euleriano. Quanto menor o *miss ratio*, melhor é o aproveitamento da memória cache.

4.2.2.1 Algoritmo DFS

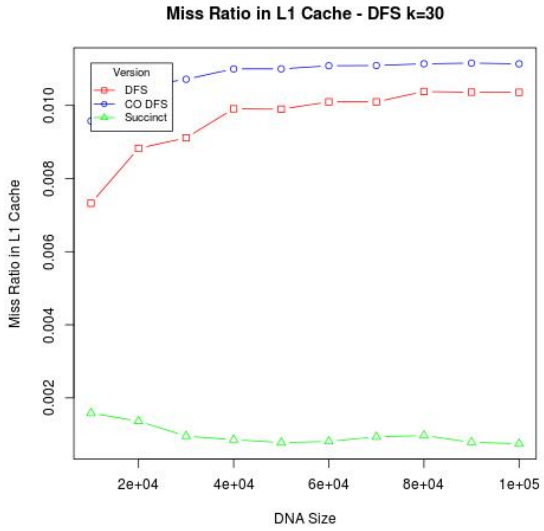
Nas Figuras 4.5 e na 4.6 são exibidos os resultados de *miss ratio* para as caches L1 e LL, respectivamente, para a execução da DFS com as três representações do grafo de *de Bruijn* para valores de k variando de 5 a 30. Em cada gráfico, os valores são exibidos para um k fixo e para uma sequência S de tamanho variando entre 10kb e 100kb.



(a) Grafo 5-dimensional.

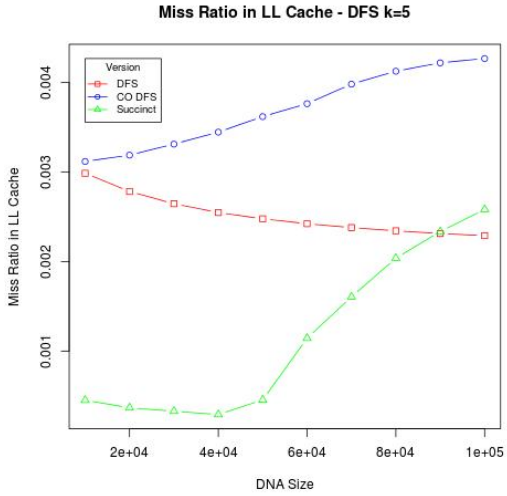


(b) Grafo 15-dimensional.

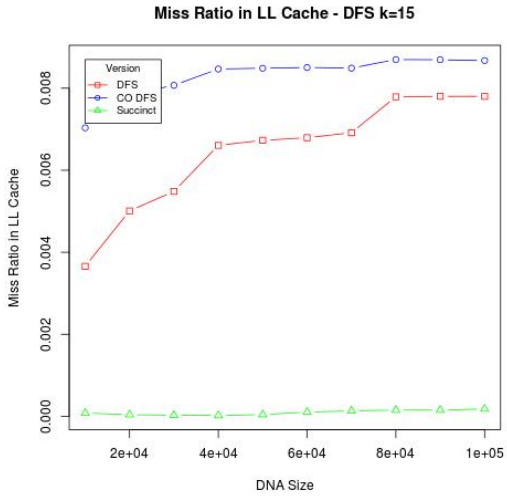


(c) Grafo 30-dimensional.

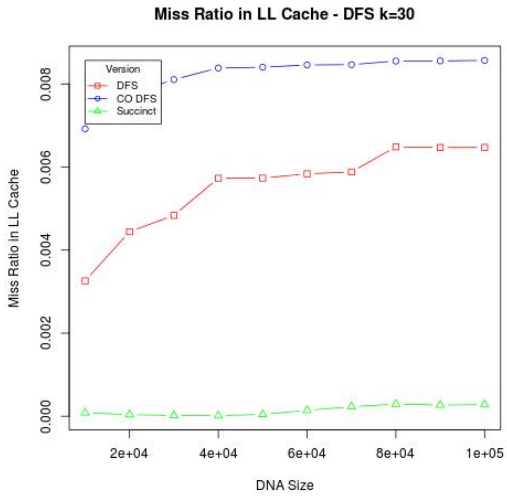
Figura 4.5: *Miss ratio* na Cache L1.



(a) Grafo 5-dimensional.



(b) Grafo 15-dimensional.



(c) Grafo 30-dimensional.

Figura 4.6: *Miss ratio* na Cache LL.

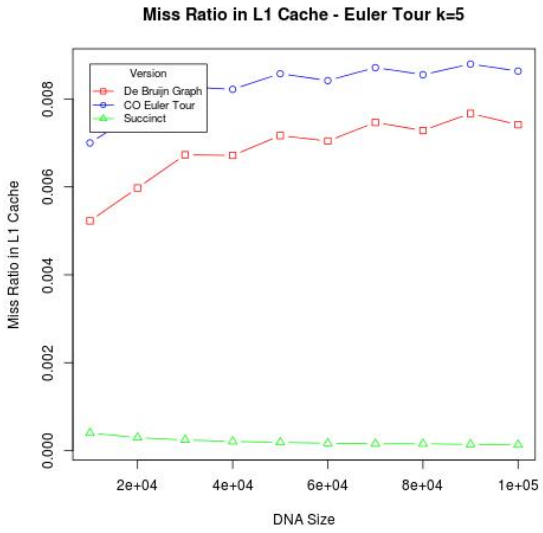
Os resultados obtidos quanto ao *miss ratio* na cache L1 mostra que a versão sucinta do DFS tem um aproveitamento muito maior que os demais casos, e mesmo uma tendência decrescente com o crescimento da sequência de entrada. A versão *cache-oblivious* tem um desempenho pior até $k = 10$. Com $k > 10$, o desempenho das versões *cache-oblivious* e canônica são comparáveis e com tendência crescente, com uma piora mais rápida na versão canônica.

Na cache LL, o comportamento observado é bastante semelhante ao observado na cache L1 com $k \geq 10$. Com $k = 5$, entretanto, notou-se um comportamento semelhante ao observado nos valores absolutos de cache misses na cache LL nos experimentos realizados com DFS e *tour* euleriano, que consiste em um crescimento rápido do número de cache misses com uma sequência de tamanho maior que 40000 bases. Com a análise do *miss ratio* é possível concluir que, apesar de ter um desempenho pior em termos absolutos, a implementação sucinta tem um aproveitamento maior da cache em relação às demais versões.

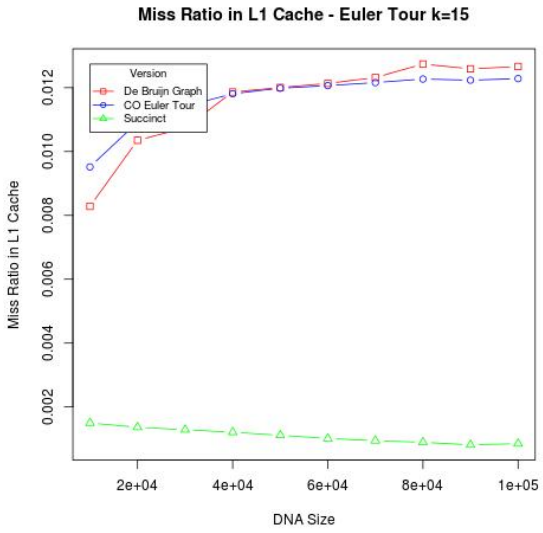
4.2.2.2 Algoritmo de *Tour* Euleriano

Nas Figuras 4.7 e 4.8 são exibidos os resultados de *miss ratio* para as caches L1 e LL, respectivamente, para a execução do *tour* euleriano com as três representações do grafo de *de Bruijn* para valores de k variando de 5 a 30. Em cada gráfico, os valores são exibidos para um k fixo e para uma sequência S de tamanho variando entre 10kb e 100kb.

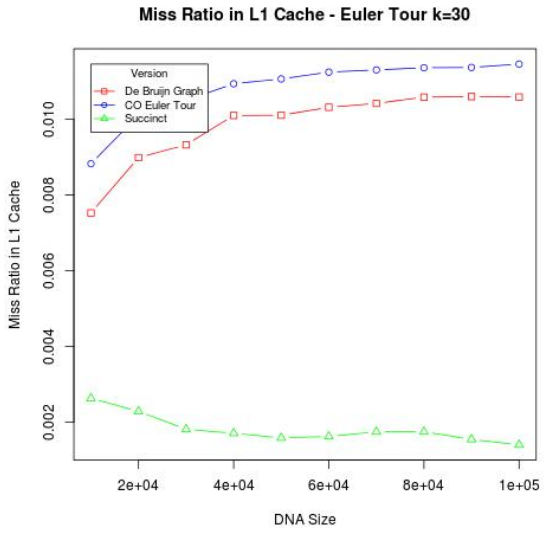
O desempenho observado em termos de *miss ratio* no teste das implementações de *tour* euleriano tiveram um comportamento bastante parecido com o observado na análise das implementações do algoritmo de DFS. No *tour* euleriano, entretanto, a implementação sucinta teve o melhor desempenho em todos os casos. Além disso, a versão canônica chega a ser pior que a versão *cache-oblivious* em alguns momentos, variando de acordo com o valor de k e o tamanho da cadeia.



(a) Grafo 5-dimensional.

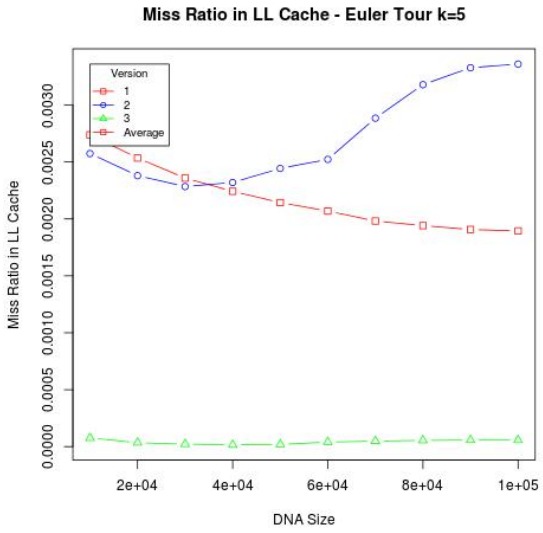


(b) Grafo 15-dimensional.

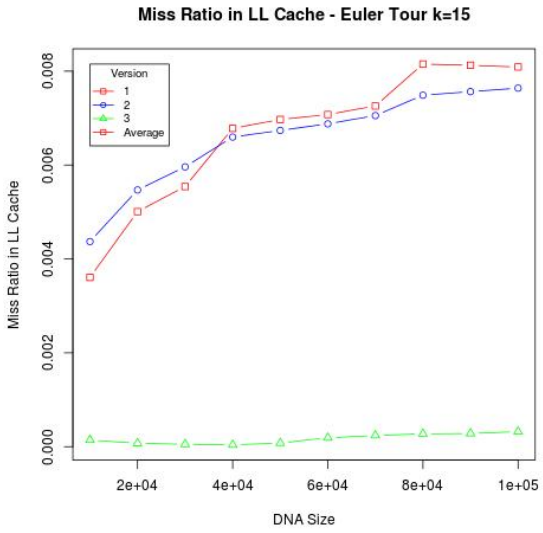


(c) Grafo 30-dimensional.

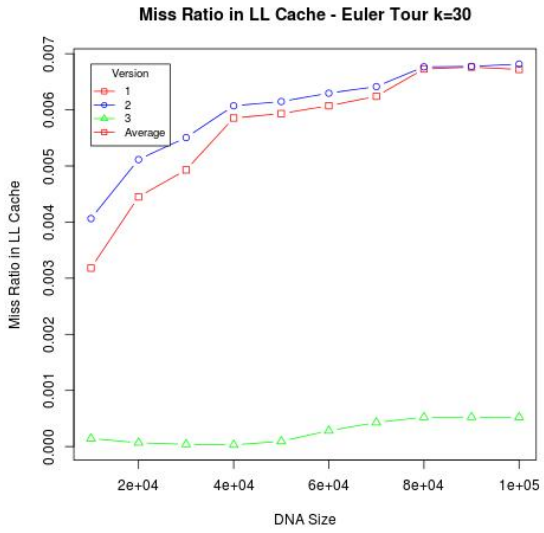
Figura 4.7: *Miss ratio* na Cache L1.



(a) Grafo 5-dimensional.



(b) Grafo 15-dimensional.



(c) Grafo 30-dimensional.

Figura 4.8: *Miss ratio* na Cache LL.

5

Conclusão

O objetivo deste trabalho é estudar o problema do sequenciamento genômico baseado em grafos de *de Bruijn* do ponto de vista de utilização de memória. Com efeito, as técnicas baseadas em gbD representam o estado da arte para o problema da montagem de fragmentos, sendo utilizadas em diversas ferramentas atuais, sobretudo com o advento das novas tecnologias de sequenciamento de alto rendimento. Nessas técnicas, a quantidade de sequência produzida é muito volumosa, o que muitas vezes inviabiliza a adoção de técnicas baseadas em grafos de sobreposição como na estratégia *overlap-layout-consensus*.

A etapa crítica na montagem de fragmentos utilizando grafos de *de Bruijn* corresponde, essencialmente, a encontrar um *tour* euleriano no grafo formado a partir das k-mers encontradas nos fragmentos sequenciados. Embora seja um problema de solução teórica polinomial, o tamanho do grafo resultante pode alcançar centenas de *gigabytes*, o que pode representar um gargalo. Diante deste cenário, faz-se necessário o estudo de alternativas eficientes para reduzir o espaço total utilizado pelo grafo e/ou otimizar a localidade espacial e temporal da informação durante a execução dos algoritmos. Alguns desenvolvimentos teóricos recentes no campo da algoritmia representam alternativas atrativas para abordar esse problema.

Os algoritmos e estruturas de dados cache oblivious vem sendo propostos como uma alternativa aos algoritmos "tradicionais" em memória externa, muitas vezes igualando a complexidade assintótica teórica dos seus correspondentes no modelo I/O, sem todavia requerer operações primitivas explícitas para movimentação de blocos de memória, nem parametrização quanto a

características físicas específicas da memória. Esses algoritmos organizam implicitamente a informação e a ordem das operações de forma a otimizar o uso da memória cache, normalmente ordens de magnitude mais rápidas do que os níveis superiores de memória (RAM, disco, etc.) Entretanto, embora haja na bibliografia algumas abordagens *cache-oblivious* para algoritmos e estruturas de dados para grafos, existe muito pouca disponibilidade de implementações reais ou trabalhos relatando seu uso em situações práticas.

Uma outra possibilidade que vem ganhando popularidade nos últimos anos é a adoção de estruturas de dados ditas sucintas, capazes de representar a informação em espaço muito próximo ao limite teórico, apenas com uma sobrecarga sublinear de bits, idealmente sem prejuízo à complexidade teórica das operações suportadas, em comparação com representações mais explícitas. Com essas estruturas, é naturalmente possível representar grafos maiores no mesmo espaço de memória. Entretanto, em função da maneira indireta segundo a qual os dados são representados, normalmente as operações são igualmente efetuadas de maneira indireta em muitos passos, o que representa também mais acessos à memória.

Neste trabalho foram, portanto, examinadas três implementações dos gdB com o objetivo de avaliar o seu comportamento quanto à utilização da cache, em particular durante a execução dos algoritmos DFS e *tour* euleriano, dadas as suas importâncias para o problema da montagem de fragmentos. De maneira geral, os resultados experimentais obtidos mostraram que a versão canônica, que usa estruturas de dados mais explícitas e custosas do ponto de vista de memória são também mais eficientes do ponto de vista do tempo de execução em praticamente todos os casos avaliados. Isso deveu-se ao fato de que as operações envolvem muito menos instruções primitivas e, portanto, acessos à memória.

Observou-se nos experimentos que, embora, do ponto de vista assintótico, as versões *cache-oblivious* usem a memória cache de maneira ótima, as técnicas necessárias causam uma sobrecarga que torna essas implementações mais custosas. Além disso, o modelo subjacente da cache ideal está baseado em premissas que não necessariamente se verificam na prática, em particular a associatividade total da cache, segundo a qual qualquer linha da cache pode ser utilizada por qualquer bloco a qualquer instante, independente da sua posição em níveis hierárquicos superiores de memória. Algumas constantes desconsideradas na análise assintótica

podem ter impacto no desempenho da implementação real.

Conforme previsto, a implementação sucinta requer uma quantidade muito maior de operações primitivas, tipicamente do tipo *rank* e *select*, que implicam cada uma em alguns acessos à memória em posições não necessariamente contíguas. Portanto, o tempo de execução e a quantidade absoluta de cache misses são muito maiores nesse caso.

Apesar de ser a menos eficiente em termos absolutos, uma característica notável da implementação sucinta é que ela apresenta o melhor desempenho relativo, medido como a razão entre o número de cache misses e o número total de acessos à cache. Essa representação foi a única que apresentou uma tendência negativa nesse aspecto, ou seja, à medida que os dados tornam-se mais volumosos, diminui proporcionalmente a necessidade de transferências entre níveis sucessivos de memória. Essa observação permite supor que, em casos mais extremos, o tempo acrescido de execução poderia ser compensado pela utilização mais eficiente da cache.

5.1 Trabalhos Futuros

Além dos fatores específicos analisados neste trabalho, naturalmente também é possível e necessário ampliar as observações para tentar estabelecer empiricamente, por exemplo, a relação efetiva entre o aproveitamento de uso de memória e o tempo de execução de um algoritmo. Também se poderia estender os testes de forma a analisar o aproveitamento do nível superior de memória, ou seja, a interface entre a memória principal (RAM) e a secundária (Disco), o que significaria ampliar os dados e ferramentas utilizadas. Também se poderia investigar os mesmos efeitos em arquiteturas distintas, de modo a aferir o impacto de diferentes configurações e otimizações de *hardware* sobre o problema estudado.

Além disso, o *tour* euleriano aqui estudado é apenas um modelo do problema real de montagem de fragmentos. Na prática, as ferramentas executam diversas outras tarefas de pré e pós-processamento sobre o grafo de *de Bruijn* para tratar problemas como erros de sequenciamento ou a existência de repetições. Assim sendo, o estudo pode ser ampliado para situações mais próximas à realidade.

Apesar do uso da memória cache ser um fator importante para o desempenho final dos algoritmos, na prática, a sobrecarga imposta pela utilização de estruturas e estratégias

"preventivas" *cache-oblivious* acabam por mitigar o eventual ganho no tempo devido ao menor número de transferências de memória e impedir que ele se revele nas condições usuais de utilização, portanto distante dos limites assintóticos. As estruturas sucintas, por sua vez, não se preocupam diretamente com a gestão da cache ou com a localidade dos dados. Ao invés disso, a preocupação é simplesmente representar os dados no menor espaço possível sem comprometer o custo assintótico das operações, confiando nas estratégias genéricas de paginação de memória oferecidas pelas camadas inferiores como o sistema operacional e o *hardware*. Os dados obtidos parecem apontar que essa estratégia pode ser promissora, sobretudo se combinada com algumas intervenções pontuais, a exemplo da organização *van Emde Boas* usada na implementação proposta para os grafos de *de Bruijn*.

Relativamente ao encontrado durante a etapa de Revisão Sistemática, é interessante incluir nesse estudo implementações baseadas em compressão, filtros de *Bloom* e BWT, já que existem trabalhos que utilizam estas estruturas. Além disso, existem também abordagens sucintas aplicadas especificamente à etapa de construção do grafo, o que também pode contribuir para o custo total do processo. Uma abordagem publicada recentemenete que também pode ser objeto de um estudo semelhante ao realizado neste trabalho é MILICCHIO et al. (2016), a qual utiliza estruturas *cache-oblivious* denominadas *funnels* para representar os grafos de *de Bruijn*, em contrapartida às BRTs utilizadas nesse trabalho, as quais também são estruturas *cache-oblivious*.

Referências

AGGARWAL, A.; VITTER JEFFREY, S. The input/output complexity of sorting and related problems. **Communications of the ACM**, [S.l.], v.31, n.9, p.1116–1127, 1988.

ALBERTS, B. et al. **Molecular Biology of the Cell**. 6th.ed. New York and London: Garland Science, 2014. 1464p.

ALLULLI, L. **Cache Oblivious Computation of Shortest Paths : theoretical and practical issues**. 2007. 102p. Thesis — Università degli Studi di Roma "La Sapienza".

ARGE, L. The buffer tree: a technique for designing batched external data structures. **Algorithmica (New York)**, [S.l.], v.37, n.1, p.1–24, 2003.

ARGE, L. et al. Cache-oblivious priority queue and graph algorithm applications. **Proceedings of the thiry-fourth annual ACM symposium on Theory of computing - STOC '02**, New York, New York, USA, p.268, 2002.

ARGE, L. et al. An Optimal Cache Oblivious Priority Queue and Its Application to Graph Algorithms. **SIAM Journal on Computing**, [S.l.], v.36, n.6, p.1672–1695, jan 2007.

ARGE, L.; SITCHINAVA, N.; GOODRICH, M. T. Parallel external memory graph algorithms. **Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010**, [S.l.], 2010.

BELAZZOUGUI, D. et al. Bidirectional Variable-Order de Bruijn Graphs. In: LATIN 2016: THEORETICAL INFORMATICS: 12TH LATIN AMERICAN SYMPOSIUM, ENSENADA, MEXICO, APRIL 11-15, 2016, PROCEEDINGS, Berlin, Heidelberg. **Anais...** Springer Berlin Heidelberg, 2016. p.164–178.

BELLER, T.; OHLEBUSCH, E. Efficient Construction of a Compressed de Bruijn Graph for Pan-Genome Analysis. In: ANNUAL SYMPOSIUM ON COMBINATORIAL PATTERN MATCHING - CPM 2015, 26., Ischia Island. **Proceedings...** Springer, 2015. p.40–51.

BENDER, M. a.; FARACH-COLTON, M.; KUSZMAUL, B. C. Cache-oblivious string B-trees. **Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '06**, New York, New York, USA, p.233, 2006.

BENDER, M.; FARACH-COLTON, M. Cache-oblivious streaming B-trees. **Proceedings of the ...**, [S.l.], p.81–92, 2007.

BENTLEY, J. **Programming Pearls, 2/E**. 2nd.ed. [S.l.]: Pearson Education India, 2000.

BOWE, A. et al. Succinct de Bruijn Graphs. In: INTERNATIONAL WORKSHOP ON ALGORITHMS IN BIOINFORMATICS. **Anais...** Springer, 2012. p.225–235.

BRODAL, G. et al. Cache-Oblivious Data Structures and Algorithms for Undirected Breadth-First Search and Shortest Paths. **Algorithm Theory-SWAT 2004**, [S.l.], v.14186, n.February, p.480–492, 2004.

BRUIJN, N. G. de. **A combinatorial problem**. [S.l.]: Koninklijke Nederlandse Akademie van Wetenschappen, 1946.

BUCHSBAUM, A. L. et al. On External Memory Graph Traversal. In: ACM-SIAM SYMPOSIUM ON DISCRETE ALGORITHMS. **Proceedings...** [S.l.: s.n.], 2000. p.859–860.

CHAISSON, M. J.; BRINZA, D.; PEVZNER, P. A. De novo fragment assembly with short mate-paired reads: does the read length matter? **Genome Research**, [S.l.], v.19, n.2, p.336–346, 2009.

CHIANG, Y.-J. et al. External-Memory Graph Algorithms. In: ACM-SIAM SYMPOSIUM ON DISCRETE ALGORITHMS. **Anais...** [S.l.: s.n.], 1995. p.139–149.

CHIKHI, R. et al. On the representation of de bruijn graphs. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, [S.l.], v.8394 LNBI, p.35–55, 2014.

CHIKHI, R.; RIZK, G. Space-efficient and exact de Bruijn graph representation based on a bloom filter. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, [S.l.], v.7534 LNBI, p.236–248, 2012.

CLAROS, M. G. et al. Why assembling plant genome sequences is so challenging. **Biology**, [S.l.], v.1, n.2, p.439–59, 2012.

CONWAY, T. C.; BROMAGE, A. J. Succinct data structures for assembling large genomes. **Bioinformatics**, [S.l.], v.27, n.4, p.479–486, 2011.

CONWAY, T. et al. Gossamer - A resource-efficient de novo assembler. **Bioinformatics**, [S.l.], v.28, n.14, p.1937–1938, 2012.

CORMEN, T. H. et al. **Introduction to Algorithms, Third Edition**. 3rd.ed. [S.l.]: The MIT Press, 2009.

DEMAINE, E. Cache-oblivious algorithms and data structures. **Lecture Notes from the EEF Summer School on Massive Data Sets**, [S.l.], 2002.

ENGLANDER, I. **The Architecture of Computer Hardware, System Software, and Networking: an information technology approach**. 4th.ed. [S.l.]: John Wiley & Sons Software, 2009. 708p.

FRIGO, M. et al. Cache-Oblivious Algorithms. **40th Annual Symposium on Foundations of Computer Science**, [S.l.], v.8, n.1, p.285–287, jan 1999.

GROSSI, R.; GUPTA, A.; VITTER, J. S. High-Order Entropy-Compressed Text Indexes. **Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms**, [S.l.], v.2068, p.841–850, 2003.

HIERHOLZER, C.; WIENER, C. Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. **Mathematische Annalen**, [S.l.], v.6, n.1, p.30–32, mar 1873.

IDURY, R. M.; WATERMAN, M. S. A New Algorithm for DNA Sequence Assembly. **Journal of Computational Biology**, [S.l.], v.2, n.2, p.291–306, 1995.

JACOBSON, G. **Space-efficient static trees and graphs**. 1989. 549–554p.

KARP, R. M. Reducibility among Combinatorial Problems. In: **Complexity of Computer Computations**. Boston, MA: Springer US, 1972. p.85–103.

LADNER, R.; FORTNA, R.; NGUYEN, B. H. A Comparison of Cache Aware and Cache Oblivious Static Search Trees Using Program Instrumentation. **Experimental Algorithmics**, [S.l.], p.78–92, 2002.

LI, R. et al. De novo assembly of human genomes with massively parallel short read sequencing. **Genome research**, [S.l.], p.265–272, 2010.

LI, Z. et al. Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-bruijn-graph. **Briefings in Functional Genomics**, [S.l.], v.11, n.1, p.25–37, 2012.

LIU, L. et al. Comparison of next-generation sequencing systems. **Journal of Biomedicine and Biotechnology**, [S.l.], v.2012, 2012.

MARDIS, E. R. Next-generation DNA sequencing methods. **Annual review of genomics and human genetics**, [S.l.], v.9, p.387–402, jan 2008.

MILICCHIO, F. et al. High-performance Data Structures for De Novo Assembly of Genomes: cache oblivious generic programming. In: ACM INTERNATIONAL CONFERENCE ON BIOINFORMATICS, COMPUTATIONAL BIOLOGY, AND HEALTH INFORMATICS, 7., New York, NY, USA. **Proceedings...** ACM, 2016. p.657–662. (BCB '16).

MILLER, J. R.; KOREN, S.; SUTTON, G. Assembly algorithms for next-generation sequencing data. **Genomics**, [S.l.], v.95, n.6, p.315–327, 2010.

NAVARRO, G.; PROVIDEL, E. Fast, Small, Simple Rank/Select on Bitmaps. In: KLASING, R. (Ed.). **Experimental Algorithms**: 11th international symposium, sea 2012, bordeaux, france, june 7-9, 2012. proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p.295–306.

NELSON, D. L.; COX, M. M. **Lehninger Principles of Biochemistry**. Fifth Edit.ed. [S.l.]: W. H. Freeman, 2008.

PEVZNER, P. A.; TANG, H.; WATERMAN, M. S. An Eulerian path approach to DNA fragment assembly. **Proceedings of the National Academy of Sciences of the United States of America**, [S.l.], v.98, n.17, p.9748–9753, 2001.

PRIMROSE, S. B.; TWYMAN, R. M. **Principles of Genome Analysis and Genomics**. 3rd.ed. Malden, MA (USA): Blackwell, 2003. v.1.

REIF, J. H. **Synthesis of Parallel Algorithms**. 1st.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. 1011p.

SACH, B.; CLIFFORD, R. An Empirical Study of Cache-Oblivious Priority Queues and their Application to the Shortest Path Problem. **Computing Research Repository**, [S.l.], v.abs/0802.1, 2008.

SALIKHOV, K.; SACOMOTO, G.; KUCHEROV, G. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. **Algorithms for Molecular Biology**, [S.l.], v.9, n.1, p.2, 2014.

SANGER, F.; NICKLEN, S.; COULSON, a. R. DNA sequencing with chain-terminating inhibitors. **Proceedings of the National Academy of Sciences of the United States of America**, [S.l.], v.74, n.12, p.5463–7, 1977.

SEWARD, J. et al. **Valgrind Documentation**. 2016.

SHENDURE, J.; JI, H. Next-generation DNA sequencing. **Nature Biotechnology**, [S.l.], v.26, n.10, p.1135–1145, 2008.

SILVA, I. B. F. da. **Representações cache eficientes para índices baseados em wavelet trees**. 2016. Dissertação (Mestrado em Ciência da Computação) — Centro de Informática, Universidade Federal de Pernambuco.

SIMPSON, J. T. et al. ABySS: a parallel assembler for short read sequence data. **Genome research**, [S.l.], v.19, n.6, p.1117–1123, 2009.

STADEN, R. A strategy of DNA sequencing employing computer programs. **Nucleic Acids Research**, [S.l.], v.6, n.7, p.2601–2610, 1979.

TANENBAUM, A. S.; BOS, H. **Modern operating systems**. 4th.ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014. 1136p.

TARJAN, R. E.; VISHKIN, U. Finding biconnected componemts and computing tree functions in logarithmic parallel time. **25th annual Symposium on Foundations of Computer Science (FOCS)**, [S.l.], p.12–20, 1984.

van Emde Boas, P. Preserving order in a forest in less than logarithmic time. In: ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, 16., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1975. p.75–84.

VISHKIN, U. On efficient parallel strong orientation. **Information Processing Letters**, [S.l.], v.20, p.235–240, 1985.

VITTER, J. S. External Memory Algorithms and Data Structures: dealing with massive data. **DIMACS Series in Discrete Mathematics and Theoretical Computer Science**, [S.l.], v.33, n.June, p.1–39, 2007.

WYLLIE, J. C. **The complexity of parallel computation**. 1979. PhD — Cornell University.

ZERBINO, D. R.; BIRNEY, E. Velvet: algorithms for de novo short read assembly using de bruijn graphs. **Genome research**, [S.l.], v.18, n.5, p.821–829, 2008.