4
Velvet

O programa Velvet foi escolhido como objeto de estudo deste trabalho de mestrado pelo amplo uso nas pesquisas da UFRJ com quem a PUC-Rio possuiu uma parceria.

E neste capítulo serão apresentadas a estrutura do Velvet e o seu funcionamento, fazendo a separação dos módulos Velvet H e Velvet G. Além de apresentar uma visão detalhada da implementação de um dos módulos, o Velvet H.

Também será apresentada uma análise realizada sobre o desempenho do módulo Velvet H em grandes volumes de dados e trazer duas melhorias no código afim de entender melhor o funcionamento e a sua relação com o consumo de memória principal.

4.1.
Estrutura

Velvet é descrito pelos seus criadores, Daniel R. Zerbino e Ewan Birney, como um conjunto de algoritmos criados para a manipulação de grafos de Bruijn para a montagem de sequências muito curtas (NGS) [4].

Estes algoritmos podem ser estudados em 7 fases distintas, e por sua vez agrupadas em dois subprogramas programas, VelvetH e VelvetG, destacadas abaixo e descritas em seguida.

VelvetH

1. Padronização das sequências
2. Criação das anotações de sobreposições das sequências

VelvetG

3. Criação do pré-grafo
4. Criação do grafo
5. Inclusão de informações dos nós
6. Heurísticas para remoção de erros
7. Calcula o caminho Euleriano para construir os contigs

O subprograma VelvetH é responsável pelo pré-processamento do grafo de Bruijn. Em sua primeira fase, os dados de entrada, que são as bibliotecas com os fragmentos gerados no sequenciamento, são processados e colocados em um único arquivo, com formato canônico do próprio Velvet usado em todas as outras fases, além de incluir um identificador inteiro único para cada fragmento processado.

Na segunda fase, o programa processa este arquivo canônico construído na fase anterior, criando uma série de anotações acerca das repetições dos k-mers nos fragmentos, gerando um segundo arquivo chamado Roadmap, que será usado na terceira fase.

A terceira fase, já executada no módulo VelvetG, tem como entrada os 2 arquivos criados pelas fases 1 e 2 no VelvetH. Esta fase se utiliza das anotações das áreas de repetição entre os fragmentos para criar um pré-grafo inicial, desprezando os fragmentos que não possuam repetições, pois como a cobertura para os fragmentos curtos são altas, fragmentos sem repetições é um indicativo de fragmento com erro de leitura no sequenciamento. Neste ponto também é realizada operações para unir os nós que possuem apenas um vizinho de cada lado e remover as pontas do grafo, que são áreas com baixa repetição, que é um outro indicativo de erro de sequenciamento. Esta fase tende a reduzir o tamanho do grafo, se comparado a uma geração direta dos fragmentos, numa escala de 10 a 100 vezes[12].

Após a redução inicial do grafo, a quarta fase cria o grafo completo, utilizando uma estrutura de dados maior para o nó, para que na quinta fase possam ser incluídas ao grafo informações adicionais geradas a partir dos dados de entrada.

Estes dados adicionados ao grafo são utilizados pela sexta fase, quando são executadas heurísticas de transformações no grafo com o intuito de determinar ambiguidades geradas por erros e removê-las do grafo.

Em sua 7ª e última fase, o programa calcula em cada grafo o caminho Euleriano, gerando um conjunto dos *contigs*[19].

4.2.
VelvetH e o Consumo de memória RAM

O subprograma VelvetH tem como principal objetivo reduzir o tamanho do grafo devido ao elevado número de erros de sequenciamento. Com o uso do Roadmap, é possível descartar parte dos erros de sequenciamento antes da criação do grafo. Com este descarte, é possível reduzir em até 8 vezes[12] o tamanho do grafo de Brujin, resultando diretamente em uma redução significativa no consumo de memória principal.

A Figura 8, ilustra esta diferença no consumo de memória com o uso do Roadmap, em diferentes tamanhos de genoma. Neste exemplo, foram usadas as amostras de S. suis, E. coli, S. cerevisiae (levedura) e C. elegans (Nemátodos), a cobertura foi de 50X e K igual a 21. A primeira série, marcada por losangos, indica o consumo de memória principal para a geração do grafo sem o uso do Roadmap, indicando um comportamento linear ao tamanho do genoma. A segunda série, marcada por quadrados, indica o consumo utilizando o Roadmap para a geração do grafo, e indica o mesmo comportamento linear no consumo de memória em relação ao tamanho genoma. Porém com um gasto significativamente menor de memória principal.

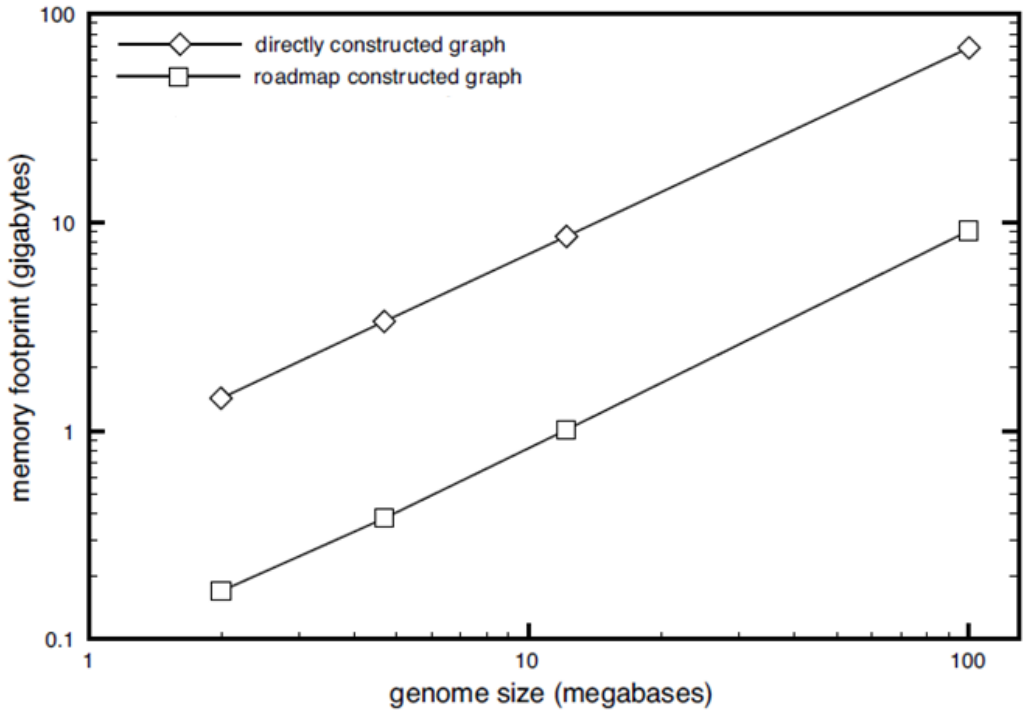

Figura 8 Diferença de consumo de memória principal com uso do Roadmp [12].

4.3.
Testes de execução

Com o objetivo de entender o consumo de memória principal da aplicação em maquinas de prateleira, decidimos realizar alguns testes experimentais com um certo conjunto de dados e medir o consumo de memória principal do módulo VelvetH.

Ambiente Computacional e Dados

Foi criada uma máquina virtual, através do Virtual Box (V4.3.34), com uma instalação de sistema operacional Ubuntu 64 (V14.04 LTS) e 4GB de memória RAM disponíveis, além de 2 processadores (Intel® Core™ i7-4510U CPU @ 2.00GHz × 2) e 20GB de espaço em disco.

Usamos 3 bibliotecas de dados de sequências, uma com 50.000 (10MB) sequências, outra com 2.191.196 (500MB) sequencias e outra com 12.671.416 (3.9GB) sequências.

A primeira biblioteca é um conjunto de sequências de teste do próprio Velvet, as duas últimas são 2 arquivos com um subconjunto de sequências que possuem ao todo mais de 275 milhões de sequências. São os dados do projeto de mapeamento do genoma da cana-de-açúcar que a UFRJ deseja montar em seu laboratório.

Resultados

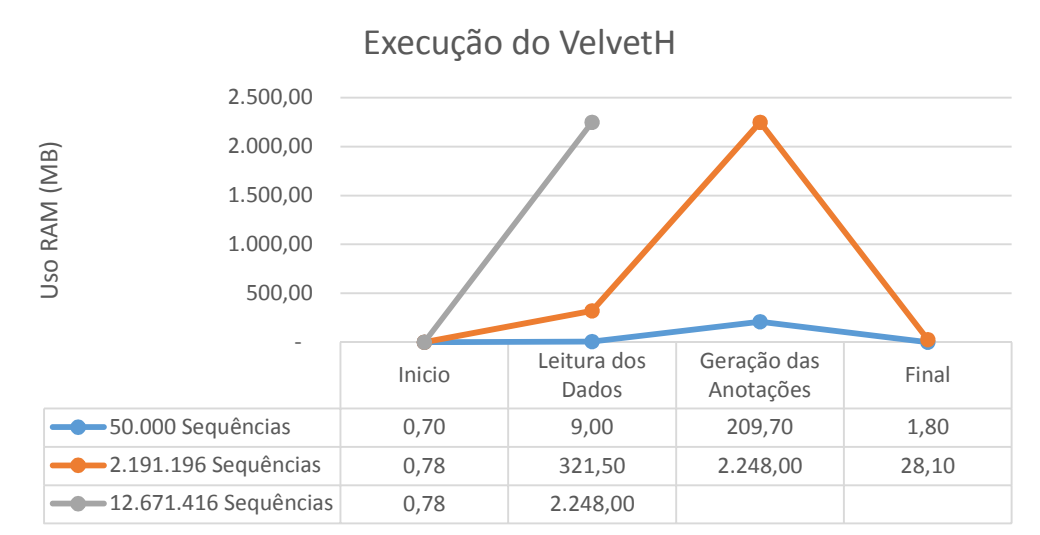

Figura 9 Consumo de memória Principal pelo VelvetH.

Como pode ser observado na Figura 9, o conjunto de dados com mais de 12 milhões de registro não conseguiu ser executado pela aplicação. Esta exceção foi interrompida após 12h de processamento e todo o trabalho processado foi perdido.

Mesmo sendo um módulo que visa a redução do consumo de memória principal, quando o conjunto de dados é grande em relação a quantidade de memória disponível, este também enfrenta problemas de consumo de memória principal.

A partir da secção 4.4, será descrito em detalhes o funcionamento do subprograma VelvetH e apresentar algumas alterações no código com o intuito de direcionar o estudo e entender melhor consumo de memória da implementação abordada nesta pesquisa.

4.4. Funcionamento do VelvetH

O módulo VelvetH é responsável pela execução das duas primeiras fases do Velvet, padronização dos dados de entrada e a geração das anotações, o *Roadmap*.

Os principais parâmetros para execução deste subprograma são:

- [output_directory]: Diretório onde os 2 arquivos, sequences e roadmap, serão escritos
- [hash_length]: Tamanho do k-mer a ser processado, sempre ímpar para evitar palíndromos.
- [-file_format]: Formato do arquivo do dado de entrada (*.fastq, *.fasta, *.sam, *.baw, *.raw, *.fmtauto)
- [-read_type]: Tipo de sequência a ser processada (short, shortPaired, short2, shortPaired2, long, longPaired)
- [filename]: Caminho do arquivo de sequência
- -strand_specific: Para a montagem considerar apenas um sentido da sequência.

4.4.1. Fase de padronização dos dados

Nesta fase o programa faz a leitura dos arquivos contendo as sequências geradas pelo processo de sequenciamento, podendo estar em diversos formatos (*.fastq, *.fasta, *.sam, *.baw, *.raw, *.fmtauto), na Figura 10 vemos o exemplo de um arquivo na extensão *.fastq. Além de gerar um arquivo chamado *sequences,* um

dos produtos deste módulo, contendo todas as sequencias padronizadas em um formato canônico do próprio Velvet, conforme ilustrado na Figura 11.

O algoritmo processa cada arquivo de acordo com seu formato, extraindo a sequência e do cabeçalho de cada sequência o nome e a categoria, como pode ser visto na Figura 10.

Após isso escreve-se no arquivo canônico um cabeçalho para cada sequência com as seguintes colunas separadas por tabulação:

- Nome da sequência
- Chave da sequência (Sequencial inteiro criado pelo VelvetH)
- Categoria da sequência

Abaixo do cabeçalho da sequência, escreve o conteúdo da sequência em linhas contendo no máximo 60 caracteres cada linha.

reads.fq

```
@HWUSI-EAS-100R_0001:7:1:2:1023#TGACCA/1
TGTCCACAACCCCGGAAACAAGTTTCCGGTTTGGGCTCTTTCCCTTTCGCTCGCCGCTACTACGAAACTCGATTTTTCT
+
a_\ba^a`[aba`RY\a`_]Z^Q^\_`Y]V\_S\\aU^VZW_U\WWVYKTUBBBBBBBBBBBBBBBBBBBBBBBBBBB
@HWUSI-EAS-100R_0001:7:1:12:414#TGACCA/1
TTTGGACTGTTTCCCTTTCGCTCGCCGCTACTAAGAAAATCGATTTTTCTTTCTCTTCCTCCAGGTACTTAGATGTTTC
+
aabbaababaaabbbaaabab_aaaa`a`aa_[S_[R[O\_][[]Z`Y]\^\`\a^\`_W__YUQOU^XMFRQZZXYQX
@HWUSI-EAS-100R_0001:7:1:12:1753#TGACCA/1
ACAAGTTTCCGGTTTGGGCTCTTTCCCTTTCGCTCGCCGCTACTAAGAAAATCGATTTTTCTTTCTCTTCCTCCAGGTA
+
aba_a^aababa__```aa`a^\^_]_]Z]^V_^`X^YU[WQ[TGMWONONWYNHPSRXSOJVRYY^VJOW\BBBBBBB
@HWUSI-EAS-100R_0001:7:1:12:1798#TGACCA/1
CGCTCTTTCCCTTTCGCTCGCCGCTACTAAGAGGGAAATGCTCTTCGCTCTTTCCCTTTCGCTCGCCGCTACTATCCCT
+
aba_a^aababa__```aa`a^\^_]_]Z]^V_^`X^YU[WQ[TGMWONONWYNHPSRXSOJVRYY^VJOW\BBBBBBB
```

Figura 10 Exemplo de um arquivo de sequências no formato fastq

sequences

```
>HWUSI-EAS-100R_0001:7:1:2:1023#TGACCA/1          1          0
TGTCCACAACCCCGGAAACAAGTTTCCGGTTTGGGCTCTTTCCCTTTCGCTCGCCGCTAC
TACGAAACTCGATTTTTCT
>HWUSI-EAS-100R_0001:7:1:12:414#TGACCA/1          2          0
TTTGGACTGTTTCCCTTTCGCTCGCCGCTACTAAGAAAATCGATTTTTCTTTCTCTTCCT
CCAGGTACTTAGATGTTTC
>HWUSI-EAS-100R_0001:7:1:12:1753#TGACCA/1         3          0
ACAAGTTTCCGGTTTGGGCTCTTTCCCTTTCGCTCGCCGCTACTAAGAAAATCGATTTTT
CTTTCTCTTCCTCCAGGTA
>HWUSI-EAS-100R_0001:7:1:12:1798#TGACCA/1         4          0
CGCTCTTTCCCTTTCGCTCGCCGCTACTAAGAGGGAAATGCTCTTCGCTCTTTCCCTTTC
GCTCGCCGCTACTATCCCT
```

Figura 11 Exemplo de um arquivo *sequences* criado pelo VelvetH

4.4.2.
Fase de geração do *Roadmap*

Nesta fase o programa tem como objetivo criar um arquivo com um conjunto de anotações acerca das repetições dos k-mers nas sequências, que será usado na pelo VelvetG, afim de reduzir o tamanho do grafo.

Conforme apresentado na Figura 12, podemos dividir esta fase em duas etapas:

1. Faz a leitura de todos os dados a serem processados
2. Realiza o cruzamento entre as sequências afim de descobrir as sobreposições existentes e gerar as anotações.

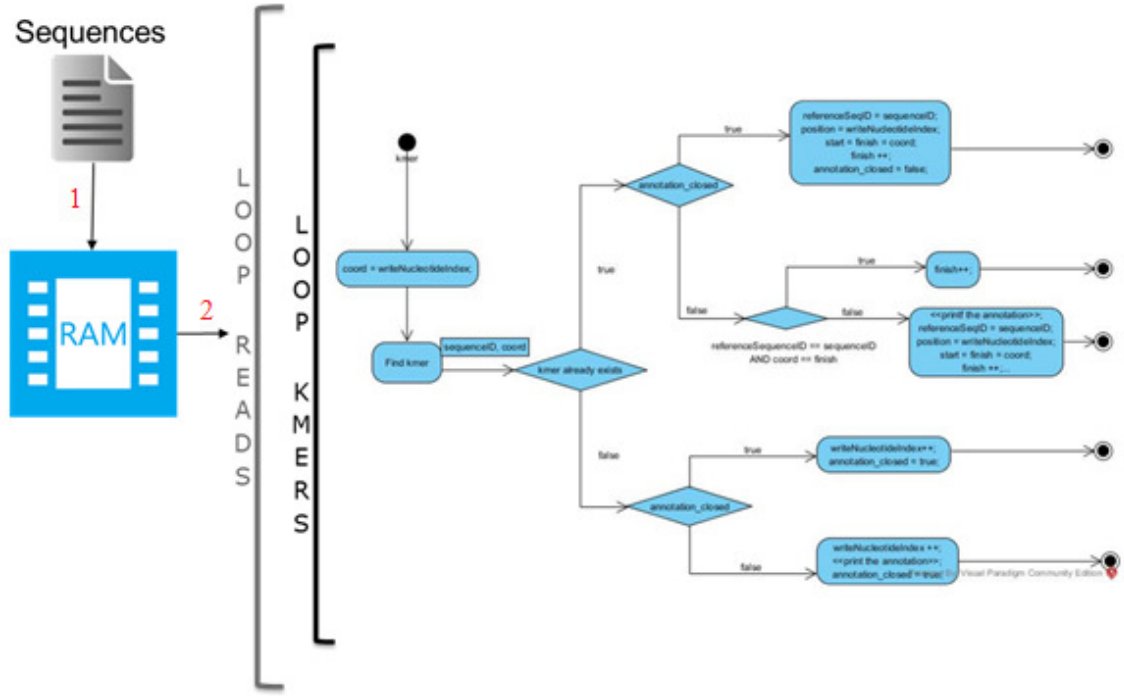

Figura 12 Esquemático da geração do Roadmap

A análise de forma separada justifica-se devido ao encontro de pontos de melhoria em cada uma destas etapas.

4.4.2.1.
Leitura dos Dados

Nesta etapa o programa tem por objetivo preencher a estrutura de dados, descrita na Figura13, que será usada na fase seguinte, a de geração das anotações.

```
struct readSet_st {
        char **sequences;
        TightString *tSequences;
        char **labels;
        char *tSeqMem;
        Quality **confidenceScores;
        Probability **kmerProbabilities;
        IDnum *mateReads;
        Category *categories;
        unsigned char *secondInPair;
        IDnum readCount;
};
```

Figura 13 Estrutura de dados utilizada na fase de geração do *Roadmap*

O primeiro passo do algoritmo é ler 3 vezes o arquivo *sequences*, criado pela fase anterior. A primeira leitura para fazer a contagem total de sequências existentes no arquivo. A segunda leitura faz a alocação de memória para cada sequência, de acordo com o tamanho de caracteres necessários para armazenar o cabeçalho, e o tamanho necessário para armazenar o conteúdo da sequência, não divido em linhas como no arquivo, mas sim a cadeia de caracteres completa. Na 3ª e última leitura do arquivo o programa preenche a estrutura já alocada na leitura anterior com o conteúdo das sequências e processa, de forma separada, as sequencias de referência.

Após o preenchimento da estrutura, o sistema cria o arquivo de output *Roadmap* e já preenche com o cabeçalho, que possuiu as seguintes colunas separadas por tabulação:

- Quantidade de sequências
- Quantidade de sequências de referência
- Tamanho do K-mer utilizado
- Indicador booleano da verificação das sequências em sentido único ou sentido duplo.

Finalizando esta etapa, o programa converte o *array* de *char* para um *array* de *TightString*, um tipo criado pelo programa para armazenar as sequências onde cada caractere ocupe 2 bits e não 8, aliviando assim o uso da memória.

4.4.2.2. Geração das anotações

Este passo engloba a lógica de identificação dos trechos de sobreposição existentes entre as sequências. Para isso, o programa precisa identificar os k-mers únicos de cada sequência, mapeando o seu identificador único e a posição relativa de onde se encontra nesta sequência.

A posição é relativa na sequência pois esta posição não é a posição literal da substring, k-mer, na string da sequência. Esta posição é referente unicamente aos k-mers exclusivos, que são os k-mers não repetidos nas sequências anteriormente processadas. Caso uma sequência possua apenas um k-mer não repetido e este seja o último substring possível da sequência, este ganhará a posição 1 do k-mer na sequência, já que os outros k-mers existentes são apenas repetições de outras sequências já processadas.

Para realizar este trabalho o algoritmo separa o universo de k-mer possíveis em índices, usando uma função hash e os armazena na memória principal, através de uma SplayTree, que é uma árvore balanceada de busca, alocando os k-mers nos nós [20].

O algoritmo começa a montagem das anotações a partir da estrutura já reduzida das sequências, faz um laço para cada sequência e escreve no arquivo de saída Roadmap o registro que está sendo processado em 2 colunas separadas por tabulação:

- Texto fixo “ROADMAP”
- Número identificador da sequência criado na etapa de padronização dos dados

Após a entrada no arquivo, o programa faz um novo laço, agora pela quantidade de k-mer existente na sequência, que é dada pela equação

Qtd. K-mer = (tamanho da sequência - tamanho k-mer +1).

Para cada k-mer, o programa faz uma busca na SplayTree. Caso não o encontre, aloca-se um espaço de memória para o nó do k-mer e adiciona-o ele na árvore junto com o identificador da sequência e a posição relativa em que se encontra na sequência. Caso o k-mer tenha sido encontrado, a função de busca retorna o identificador da sequência e a posição relativa em que este k-mer se encontra mapeado, e não tenha anotação em memória em aberto, inicia-se uma nova

anotação em memória guardando o identificador da sequência sobreposta, a última posição de k-mer localizado na sequência que está sendo processada, a posição relativa do k-mer encontrado em sua sequência original e a largura da sobreposição, que na primeira vez que é encontrada a repetição é a posição relativa do k-mer em sua sequência original mais 1.

No caso de uma anotação já estar iniciada, o programa verifica se a repetição é em sequência e se é da mesma sequência de origem, ou seja, caso tenham 2 k-mers repetidos em sequência em outra sequência, o algoritmo atualiza o tamanho da largura da sobreposição somando mais 1.

Caso a repetição não seja em sequência, ou não tenha outra repetição, fecha-se esta anotação escrevendo no arquivo de saída Roadmap as seguintes colunas separadas por tabulação:

- Identificador da sequência sobreposta
- Última posição de k-mer encontrado na sequência que está sendo processada.
- Primeira posição relativa da repetição encontrada na sequência sobreposta
- Largura da sobreposição, que é a primeira posição relativa da repetição encontrada mais a quantidade de k-mer sobrepostos em sequência.

Após processar todos os *k-mers* de todas as sequências, o arquivo de saída *Roadmap*, como mostrado na Figura 14, estará pronto e o sistema irá liberar todo o espaço em memória utilizado e encerrar a sua execução.

Roadmap

4	0	29	0
ROADMAP 1			
ROADMAP 2			
ROADMAP 3			
1	0	17	34
2	4	9	39
ROADMAP 4			
3	1	0	3
4	42	0	1

Figura 14 Exemplo de um arquivo *Roadmap*

4.5. Modificações realizadas no VelvetH

Com o intuito de entender melhor o código e trazer melhoria em seu desempenho, foram realizadas duas alterações pontuais, ambas focadas no processamento realizado antes do início da montagem do *Roadmap*, afim de entender a participação desta parte do código no consumo de memória.

A primeira é na realidade um ajuste no código que faz a leitura nos dados, reduzindo o gasto com memória neste passo e a segunda uma modificação nos passos de geração do *Roadmap*, integrando a leitura dos dados com a geração das anotações, dado que o processamento é serial e a leitura da sequência pode ser feito em conjunto com a montagem do *Roadmap*.

4.5.1. Ajuste 1: Alocação de memória na leitura dos dados

No passo em que o algoritmo lê o arquivo *sequences* pela segunda vez, visando determinar o tamanho de cada sequência e alocar a memória necessária, identificamos um pequeno problema na implementação que faz a alocação de memória para o conteúdo da sequência ser maior que o necessário.

Conforme marcação na Figura 15, por conta deste *IF* e não o *ELSE IF* como deveria ser, a quantidade de caracteres existentes no *header* é somada à quantidade de caracteres necessários para alocar o conteúdo da sequência.

```
file = fopen(filename, "r");
sequenceIndex = -1;
while (fgets(line, maxline, file) != NULL) {
        if (line[0] == '>') {

                // Reading category info
                sscanf(line, "%*[^\t]\t%*[^\t]\t%hd",
                        &temp_short);
                reads->categories[sequenceIndex + 1] = (Category) temp_sh

                if (sequenceIndex != -1)
                        reads->sequences[sequenceIndex] =
                            mallocOrExit(bpCount + 1, char);
                sequenceIndex++;
                bpCount = 0;
        } if (line[0] == 'M') {;
                // Map line
        } else {
                bpCount += (Coordinate) strlen(line) - 1;

                if (sizeof(ShortLength) == sizeof(int16_t) && (bpCount >
                        velvetLog("Read %li of length %lli, longer than T
                                (long) sequenceIndex + 1, (long long) bpCo
                        velvetLog("You should modify recompile with the L
                        exit(1);
                }
        }
}
```

Figura 15 Erro na contagem de caracteres necessários para alocar o conteúdo da sequência

Com esta falha, o custo de alocação neste item seria em *bytes* igual a:

*(Tamanho Sequência + Tamanho Cabeçalho +1) * Quantidade de sequências no arquivo*

Em um cenário com 12 milhões de sequências de tamanho de 100bp e um cabeçalho de 50 caracteres o custo de memória extra seria de aproximadamente 104MB. Já em um cenário com 275 milhões de sequências, correspondente à aos dados da cana-de-açúcar utilizado nas pesquisas da UFRJ, o consumo adicional seria de aproximadamente 12,8 GB de memória RAM.

4.5.2.
Ajuste 2: Integração dos passos Leitura de Dados e Geração do *Roadmap*

A geração das anotações é feita de forma serial e não é necessário ter todas as sequências em memória para iniciar a geração das anotações.

Esta nova implementação, proposta nesta dissertação, tem como objetivo reduzir o consumo de memória da primeira parte da segunda fase do VelvetH, não alocando na memória todas as sequências existentes no arquivo *sequences* gerado pela primeira fase. Como demonstrado na Figura 16, o arquivo com as sequências é aberto uma única vez e cada fragmento é lido uma vez, gerando as anotações que lhe são referentes. Em seguida, descartado.

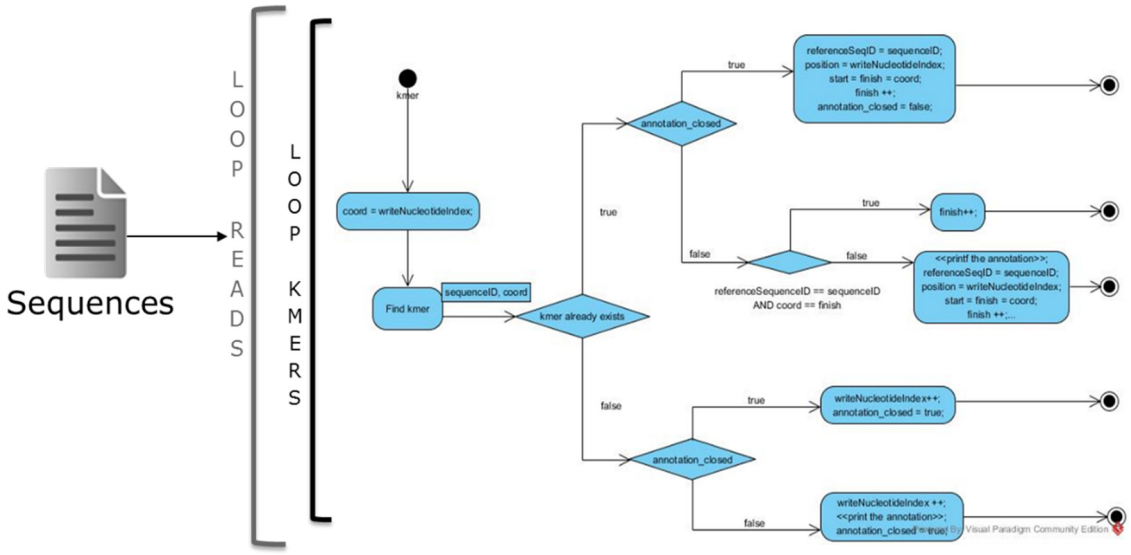

Figura 16 Alteração realizada no código fonte do VelvetH

4.5.3.
Resultado experimentais

Após as modificações no código, refizemos os testes no mesmo ambiente computacional descrito anteriormente, com os mesmos conjuntos de dados, e obtivemos os seguintes resultados.

No conjunto de dados com 50.000 sequências, o consumo de memória RAM reduziu em 25% na fase da Leitura dos Dados com o Ajuste 1 e 80% de redução do

uso de RAM com o Ajuste 2. Entretanto, na fase final de geração do RoadMap o consumo de memória permaneceu igual ao código original, conforme mostrado pela Figura 17.

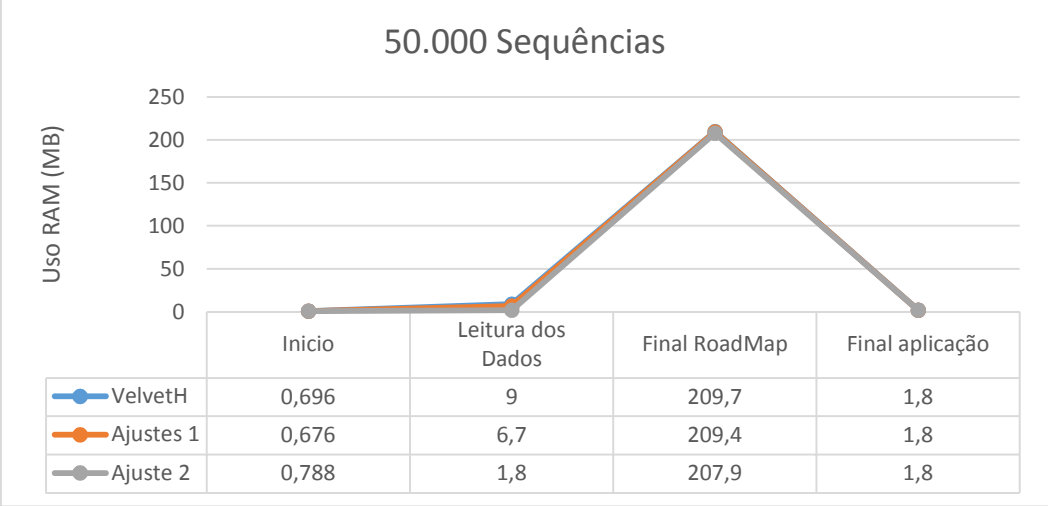

Figura 17 Resultado das alterações no conjunto de dados com 50.000 sequências

No conjunto de dados com 2.191.196 sequências, o consumo de memória RAM reduziu em 30% na fase da Leitura dos Dados com o Ajustes 1 e 99% de redução do uso de RAM com o Ajuste 2, porém execução final continuou com o consumo igual, conforme mostrado pela Figura 18.

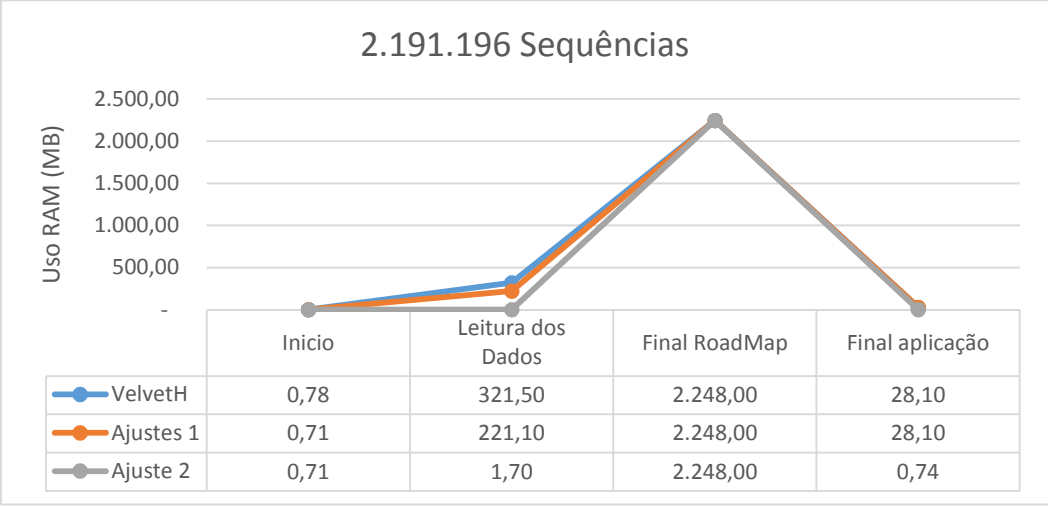

Figura 18 Resultado das alterações no conjunto de dados com 2.191.196 sequências

No conjunto de dados com 12.671.416 sequências, o consumo de memória RAM não reduziu na fase da Leitura dos Dados com o Ajustes 1 e o consumo de memória RAM foi praticamente 0 com o Ajuste 2, porém nas 3 execuções o algoritmo não executou até o final e foi interrompido depois de 12h de espera, conforme mostrado pela Figura 19.

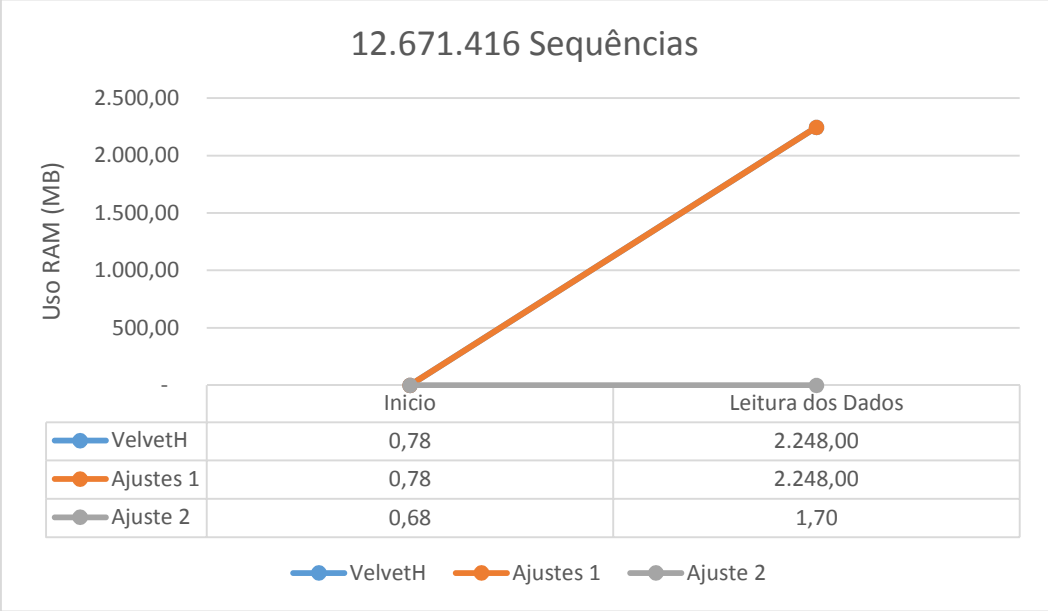

Figura 19 Resultado das alterações no conjunto de dados com 12.671.416 sequências

Analisando os dados apresentados, é possível observar a redução significativamente o consumo de memória no RAM na fase de leitura dos dados. Entretanto, não acarretou em melhorias na fase de montagem do *Roadmap* pois o consumo de memória RAM neste passo ficou igual em todos os casos testados.

4.6. Conclusão

A estrutura do funcionamento do *VelvetH*, foi apresentada com uma análise de desempenho da aplicação com grandes volumes de dados e os resultados de duas modificações realizadas no código afim de reduzir o consumo de memória, com esperança de que reduziria para a execução do *Velvet* com um todo.

No próximo capítulo será apresentada solução para o mapeamento dos dados de sobreposição das sequências em banco de dados, que tem como objetivo

viabilizar a execução do VelvetH em maquinas de prateleira, garantindo a eficácia da geração das anotações.