FABIO YANAGA

UMA EXTENSÃO DA ESTRUTURA DE INDEXAÇÃO MRS PARA USO EM MEMÓRIA SECUNDÁRIA

MARINGÁ

2006

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

FABIO YANAGA

UMA EXTENSÃO DA ESTRUTURA DE INDEXAÇÃO MRS PARA USO EM MEMÓRIA SECUNDÁRIA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Ricardo Rodrigues Ciferri
Co-orientador: Prof. Dr. Ademir Aparecido Constantino

MARINGÁ

2006

Dados Internacionais de Catalogação-na-Publicação (CIP)
(Biblioteca Central - UEM, Maringá – PR., Brasil)

Yanaga, Fabio
Y21u Uma extensão da estrutura de indexação MRS para uso em memória secundária / Fabio Yanaga. -- Maringá : [s.n.], 2006.
100 f. : il. color., figs., tabs.

Orientador : Prof. Dr. Ricardo Rodrigues Ciferri.
Co-Orientador : Prof. Dr. Ademir Aparecido Constantino.
Dissertação (mestrado) - Universidade Estadual de Maringá. Programa de Pós-Graduação em Ciência da Computação, 2006.

1. Estrutura de indexação MRS. 2. Pesquisa de similaridade. 3. Memória secundária. 4. Testes de desempenho. 5. Bancos de dados biológicos. 6. BLAST. I. Universidade Estadual de Maringá. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22.ed. 005.741

FABIO YANAGA

UMA EXTENSÃO DA ESTRUTURA DE INDEXAÇÃO MRS PARA USO EM MEMÓRIA SECUNDÁRIA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Aprovado em 06/11/2006.

BANCA EXAMINADORA

Prof. Dr. Ricardo Rodrigues Ciferri
Universidade Federal de São Carlos – DC/UFSCar

Prof. Dr. Ademir Aparecido Constantino
Universidade Estadual de Maringá – DIN/UEM

Profª. Drª. Itana Maria de Souza Gimenes
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Mauro Biajiz
Universidade Federal de São Carlos – DC/UFSCar

AGRADECIMENTOS

Primeiramente, agradeço a Deus por tudo. Por não precisar de mais nada além do que já ganho todos os dias, e por saber que Ele estará comigo até o fim. Obrigado pela força e proteção que me dedicou para que eu pudesse cumprir mais um objetivo em minha vida. Agradeço aos meus pais que fizeram sacrifícios para que eu pudesse chegar onde cheguei e ser quem sou.

Ao meu irmão Edson e a minha cunhada Edna, agradeço por todo o apoio e incentivos dedicados a mim. Ao meu querido e único sobrinho, Felipe, agradeço por todos os momentos de alegria proporcionados, mesmo que estes resultassem em momentâneas interrupções no desenvolvimento deste trabalho.

Agradeço a minha namorada Camila por todo o afeto, apoio, paciência e compreensão durante a trajetória deste trabalho.

À professora Cristina Dutra de Aguiar Ciferri e ao professor Ademir Constantino, meu co-orientador, agradeço de todo o coração toda a ajuda e orientação dadas para a conclusão deste trabalho.

Reservo um especial agradecimento ao meu orientador, professor Ricardo Rodrigues Ciferri, que sempre me apoiou e me incentivou na realização deste trabalho. Obrigado pelo profissionalismo, pela dedicação, e principalmente pela compreensão. Obrigado por tudo que aprendi e pude melhorar tanto em minha vida profissional quanto pessoal.

RESUMO

A estrutura de indexação MRS é uma estrutura compacta em memória principal projetada para substituir o processo de varredura completa dos dados utilizado por algoritmos de pesquisa de similaridade, como o BLAST. O tamanho da estrutura de indexação MRS é tipicamente entre 1% e 2% do tamanho do banco de dados, possibilitando o seu uso em computadores pessoais, que possuem uma quantidade limitada de memória principal disponível. No entanto, avaliando o volume atual de dados biológicos e o crescimento dos bancos de dados biológicos, o tamanho da estrutura de indexação MRS pode exceder a quantidade de memória principal disponível. Para avaliar o desempenho da MRS em memória secundária, a estrutura de indexação foi adaptada para memória secundária, e testes de desempenho foram realizados com cromossomos provenientes do Genoma Humano. Os cromossomos humanos foram divididos em três grupos: cromossomos de volume pequeno, médio e grande. Para cada grupo, foram realizados testes de desempenho elevando-se o volume de seqüências armazenadas no banco de dados biológicos, com o intuito de analisar o desempenho da estrutura de indexação em duas fases: fase de construção do índice e fase de pesquisa no índice. Para a fase de construção do índice foi analisado o tempo gasto e o tamanho ocupado pelo índice, enquanto que para a fase de pesquisa no índice foi analisado o tempo gasto para o término de uma pesquisa no índice.

ABSTRACT

The MRS index structure is a compact structure in memory designed to replace the process of whole database scan used by similarity search algorithms, like BLAST. The size of the MRS index is typically about 1% to 2% of the database size, enabling its use in personal computers, which have a limit amount of available memory. However, evaluating the actual amount of biological data and the growing of the biological databases, the size of the MRS index structure may exceed the amount of available memory. To evaluate the performance of the MRS in disk, the index structure has been adapted to disk, and performance tests had been done with chromosomes from the Human Genome. The human chromosomes had been divided into three groups: chromosomes with low volume, medium volume and large volume. For each group, performance tests had been done uprising the volume of the sequences in the biological database, with the objective to analyze the performance of the index structure in two phases: index construction phase and index search phase. For the construction phase the elapse time and the size of the index had been analyzed, while for the index search phase the elapse time to finish a search in the index had been analyzed.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE GRÁFICOS

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

Atualmente, é muito comum encontrar aplicações do mundo real envolvendo dados de *string*. Nestas aplicações, é freqüente a necessidade de se realizar pesquisas sobre esses dados, destacando-se a comparação exata (*exact match*) e a comparação aproximada (*approximate match*). A comparação exata procura por *substrings* idênticas à *string* de consulta no banco de dados de *string*, enquanto que a comparação aproximada procura por *strings* que possuam padrões similares à *string* de consulta.

A pesquisa por comparação aproximada pode ser realizada através do alinhamento de *strings*. Durante o alinhamento de duas *strings*, são alinhados os caracteres de uma *string* com os caracteres da outra *string*. Para cada par de caracteres alinhado é atribuída uma pontuação baseada no grau de similaridade entre os caracteres. Essas pontuações são usadas para determinar qual é o melhor alinhamento entre as duas *strings*. Quanto maior a pontuação, melhor o alinhamento.

Existem, fundamentalmente, dois tipos de alinhamento: alinhamento global e alinhamento local. O alinhamento global busca alinhar ambas as *strings* considerando seus tamanhos completos, definido pelo maior valor de alinhamento obtido. Já o alinhamento local é definido pelo alinhamento de maior valor obtido de todas as *substrings*.

O alinhamento de *strings* é empregado, no contexto biológico, durante a pesquisa de similaridade entre seqüências biológicas. A similaridade entre seqüências de DNA (*deoxyribonucleic acid*) de diferentes organismos, por exemplo, pode corresponder a um relacionamento físico ou funcional entre os organismos.

A pesquisa de similaridade pode ser realizada com seqüências de nucleotídeos ou aminoácidos. As seqüências de nucleotídeos são compostas por caracteres pertencentes ao alfabeto $\sum = \{A, C, G, T, N\}$, onde N corresponde a nucleotídeos não identificados, enquanto que as seqüências de aminoácidos por caracteres pertencentes ao alfabeto $\sum = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y, Z, B, X\}$. Neste trabalho, exploramos a pesquisa de similaridade de nucleotídeos envolvendo alinhamentos locais.

Uma ferramenta bastante conhecida e utilizada por biólogos para a realização da pesquisa de similaridade é a ferramenta BLAST (*Basic Local Alignment Search Tool*). Assim como diversos outros algoritmos para alinhamento de seqüências, o algoritmo do BLAST (ALTSCHUL, 1990) (CAMERON, 2004) necessita realizar uma varredura completa do banco de dados para então executar a pesquisa de similaridade. O elevado volume de dados biológicos disponíveis e o crescimento acelerado do volume de dados dos bancos de dados biológicos em relação à memória principal disponível tornam impraticável o uso de técnicas baseadas apenas em memória principal. Por exemplo, o *GenBank*, um banco de dados de seqüências de nucleotídeos e proteínas construído pelo *National Center for Biotechnology Information* (NCBI), armazena mais de 100 *gigabytes* de dados biológicos. Ademais, o tamanho do *GenBank* tem dobrado a cada 15 meses.

Uma alternativa é o uso de estruturas de indexação em bancos de dados biológicos. Uma estrutura de indexação eficiente permite reduzir o espaço de busca, direcionando a pesquisa para regiões do banco de dados com elevado potencial de similaridade. Um fator importante é o tamanho ocupado pelas estruturas de indexação, uma vez que o tamanho de algumas estruturas são superiores ao próprio tamanho do banco de dados, tornando impraticável o seu uso. Um exemplo de estrutura compacta em memória principal é a estrutura de indexação *Multi Resolution String* (MRS) (KAHVECI, 2001). O seu tamanho ocupa aproximadamente entre 1% e 2% do tamanho do banco de dados.

Apesar do reduzido espaço de armazenamento em memória principal requerido pela estrutura de indexação MRS, com o crescente aumento do volume de dados dos bancos de dados biológicos a estrutura de indexação MRS pode não mais ser suportado pela memória principal, tendo parte do índice obrigatoriamente alocado em memória secundária. Com base nesta premissa, neste trabalho a MRS foi adaptada para gerenciar dados biológicos em memória secundária. Baseado no ambiente de teste proposto por Nakano (NAKANO, 2005), testes de desempenho foram realizados com cromossomos pertencentes ao Genoma Humano, divididos em três grupos: volume pequeno, médio e grande. Com o intuito de avaliar o desempenho da estrutura de indexação MRS em memória secundária, baterias de testes foram realizadas elevando-se o volume dos dados biológicos armazenados nos bancos de dados biológicos.

O trabalho está dividido em 7 capítulos. No Capítulo 2 são apresentados conceitos básicos relacionados ao alinhamento de *strings* e à pesquisa de similaridade de seqüências biológicas. O Capítulo 3 apresenta a estrutura de indexação MRS para, em seguida, no Capítulo 5 ser apresentada a implementação da estrutura de indexação MRS voltada para memória secundária. O Capítulo 4 descreve diversas propostas alternativas de estruturas de indexação. No Capítulo 6 são apresentados os testes de desempenho realizados sobre a estrutura de indexação MRS com um elevado volume de dados. E por fim, no Capítulo 7 são apresentadas as conclusões deste trabalho, bem como sugestões para trabalhos futuros.

CAPÍTULO 2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os conceitos básicos relacionados à pesquisa de *strings* em bancos de dados de *strings* por meio da comparação exata e aproximada. Também é descrita uma das inúmeras aplicações de comparação de *strings*, a pesquisa de similaridade em bancos de dados biológicos. Na seção 2.1 são apresentados conceitos e algoritmos relacionados a alinhamentos de *strings*, na seção 2.2 é inserido o contexto de banco de dados biológicos para então, na seção 2.3 descrever a aplicação da pesquisa de similaridade em banco de dados biológicos. Na seção 2.4 é descrito o algoritmo de uma das ferramentas mais populares para pesquisa de similaridade com dados biológicos e, por fim, na seção 2.5 é apresentada a motivação em se desenvolver estruturas de indexação eficientes para dados biológicos.

2.1 Alinhamentos

Dados de *string* naturalmente existem em aplicações do mundo real incluindo dados biológicos, dados da *web* e seqüências de eventos. Estas aplicações geralmente envolvem banco de dados com elevado volume que tendem a crescer exponencialmente. Pesquisar nestes volumosos bancos de dados de *strings* se tornou uma prática comum e a necessidade de mecanismos de indexação eficientes para *strings* se tornou imprescindível. Por exemplo, biólogos freqüentemente necessitam pesquisar por amostras similares de um determinado DNA ou região de uma proteína em bancos de dados volumosos de seqüências decodificadas. Atualmente, a quantidade de seqüências mapeadas excede 100 Gigabases (BENSON, 2006) e continua a crescer em um ritmo exponencial. Em especial, o tamanho do genoma humano

ocupa aproximadamente 30 Gigabases (BENSON, 2006). Entretanto, a falta de um mecanismo de indexação efetivo faz com que arquivos sejam utilizados como formato padrão para armazenar as enormes seqüências biológicas.

A pesquisa de similaridade em um banco de dados de *string* pode ser classificada em duas categorias (YANG, 2004): comparação exata (*exact match*) ou comparação aproximada (*approximate match*). A pesquisa de comparação exata procura por *substrings* no banco de dados que sejam exatamente idênticas ao padrão da *string* de pesquisa. Já a pesquisa de comparação aproximada procura por um padrão no banco de dados de *string* onde uma delas sofreu algum tipo de corrupção indesejada, como substituições entre determinados símbolos, algum grau de desemparelhamento, podendo até considerar caracteres curinga na *string* de consulta (YANG, 2004) (NAVARRO, 2000).

O grau de similaridade entre duas *strings* pode ser definido através de uma métrica de distância, a *edit distance* (distância de edição). Uma *string* pode se transformar em uma outra *string* utilizando três tipos de operações de edição: inserção, remoção e substituição. A *edit distance* entre duas *strings* é definida pelo número mínimo de operações de edição necessário para transformar uma *string* na outra.

O alinhamento de *strings* é realizado através do emparelhamento dos caracteres de uma determinada *string* com os caracteres de uma outra *string* em ordem crescente. Para cada par de caracteres é atribuído um valor baseado em suas similaridades. O valor de um alinhamento é determinado pela soma dos valores de todos os pares de caracteres. Em princípio, existem diversos métodos de se alinhar duas seqüências, mas na prática, um método é mais utilizado que qualquer outro. Atualmente, o método mais popular é o utilizado pela ferramenta BLAST (seção 2.4).

O alinhamento global e o alinhamento local podem ser determinados em tempo O(*m***n*) utilizando programação dinâmica (KAHVECI, 2001) (KORF, 2003).

Alinhamento Global

Um algoritmo conhecido para alinhamento global de *strings* que utiliza programação dinâmica é o algoritmo Needleman-Wunsch (NEEDLEMAN-WUNSCH, 1970). O algoritmo utiliza uma matriz bidimensional, na qual cada célula corresponde a um par de letras, uma letra de cada *string*. Em cada célula são armazenados dois valores: uma pontuação e um ponteiro. Durante o alinhamento, é adotado um esquema de pontuação. Um esquema simples consiste de: +1 para letras que correspondem, -1 para letras que não correspondem e -1 para *gap* (espaço). O valor da pontuação é derivado do esquema de pontuação, enquanto que o ponteiro é um indicador direcional que pode apontar para cima, esquerda ou diagonalmente para cima e para esquerda. Este esquema será adotado em nosso exemplo de alinhamento global entre as *strings* PROCESSAMENTO e ACESSO. O algoritmo é divido em 3 fases: inicialização, preenchimento e rastreio (*trace-back*). Ao final, é obtido o alinhamento de maior pontuação entre as duas *strings*.

Inicialização

Na fase de inicialização, pontuações são atribuídas para a primeira coluna e linha, conforme Figura 2.1 A pontuação de cada célula é modificada pelo valor do *gap* multiplicado pela distância da origem. *Gaps* podem estar presentes no início de cada *string*, e sua pontuação é a mesma para qualquer localidade. Todos os ponteiros apontam para a origem, o que assegura que os alinhamentos convirjam para a origem.

P R O C E S S A M E N T O

	0	← -1	← -2	← -3	← -4	← -5	← -6	← -7	← -8	← -9	← -10	← -11	← -12	← -13
A	↑-1													
C	↑-2													
E	↑-3													
S	↑-4													
S	↑-5													
O	↑-6													

Figura 2.1: Inicialização da matriz de alinhamento

Preenchimento

Na fase de preenchimento, a matriz é preenchida com pontuações e ponteiros através de uma operação que necessita das pontuações das células vizinhas da diagonal, vertical e horizontal. Com o auxílio destas células vizinhas, são calculadas três pontuações: a pontuação de correspondência, a pontuação de *gap* vertical e a pontuação de *gap* horizontal. A pontuação de correspondência é a soma da pontuação da célula diagonal com o valor de uma correspondência (+1) ou não correspondência (-1). Já a pontuação de *gap* horizontal é a soma da célula à esquerda com a pontuação para *gap*, enquanto que a pontuação de *gap* vertical é a soma da célula acima com a pontuação para *gap*. Após o cálculo destas três pontuações, são atribuídos à célula a pontuação de maior valor e um ponteiro na direção da pontuação máxima. Esta operação é realizada até que toda a matriz esteja preenchida, e em cada célula contenha uma pontuação e um ponteiro para o melhor alinhamento possível.

Observando a matriz de alinhamento (Figura 2.1), é possível identificar apenas uma célula em que se encontram disponíveis as três células vizinhas. Esta é a célula onde o

processo de preenchimento é inicializado. Para esta primeira célula, a pontuação de preenchimento é a soma da célula diagonal precedente (pontuação = 0) com a pontuação para o alinhamento de P e A (-1), resultando em uma pontuação igual a -1. Já a pontuação de *gap* horizontal é a soma da pontuação da célula à esquerda (-1) com a pontuação do *gap* (-1), resultando em uma pontuação igual a -2. Calculando a pontuação de *gap* vertical também obtemos uma pontuação igual a -2. Sendo assim, a maior pontuação é a pontuação de correspondência (-1), que é atribuída à primeira célula e seu ponteiro aponta para a diagonal (Figura 2.2). Depois de computada a primeira célula, se torna possível computar a célula à direita ou a célula abaixo. Este processo segue até que toda a matriz de alinhamento esteja completamente preenchida.

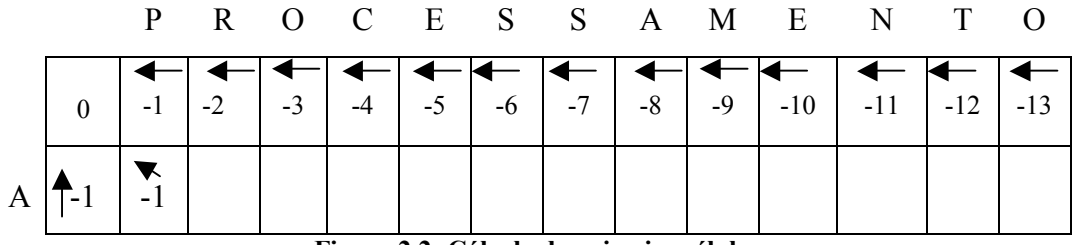


Figura 2.2: Cálculo da primeira célula

Durante o cálculo da pontuação de uma determinada célula, pode ocorrer o caso em que existe uma igualdade nas pontuações calculadas. Por exemplo, calculando a pontuação da próxima célula à direita, a pontuação de correspondência é a soma da pontuação da célula diagonal (-1) com a pontuação do alinhamento de R e A (-1), resultando em uma pontuação igual a -2. Esta mesma pontuação é obtida quando calculado a pontuação de *gap* horizontal, sendo possível obter uma pontuação máxima de -2 tanto pela diagonal quanto pela esquerda. Para solucionar este tipo de impasse é necessário realizar uma escolha consistente e arbitrária, como, por exemplo, sempre escolher a diagonal ao invés de um *gap* (solução aplicada em nosso exemplo). Esta solução é apresentada na Figura 2.3.

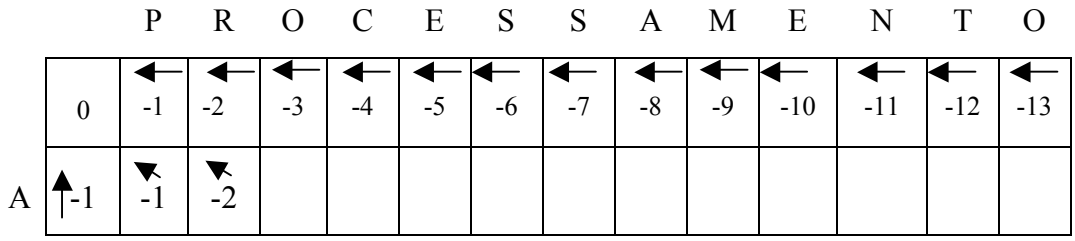


Figura 2.3: Cálculo da segunda célula

Após o preenchimento completo da matriz de alinhamento utilizando o processo de maximização para cada célula, a matriz da Figura 2.4 é obtida.

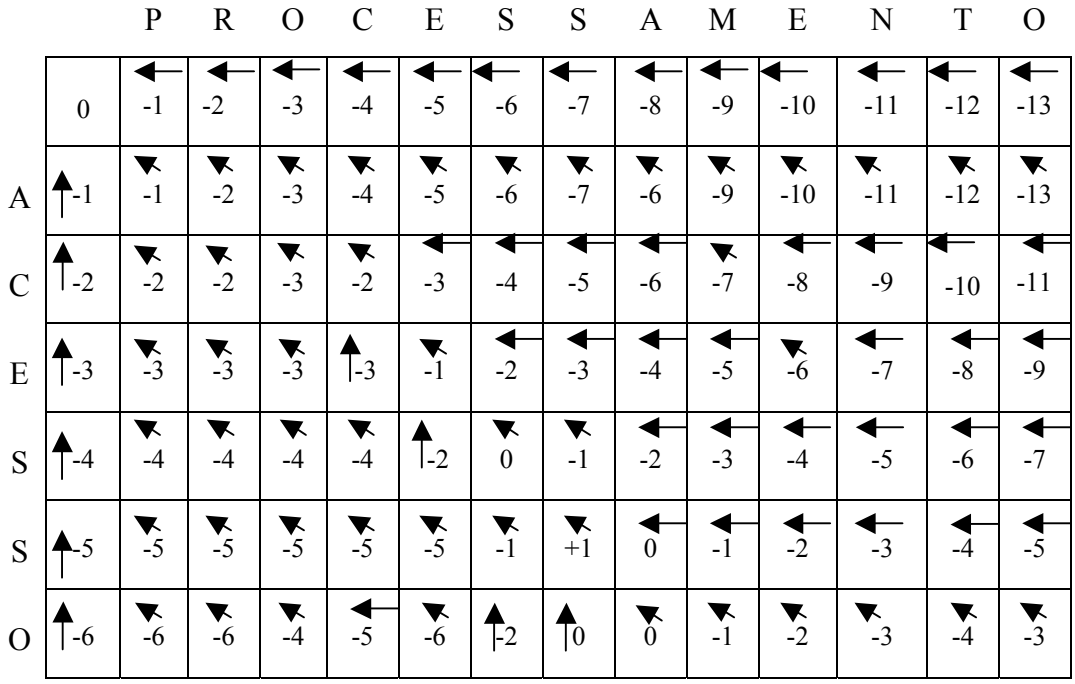


Figura 2.4: Matriz de alinhamento preenchida

Rastreio

O passo de rastreio possibilita recuperar o alinhamento global da matriz de alinhamento. Neste passo, o alinhamento resultante é obtido iniciando pelo canto inferior direito e seguindo os ponteiros até alcançar a origem, a cada célula, armazenar as letras

correspondentes ou hífens para os símbolos de *gap*. Como o processo é realizado de trás para frente, é necessário inverter o alinhamento obtido. O alinhamento resultante é:

P	R	O	C	E	S	S	A	M	E	N	T	O
-	-	A	C	E	S	S	-	-	-	-	-	O

A Figura 2.5 apresenta as *strings* alinhadas dentro da matriz de alinhamento, note que toda letra de cada *string* é alinhada com uma letra ou um *gap*.

		P	R	O	C	E	S	S	A	M	E	N	T	O
		P	R											
A				O A										
C					C C									
E						E E								
S							S S							
S								S S	A	M	E	N	T	
O														O O

Figura 2.5: Exemplo de uma matriz de alinhamento

Alinhamento Local

O algoritmo Needleman-Wunsch encontra alinhamentos ótimos de seqüências completas, que são alinhamentos de pontuação máxima utilizando programação dinâmica. Entretanto, é freqüentemente necessário encontrar fragmentos de *strings* altamente similares.

Identificar essas regiões de similaridade, de modo que a pontuação do alinhamento dessas duas regiões locais seja maximizada é o objetivo do alinhamento local.

Um modo de visualizar o alinhamento local de *strings* é considerar que a eliminação de prefixos e sufixos é livre. Caso um alinhamento global possua pontuações negativas em seu final, é possível eliminar este final para se obter um melhor alinhamento local. Conseqüentemente, se um mesmo esquema de pontuação é utilizado, a pontuação de um alinhamento local de duas *strings* é sempre maior ou igual à pontuação do alinhamento global. Entretanto, existem casos em que o alinhamento local ótimo pode não fazer parte do alinhamento global, isto ocorre quando um determinado fragmento de uma das *strings* é alinhada a um outro fragmento da outra *string* que não estava previamente alinhado no alinhamento global (BROWN, 2003).

Para computar o alinhamento local entre duas *strings* é necessário realizar três modificações no algoritmo Needleman-Wunsch, obtendo o algoritmo denominado Smith-Waterman:

- As bordas da matriz de alinhamento são inicializados com 0 ao invés de penalidades de *gap* crescentes;
- A pontuação máxima nunca é menor que 0, e nenhum ponteiro é gravado a não ser que sua pontuação seja superior a 0; e
- O rastreamento inicia a partir da pontuação mais elevada na matriz de alinhamento e termina em uma pontuação igual a 0.

Essas modificações ocasionam um profundo efeito no comportamento do algoritmo.

Utilizando as mesmas *strings* e o mesmo esquema de pontuação do alinhamento global, é apresentada a matriz de alinhamento resultante na Figura 2.6.

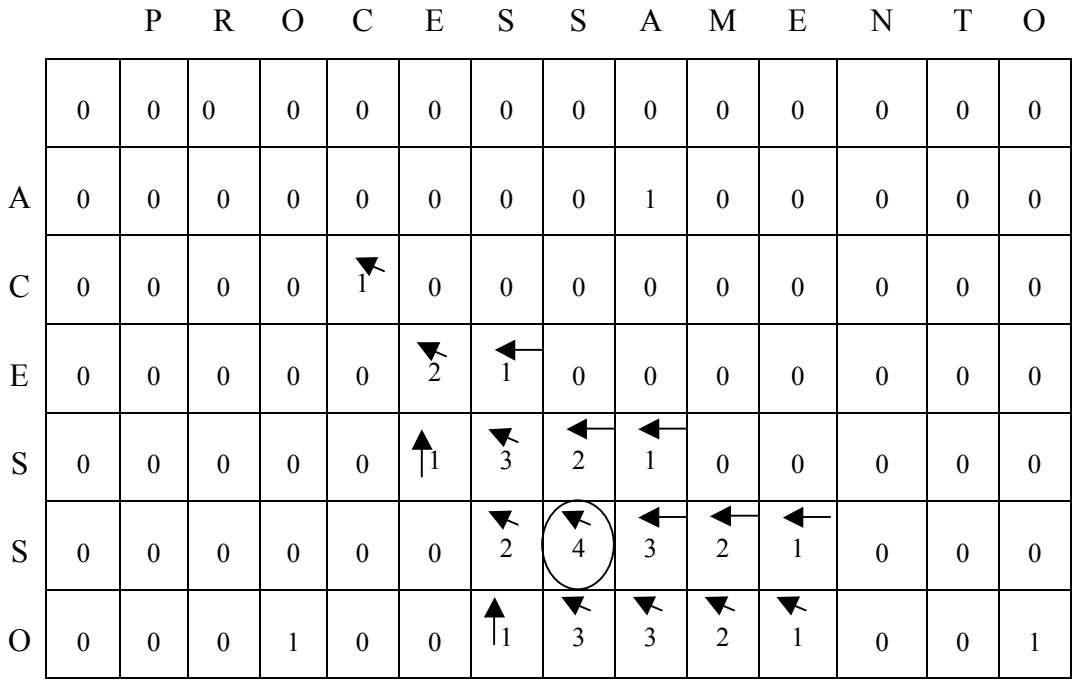


Figura 2.6: Matriz de alinhamento Smith-Waterman

O resultado do alinhamento local é obtido a partir da célula com um círculo que corresponde à célula de pontuação máxima. Podem ocorrer casos em que existam mais de uma célula com a mesma pontuação máxima, e neste caso, é necessário algum tipo de decisão arbitrária. Realizando o rastreio a partir da pontuação máxima até uma pontuação zero, é criado o alinhamento de trás para frente, necessitando que a *string* resultante seja invertida. O alinhamento local obtido é:

C	E	S	S
C	E	S	S

Alinhamentos com Modificações no Gap

No contexto biológico, o objetivo do alinhamento de seqüências é determinar se duas seqüências de DNA ou proteína são homólogas, possuem uma mesma origem embriológica. Mas, somente realizar o alinhamento não é suficiente para dizer precisamente se duas

seqüências são homólogas sem que haja um conhecimento íntimo da origem das seqüências biológicas, mesmo que estas sejam muito similares em suas seqüências (BROWN, 2003). Entretanto, o número de mutações evolucionárias para transformar uma seqüência na outra pode ser usado para estimar a probabilidade de que duas seqüências sejam homólogas.

É comum observar regiões consistindo completamente de caracteres de espaço em um alinhamento de seqüências. Em seqüências homólogas, esses *gaps* correspondem a mutações de remoção ou inserção. Se for utilizado um esquema de pontuação similar ao utilizado no exemplo anterior no qual é pontuada uma constante negativa para alinhar qualquer letra a um *gap*, o custo do *gap* é proporcional ao seu tamanho. Este não é um esquema muito bom para o contexto biológico, pois *gaps* tendem a ocorrer freqüentemente. Uma típica solução é empregar *affine gaps*, onde a pontuação para um *gap* de tamanho i é $o + i * e$. Neste esquema, a penalidade para a abertura de *gap* o é tipicamente muito mais negativa que a penalidade de extensão e que é acrescentado a cada *gap* adicional.

2.2 Bancos de Dados Biológicos

Bancos de Dados (BD) são utilizados para armazenar informações de forma organizada, visando facilitar consultas, inserções e remoções de dados. Os bancos de dados biológicos (BDBs) são usados na área da biologia molecular para armazenar diversos tipos de dados referentes aos organismos vivos, tais como seqüências de nucleotídeos e de aminoácidos, estruturas de proteínas, mapas genéticos, identificação e anotação de genes, literatura associada, e qualquer informação derivada destas ou necessárias para a compreensão das mesmas.

É crescente o número de BDBs públicos, que permitem o acesso de qualquer usuário aos seus dados para obter informações a respeito de um determinado organismo que esteja

disponível, sendo interessante o fato de que a geografia dos BDBs também continua a se expandir, com países disponibilizando seus primeiros BDBs (GALPERIN, 2004).

Porém, não é uma tarefa fácil manter esses BDBs públicos. Muitos projetos são descontinuados, mesclados em projetos maiores, ou convertidos para acesso comercial. É preciso gerenciar para sobreviver, mesmo que seja necessário mudar seu esquema de fundos ou migrar de uma instituição para outra.

A maioria dos BDBs se organiza em portais da *Web*, sendo um dos principais o NCBI (http://www.ncbi.nlm.nih.gov). Tais portais disponibilizam, além de informações de diversos organismos, recursos para pesquisar diretamente os dados, sem a necessidade de programas específicos para cada BD. O NCBI disponibiliza o BD chamado *GenBank* (BENSON, 2004) que, além das suas próprias seqüências, se mantém atualizado com os principais BDBs do mundo: a biblioteca de dados do *European Molecular Biology Laboratory* (EMBL) (STOESSER, 2002) localizado no Reino Unido e o DNA *Data Bank of Japan* (DDBJ) (TATENO, 2002) (MIYAZAKI, 2004), através de um projeto de colaboração entre os Bancos de Dados de Seqüência de Nucleotídeos Internacionais, para os quais dados são trocados diariamente, assegurando uma coleção uniforme de seqüências.

Com o avanço da tecnologia, os processos de seqüenciamento se tornaram cada vez mais rápidos e práticos, o que elevou o volume de dados coletados e armazenados. O volume de dados, que inicialmente era pequeno e facilmente armazenado em arquivos texto, aumentou em um nível que se tornou indispensável o uso de um Sistema Gerenciador de Banco de Dados (SGBD) para o gerenciamento de grandes volumes de dados.

Dentre os maiores BDBs disponíveis atualmente, é possível destacar: o *GenBank* (BENSON, 2004), o EMBL (BAKER, 2000), o DDBJ (MIYAZAKI, 2004), o *Genome Sequence Database* (GSDB) (FASMAN, 1996) (HARGER, 2000) e o *Protein Information Resource* (PIR) – *International Protein Sequence Database* (PSD) (BARKER, 2000).

É comum encontrarmos BDBs utilizando modelos de dados diferentes. Inicialmente, o modelo relacional foi o escolhido para abrigar os dados biológicos, sendo possível encontrar diversos projetos que utilizem esta tecnologia, como: o FlyBase (FLYBASE CONSORTIUM, 2003) implementado em Sybase, o SMART (Simple Modular Architecture Research Tool) (LETUNIC, 2002) implementado em PostgreSQL, o XPro (GOPALAN, 2004) implementado em MySQL, entre outros. Também existem os BDBs que utilizam outras tecnologias, como a tecnologia de banco de dados orientado a objetos (MEWES, 2000) (MAIDAK, 1999) (BARILLOT, 1999).

O grande volume de dados exige que ferramentas sejam disponibilizadas para viabilizar consultas aos dados biológicos, além de que é necessário que a interface de acesso para tais ferramentas seja intuitiva e de fácil configuração. A maioria dos BDBs disponibilizam, além de seus dados, um conjunto de ferramentas para consulta, como é o caso do NCBI que oferece a ferramenta BLAST para a execução de pesquisas rápidas de similaridades de seqüências. Dentre outros exemplos (WHEELER, 2001) é possível destacar o EMBL (STOESSER, 2002) que oferece o FASTA e o DDBJ (MIYASAKI, 2004) que oferece o SQmatch.

Ao contrário dos primórdios dos BDBs, em que cada banco de dados era criado e mantido isoladamente, atualmente há a preocupação em integrar estes BDBs. Existem diversos sistemas que integram BDBs (STEVENS, 2000) (EILBECK, 2003), apesar das dificuldades encontradas, pois por não existir um padrão em nenhum nível de abstração, como o modelo de dados conceitual ou lógico, ou a linguagem de consulta, cada BDB é diferente um do outro. E além da heterogeneidade estrutural e de representação já destacadas, também existe a heterogeneidade semântica, em que conceitos básicos, como gene, possuem diferentes definições (SCHULZE-KREMER, 1997).

O grande volume de dados biológicos resultante do seqüenciamento de genomas de diversos organismos vivos e a diversidade de informações extraídas da análise das seqüências, tais como seqüências de nucleotídeos, seqüências de aminoácidos, estruturas de proteínas e árvores filogenéticas, exigem a separação dos dados a serem armazenados em BDBs específicos, como os já apresentados.

2.3 Similaridade de Seqüências Biológicas

Os algoritmos para alinhamentos de seqüências de caracteres apresentados anteriormente utilizam a pontuação como métrica para determinar o melhor alinhamento obtido. Em se tratando de similaridade de seqüências biológicas, torna-se ainda necessário diferenciar um alinhamento obtido aleatoriamente de um biologicamente significativo. Caso o alinhamento seja significativo, avalia-se a sua importância. Para a efetivação desta avaliação, uma matriz de pontuação, também conhecida como matriz de substituição, deve ser construída.

Para o alinhamento de seqüências de DNA, a matriz de pontuação é tipicamente simples. Um exemplo consiste na matriz de pontuação padrão utilizada pelo programa de pesquisa de similaridade de seqüências de DNA BLASTN cuja pontuação para bases iguais é 1, bases diferentes é -3 e base com *gap* é -1. Já penalidade por abertura de *gap* é diferenciada, por padrão -5.

Em se tratando do alinhamento de seqüências de aminoácidos, uma matriz de pontuação, construída com base em alinhamentos de seqüências conhecidamente homólogas, é utilizada para determinar a probabilidade de uma determinada substituição ocorrer ao acaso (aleatoriamente) ou por evolução. Existem 20 aminoácidos diferentes: A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y e V, sendo que esta matriz de pontuação, muitas vezes uma matriz 20x20, representa a equivalência de cada uma dos 210 possíveis pares de aminoácidos

(190 pares de aminoácidos diferentes + 20 pares de aminoácidos idênticos). As matrizes de pontuação mais comumente utilizadas são as matrizes PAM e BLOSUM.

Matrizes PAM (Point Accepted Mutation)

As matrizes PAM foram desenvolvidas por Dayhoff e colaboradores (DAYHOFF, 1978) e estabelecem esquemas de pontuação para grupos de seqüências relacionadas. Existem diversas matrizes PAM, cada uma com um sufixo numérico. A matriz PAM1 foi construída com um conjunto de proteínas com 85% ou mais de identidade (KORF, 2003) e equivale a uma alteração média de 1% em todas as posições de aminoácidos. As outras matrizes PAM são obtidas a partir da matriz PAM1, multiplicando-a por ela mesma sucessivamente conforme o número desejado: 100 vezes para a PAM100, 160 vezes para a PAM160, e da mesma forma para qualquer outra numeração.

As pontuações foram determinadas utilizando uma abordagem conhecida como *log-odds*, na qual as pontuações correspondem ao logaritmo neperiano da proporção de probabilidade de ocorrência significativa (*target frequency*) em relação à probabilidade de ocorrência aleatória (*background frequency*).

Por existirem poucos dados de seqüências de proteínas na época em que foram criadas (anos 70), as matrizes PAM são uma boa opção para explorar grandes distâncias evolucionárias (KORF, 2003).

Matrizes BLOSUM (BLOcks SUbstitution Matrix)

As matrizes BLOSUM foram estabelecidas nos anos 90, quando existia um maior número de dados de seqüências de proteínas disponíveis, permitindo uma abordagem mais empírica que a utilizada para a construção das matrizes PAM. Para a construção destas

matrizes são extraídos segmentos sem *gaps* (blocos) de um conjunto de famílias de proteínas multiplamente alinhadas, para então agrupar esses blocos com base em seus percentuais de identidade. O valor associado à uma matriz BLOSUM indica o percentual de identidade necessário para que uma seqüência seja introduzida em um mesmo agrupamento. Em uma matriz BLOSUM50 (Figura 2.7), por exemplo, todas as seqüências possuem ao menos 50% de identidade com algum membro do bloco.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	5	-2	-1	-2	-1	-1	-1	0	-2	-1	-2	-1	-1	-3	-1	1	0	-3	-2	0
R	-2	7	-1	-2	-4	1	0	-3	0	-4	-3	3	-2	-3	-3	-1	-1	-3	-1	-3
N	-1	-1	7	2	-2	0	0	0	1	-3	-4	0	-2	-4	-2	1	0	-4	-2	-3
D	-2	-2	2	8	-4	0	2	-1	-1	-4	-4	-1	-4	-5	-1	0	-1	-5	-3	-4
C	-1	-4	-2	-4	13	-3	-3	-3	-3	-2	-2	-3	-2	-2	-4	-1	-1	-5	-3	-1
Q	-1	1	0	0	-3	7	2	-2	1	-3	-2	2	0	-4	-1	0	-1	-1	-1	-3
E	-1	0	0	2	-3	2	6	-3	0	-4	-3	1	-2	-3	-1	-1	-1	-3	-2	-3
G	0	-3	0	-1	-3	-2	-3	8	-2	-4	-4	-2	-3	-4	-2	0	-2	-3	-3	-4
H	-2	0	1	-1	-3	1	0	-2	10	-4	-3	0	-1	-1	-2	-1	-2	-3	2	-4
I	-1	-4	-3	-4	-2	-3	-4	-4	-4	5	2	-3	2	0	-3	-3	-1	-3	-1	4
L	-2	-3	-4	-4	-2	-2	-3	-4	-3	2	5	-3	3	1	-4	-3	-1	-2	-1	1
K	-1	3	0	-1	-3	2	1	-2	0	-3	-3	6	-2	-4	-1	0	-1	-3	-2	-3
M	-1	-2	-2	-4	-2	0	-2	-3	-1	2	3	-2	7	0	-3	-2	-1	-1	0	1
F	-3	-3	-4	-5	-2	-4	-3	-4	-1	0	1	-4	0	8	-4	-3	-2	1	4	-1
P	-1	-3	-2	-1	-4	-1	-1	-2	-2	-3	-4	-1	-3	-4	10	-1	-1	-4	-3	-3
S	1	-1	1	0	-1	0	-1	0	-1	-3	-3	0	-2	-3	-1	5	2	-4	-2	-2
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	2	5	-3	-2	0
W	-3	-3	-4	-5	-5	-1	-3	-3	-3	-3	-2	-3	-1	1	-4	-4	-3	15	2	-3
Y	-2	-1	-2	-3	-3	-1	-2	-3	2	-1	-1	-2	0	4	-3	-2	-2	2	8	-1
V	0	-3	-3	-4	-1	-3	-3	-4	-4	4	1	-3	1	-1	-3	-2	0	-3	-1	5

Figura 2.7: BLOSUM 50 Fonte: Yang (2004)

2.4 BLAST

Durante a pesquisa de similaridade, seqüências são alinhadas e é aplicada uma pontuação para cada posição onde ocorrer uma correspondência ou não entre as seqüências. Utilizando a técnica de programação dinâmica, este processo se torna custoso proporcionalmente ao tamanho das seqüências a serem comparadas.

A ferramenta BLAST (Altschul, 1990) (Cameron, 2004) (Nakano, 2004) surgiu com a finalidade de solucionar o problema da comparação de uma seqüência com um BDB contendo

milhares de seqüências. Por seu desempenho e resultados serem bastante satisfatórios, se tornou bastante conhecida e utilizada.

O algoritmo heurístico do BLAST utiliza uma tabela de *words*, na qual cada entrada na tabela corresponde a uma subseqüência da seqüência de consulta (*query sequence*). Com base nesta tabela, são gerados alinhamentos locais, sendo que cada alinhamento obtido com sucesso corresponde a um *hot spot*, uma região que é estendida em busca de um melhor alinhamento.

Este algoritmo parte do princípio de que um alinhamento representante de uma relação de analogia verdadeira entre duas seqüências possui, no mínimo, uma palavra exata em comum. A busca por palavras exatas pode ser realizada através da pré-indexação das palavras da seqüência de consulta e das seqüências a serem comparadas, não necessitando a comparação de seqüências que não estejam relacionadas.

O BLAST é um conjunto de serviços para a pesquisa de similaridade oferecida pelo NCBI. Cada serviço é específico para um determinado tipo de pesquisa, dentre os serviços disponíveis:

- **Blastp**: compara uma seqüência de consulta formada por aminoácidos contra um banco de dados de seqüências de proteína;
- **Blastn**: compara uma seqüência de consulta formada por nucleotídeos contra um banco de dados de seqüências de nucleotídeos;
- **Blastx**: compara uma seqüência de consulta formada por nucleotídeos traduzida em todos os quadros de leitura contra um banco de dados de seqüências de proteínas;
- **Tblastn**: compara uma seqüência de consulta formada por aminoácidos contra um banco de dados de seqüência de nucleotídeos traduzido em todos os quadros de leitura; e

- **Tblastx**: compara uma seqüência de consulta formada por nucleotídeos traduzida nos seis quadros de leitura contra seqüências de nucleotídeos de um BDB traduzidas nos seis quadros de leitura.

Teoria Estatística

A teoria estatística empregada no algoritmo do BLAST, durante a comparação entre duas seqüências, faz com que o algoritmo procure por segmentos de mesmo tamanho em cada seqüência que quando alinhado um ao outro sem *gaps*, formam pares localmente ótimos, cujas pontuações não podem ser melhoradas através de extensões ou reduções (ALTSCHUL, 1997). Esses pares localmente ótimos são conhecidos como HSPs (*High Scoring Pairs*).

Segundo a teoria estatística apresentada por Altschul *et. al.* (ALTSCHUL, 1997), durante o alinhamento entre duas seqüências:

- Em um modelo de proteína simples, os aminoácidos ocorrem aleatoriamente em todas as posições com probabilidade *Pi*, onde *i* é a posição do aminoácido.
- A pontuação esperada para dois aminoácidos aleatórios $\sum_{i,j} P_i P_j s_{ij}$ é negativa, sendo P_i e P_j as probabilidades de *i* e *j* ocorrerem e s_{ij} a pontuação para alinhar cada par de aminoácidos *i* e *j*.
- Utilizando dois parâmetros, λ e *K*, onde *K* é uma constante adquirida por meio de séries geométricas convergentes e λ é obtida através de uma expressão dependente somente da matriz de substituição e das probabilidades de ocorrência dos aminoácidos, ambas calculadas pela teoria básica, é possível converter pontuações HSP nominais em pontuações HSP normalizadas, possibilitando a comparação de todo o sistema de pontuação sob uma perspectiva estatística. A pontuação normalizada *S'* para um HSP é dada por:

$$S' = \frac{\lambda S - \ln K}{\ln 2}$$

- O valor da pontuação normalizada é expressa em *bits*, podendo ser denominada *Bit Score S'*. Quando duas seqüências de proteínas aleatórias, com tamanhos *m* e *n* suficientes, são comparadas, o número *E* (*Expected Value*) de HSPs distintos com pontuação normalizada de pelo menos *S'* que possa ocorrer por chance é determinado por:

  $$E = N / 2^{S'}$$

  Sendo N = *mn* o espaço de pesquisa, onde *m* corresponde ao tamanho da seqüência de consulta e *n* o tamanho da seqüência do BDB. Invertendo esta equação, obtemos $S' = \log_2(N/E)$, a pontuação normalizada necessária para alcançar um determinado valor *E* (*E-value*).

- Devido aos melhoramentos feitos nas heurísticas utilizadas por BLAST bem como a inclusão de tratamento para *gaps* nas seqüências, os parâmetros λ e *K* não podem mais ser adquiridos analiticamente. Ambos são estimados por meio de processos reais ou de simulação executados antecipadamente.

O algoritmo do BLAST

O algoritmo do BLAST pode ser descrito em quatro passos (Cameron, 2004) (Nakano, 2003) (Nakano, 2004):

- **1º Passo**: consiste na criação da tabela de *words*, na qual todas as palavras formadas por *w* caracteres (*w-mer*), com algum *w-mer* da *query sequence*, e cujo *score* seja superior a um limiar *T*.

- **2º Passo**: baseado na tabela de *words*, é realizado a busca de *hits* no BDB (i.e., alinhamento da palavra com a seqüência pesquisada), onde cada *hit* corresponde a uma semente, que é um segmento da *string* a ser posteriormente estendido. Cada posição do *hit* corresponde a uma identidade.

- **3º Passo**: as sementes são estendidas para ambos os lados considerando-se *gaps*, desde que o *score* da semente possua um valor maior que um limite *T* e não exista uma outra semente a uma distância máxima. A semente é estendida até que o *score* gerado pelo alinhamento com *gap* atinja o valor do parâmetro de queda *X*. Este parâmetro é responsável por controlar a sensibilidade e velocidade do algoritmo: quanto mais alto o valor de *X*, maior é a sensibilidade do alinhamento, mas menor é a velocidade do processo. Caso o alinhamento resultante gere um score maior que *Sg*, este passa para o último passo.

- **4º Passo**: no último passo, os alinhamentos finais possuem seus *scores* redefinidos, o caminho evolucionário de cada alinhamento é gravado, e o valor do parâmetro *X* é elevado com o intuito de se procurar um alinhamento de maior score. O número de alinhamentos processados durante este último passo é limitado por dois fatores importantes. Primeiro, é determinado pelo número de seqüências que possuem um valor de *score* acima de um valor-*E* (*E-value*) no terceiro passo, e segundo por um parâmetro B que limita o número total de alinhamentos a ser exibido ao usuário.

Para o cálculo dos *scores*, uma matriz de substituição é utilizada, como exemplo a PAM ou a BLOSUM (Korf, 2003). A estratégia utilizada na penalidade de *gaps* é uma função linear do comprimento *x* do *gap*:

$$Pen_{gap}(x) = a + bx$$

onde *a* é a penalidade de se iniciar um *gap* e *b* a penalidade por cada unidade de *gap*.

O algoritmo do BLAST possui diversos parâmetros que podem ser ajustados de acordo com o objetivo de cada pesquisa. Variando os parâmetros é possível aumentar ou diminuir a sensibilidade de uma consulta, tendo em vista que quanto maior a sensibilidade, maior é o número de sementes a serem analisadas, aumentando o tempo de processamento da pesquisa.

2.5 Estruturas de Indexação em Bancos de Dados Biológicos

Com o crescente aumento do volume de dados de seqüências de DNA depositadas em BDB públicos, a pesquisa de similaridade se tornou a operação básica na biologia molecular. Mesmo os algoritmos mais rápidos, segundo os quais a pesquisa é realizada, de maneira geral, seqüencialmente, por varredura completa do BD utilizando uma abordagem de aproximação para identificar o conjunto de seqüências desejadas (Altschul, 1990), atingiram seus limites quando aplicados em comparações entre seqüências de BDBs volumosos (Williams, 2002).

Um método geral para reduzir custos de pesquisa é armazenar uma abstração ou indexação que possa ser usada para estimar ampla similaridade para uma consulta. O custo é o espaço necessário para o armazenamento do índice.

Por sua vez o ganho potencial é que trazendo um volume limitado de dados para a memória principal deve possibilitar a identificação de um pequeno número de seqüências possivelmente similares, reduzindo tanto o tráfego de dados do disco para a memória principal e vice-versa quanto o processamento computacional necessário para responder a consulta.

O desafio de se indexar seqüências de DNA ou de proteínas é o fato de que seqüências biológicas são pesquisadas de forma não exata, mas utilizando técnicas de comparação aproximada. A construção de uma estrutura de indexação apropriada é capaz de acelerar o

processo de pesquisa de similaridade. A motivação e viabilidade deste trabalho se baseiam nas seguintes observações:

- Uma estrutura de indexação pode ajudar a direcionar a pesquisa a uma porção do BDB com elevado potencial de similaridade, evitando a varredura completa dos dados. Este é um fator importante para melhorar o desempenho de uma pesquisa, tendo em vista que esta varredura pode demorar horas para analisar todas as seqüências de um BDB volumoso e a economia computacional aumenta proporcionalmente com o volume dos dados; e

- Estruturas de indexação apropriadas sobre grandes seqüências podem levar muitas horas para serem construídas, se tornando ineficiente construí-las para cada pesquisa. Por outro lado, as seqüências são relativamente estáveis e a maioria das atualizações refere-se a adições de novas seqüências, de maneira que seja possível amortizar o custo de construção da estrutura de indexação sobre diversas consultas mantendo a estrutura persistente (i.e., em memória secundária).

2.6 *Buffer*-pool (CIFERRI, 2002)

Um *buffer-pool* consiste de um bloco de RAM utilizado especificamente para armazenar de forma temporária páginas de disco relativas aos objetos indexados residentes em memória secundária. O armazenamento replicado de páginas objetiva reduzir o número de acessos a memória secundária e conseqüentemente melhorar o desempenho da estrutura de indexação. Para isto, quando uma página de disco é referenciada, primeiramente é verificado se a página de disco encontra-se armazenada no *buffer-pool*, sendo que a sua presença no *buffer* evita um acesso a disco. Na literatura de banco de dados, esta referência é conhecida como referência lógica. Caso a página de disco não se encontre armazenada no *buffer-pool*, situação tipicamente conhecida como "falha de página" ou "falta de página" (i.e., *page fault*),

esta é transferida da memória secundária para o *buffer-pool*, requisitando assim um acesso à memória secundária. Esta referência, por sua vez, é conhecida como referência física. No entanto, antes de se completar a transferência, deve-se alocar um espaço de memória no *buffer-pool* para a nova página. Caso exista algum *slot* vago, ou seja, caso haja uma porção de memória suficiente para acomodar a página de disco, simplesmente escolhe-se um dos *slots*. Porém, isto ocorre raramente, somente no início da alocação do *buffer-pool*. Em geral, o *buffer-pool* está cheio, sendo necessário substituir uma das páginas armazenadas no *buffer-pool*. Em particular, a página de disco escolhida para ser substituída do *buffer-pool* deve ser armazenada em memória secundária, caso tenha sido modificada, ou simplesmente descartada, caso contrário.

Existem diversas políticas (ou algoritmos) de substituição de páginas. Tais políticas diferem basicamente com relação ao processo de escolha da página a ser removida do *buffer-pool* quando ocorre uma falha de página. Neste caso, a página referenciada não está armazenada na memória principal, sendo necessário trazê-la do disco. Políticas de substituição de páginas são amplamente usadas por sistemas gerenciadores de banco de dados, sistemas operacionais e sistemas de informações geográficas para o gerenciamento de dados em disco. Estruturas de indexação, em especial, utilizam tais políticas para gerenciar, algumas vezes, uma extensão de sua estrutura em memória principal, isto é, um *buffer-pool* dedicado. Este *buffer* aloca um espaço reduzido quando comparado ao usado pelo *buffer-pool* do sistema, que é voltado à manipulação de dados de diversas aplicações e/ou bancos de dados.

Dentre as políticas de substituição de páginas, a mais popular é a LRU (*least recently used*). Segundo esta política, páginas de disco que não foram referenciadas nos últimos pedidos provavelmente continuarão não sendo usadas por um longo tempo. Assim deve-se remover do *buffer-pool* a página que foi usada há mais tempo. A política FIFO, por sua vez, também é muito popular e consiste simplesmente de uma fila do tipo “primeiro a entrar,

primeiro a sair", ou seja, remove-se a página que está há mais tempo no *buffer*. Já a política "segunda chance" consiste de uma pequena alteração na política FIFO, de modo a evitar que páginas muito usadas sejam descartadas. Para isto, o algoritmo deve examinar o *bit R* da página mais antiga (*bit* de referência). Se ele for 0, a página além de antiga, não tem sido referenciada, de modo que ela pode ser removida. Se *R* for igual a 1, este *bit* é zerado e a página é colocada no fim da fila, como se esta tivesse acabado de ser alocada no *buffer-pool*. Neste caso, a busca por uma página recomeça a partir do novo início da fila. Outra política simples, mas com eficiência duvidosa, é a política de substituição aleatória, que escolhe ao acaso a página a ser removida do *buffer-pool* (sem qualquer critério).

A política NRU (*not recently used*) inspeciona o *bit R* (de referência, quando ocorre leitura ou escrita) e o *bit M* (setado quando a página é modificada) associados a cada página para decidir qual página remover do *buffer-pool*. Periodicamente, o *bit R* é modificado para 0, para todas as páginas, de modo a tornar possível a identificação de páginas que não tenham sido referenciadas recentemente. Assim, quatro classes distintas são definidas: (1) classe 0: páginas que não foram referenciadas nem modificadas; (2) classe 1: páginas não referenciadas, mas modificadas; (3) classe 2: páginas que foram referenciadas, mas não modificadas e (4) classe 3: páginas referenciadas e modificadas. Esta política remove aleatoriamente uma página da classe de menor número que tenha algum representante, isto é, a política preserva, primeiramente, as páginas que tenham sido referenciadas entre dois intervalos de tempo, e em segundo lugar, as páginas que tenham sido modificadas. A política LFU (*least frequently used*) mantém um contador de freqüência para cada página do *buffer-pool*. Inicialmente, todos os contadores estão com freqüência zero. Quando uma certa página é referenciada, a freqüência de todas as páginas sofre um deslocamento para a direita (*shift* binário), sendo que para a página referenciada adicionalmente ocorre a soma de um valor *v* cujo *bit* mais significativo é 1 e todos os demais *bits* são 0. Isto garante que as páginas que

não foram referenciadas tenham as suas respectivas freqüências diminuídas, enquanto que a página referenciada passa a ter a maior freqüência. Escolhe-se a página que possui a menor freqüência para ser removida do *buffer-pool* (empates são resolvidos ao acaso).

Para este trabalho, utilizamos a política de substituição de páginas LRU, que é a política mais comumente utilizada na implementação de *buffer-pool* para estruturas de indexação (CIFERRI, 2002) (GUTTMAN, 1984) (FOLKS, 1999) (OLIVEIRA, 2001).

O tamanho do *buffer-pool* dedicado de uma estrutura de indexação, em geral, não ultrapassa 1 MB de memória principal. Esta restrição deve-se ao fato de que em um ambiente multiprogramado e multiusuário, vários processos e usuários compartilham os recursos do sistema computacional, especialmente a memória principal (RAM). Para a memória principal, tal compartilhamento reduz a oferta de espaço. Nestes ambientes, o gerenciamento da memória principal deve ser feito de modo eficiente, sendo comum a necessidade de alocação de (mais) memória por parte dos processos. O próprio sistema gerenciador de banco de dados, por exemplo, requer a alocação de porções de memória principal para conjuntos de *buffers*. A alocação de vastas porções de memória principal para estes *buffers* não é apropriada, nem permitida, pois neste caso as páginas provavelmente participarão do espaço de memória virtual, o qual é armazenado parcialmente em memória secundária. Para o *buffer-pool* dedicado, deve-se garantir que este esteja completamente contido na memória principal, caso contrário, as vantagens geradas pelo uso do *buffer-pool* são anuladas. Tal garantia é feita usando-se porções específicas e pequenas da memória principal.

A tecnologia atual lida comumentemente com blocos (ou páginas de disco) entre 1 e 4KB e alguns sistemas usam 8KB, porém menos freqüentemente. Para este trabalho optamos por utilizar blocos de 4KB, compatibilizando com os testes de desempenho realizados por NAKANO (2005).

2.7 Considerações Finais

Aplicações que envolvem elevado volume de dados de *strings* como a pesquisa de similaridade em bancos de dados biológicos sofrem quando é necessário realizar pesquisas sobre estes dados. Uma alternativa é a utilização de estruturas de indexação para otimizar o processo de busca, evitando a varredura completa do banco de dados, reduzindo o custo de tempo para a realização da pesquisa.

O uso de um *buffer-pool* é capaz de reduzir o número de acessos à disco, no caso, quando a estrutura de indexação se encontra em memória secundária e uma página de disco a ser acessada novamente, já se encontra no *buffer-pool*.

Neste trabalho, a estrutura de indexação MRS foi adaptada para memória secundária com o auxílio de um *buffer-pool*. Para evitar a varredura completa dos dados pela ferramenta BLAST, é possível utilizar uma estrutura de indexação, como a MRS.

CAPÍTULO 3

ESTRUTURA DE INDEXAÇÃO MRS

Neste capítulo é descrita a estrutura de indexação MRS (*Multi Resolution String*), uma estrutura compacta projetada para indexar dados de seqüências em memória principal. Nesta estrutura, *substrings* dos dados são mapeadas em um espaço inteiro com o auxílio de coeficientes *wavelets*, os quais são posteriormente indexados utilizando MBRs (*Minimum Bounding Rectangles*). No processamento de *range queries*, responsáveis por retornar todas as *substrings* que se encontram dentro de um determinado raio de pesquisa (*range*), uma função de distância é utilizada para o cálculo da distância de edição entre *strings*. Na seção 3.1 é apresentado o contexto à qual a estrutura de indexação MRS é aplicada, assim como os conceitos básicos utilizados para a criação da estrutura de indexação. Na seção 3.2 é apresentada a estrutura de indexação MRS e seus métodos de pesquisa: *range query* e *nearest neighbour query*. E por fim, na seção 3.3 são apresentadas algumas considerações finais.

Este capítulo foi escrito com base em (KAHVECI, 2003) (NAKANO, 2005) (KUROSHU, 2005) como parte do projeto Bioinf BDB "Bioinformática e Banco de Dados Biológicos.

3.1 Conceitos Básicos

Nos últimos anos, os avanços tecnológicos na área da biologia molecular têm proporcionado um crescimento exponencial do volume de dados armazenados em bancos de dados biológicos. Um exemplo de banco de dados volumoso é o banco de dados de proteínas e seqüências do *National Center for Biotechnology Information* (NCBI), chamado de *GenBank*. Na Figura 3.1 é apresentado um gráfico do crescimento da base de dados do

GenBank de agosto de 2000 a agosto de 2005, quando a base de dados do *GenBank* ultrapassou 100.000.000.000 pares de base. Atualmente, o *GenBank* possui 145.739.069.776 pares de bases em 79.093.266 registros de seqüências (BENSON, 2006). Estatísticas demonstram que o tamanho do *GenBank* tem dobrado a cada 15 meses.

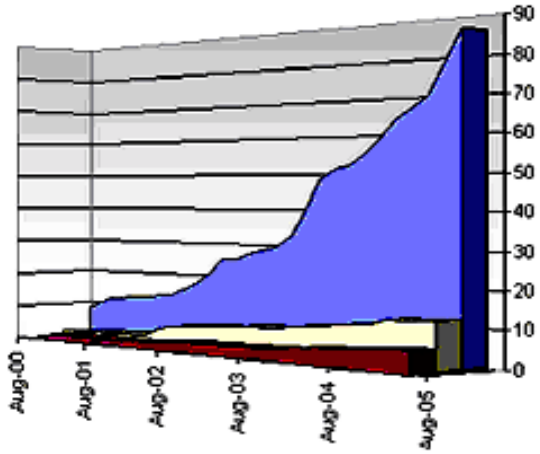


Figura 3.1: Crescimento do GenBank
Fonte: http://www.ncbi.nih.gov/Genbank/index.html

A construção de uma estrutura de indexação apropriada é capaz de acelerar o processo de pesquisa de similaridade. Uma estrutura de indexação pode ajudar a direcionar a pesquisa a uma porção do banco de dados biológico com elevado potencial de similaridade, evitando a varredura completa dos dados. Apesar de levar um tempo considerável para construir uma estrutura de indexação sobre um grande volume de dados de seqüências, torna-se ineficiente construí-las para cada pesquisa. As seqüências são relativamente estáveis e a maioria das atualizações refere-se a adições de novas seqüências. Assim, é possível amortizar o custo de construção da estrutura de indexação sobre diversas consultas mantendo a estrutura persistente (i.e., em memória secundária).

Algoritmos em memória principal podem se tornar impraticáveis à medida que o volume de dados biológicos cresce exponencialmente, devido ao alto custo das operações de E/S (entrada e saída de dados). Uma alternativa consiste da indexação de dados biológicos em

memória secundária para melhorar o desempenho das consultas, de modo a reduzir a porção do banco de dados a ser carregado em memória principal.

Estruturas de indexação baseadas em atributo estão emergindo como um importante paradigma em banco de dados de *string*. A técnica utilizada mapeia subseqüências para pontos em um espaço multidimensional e os indexa através de uma estrutura de dados multidimensional (Sun, 2004). Uma estrutura de indexação compacta em memória principal baseada nesta técnica é proposta em (Kahveci, 2001), intitulado de *Multi Resolution String* (MRS), na qual *substrings* (i.e., subseqüências no contexto biológico) de um banco de dados são mapeadas em um espaço multidimensional com o auxílio de coeficientes *wavelets*. A indexação desses coeficientes é efetuada por meio de MBRs (*Minimum Bound Rectangles*). Um MBR consiste de um retângulo *d*-dimensional com lados paralelos que envolvem, com o mínimo de volume, todos os objetos (i.e., pontos no espaço *d*-dimensional) nele contidos. É importante salientar que para cada MBR são armazenados apenas os dois pontos no espaço d-dimensional que delimitam o espaço do MBR e o ponto inicial da primeira *string* contida no MBR, reduzindo assim o requisito de armazenamento.

3.2 Funções de Distância

Dada uma seqüência s_1, esta pode ser transformada em outra seqüência s_2 através do uso de três operações de edição: inserção, remoção e substituição. A Figura 3.2 ilustra a transformação da seqüência GCACGTA para CTCCTT utilizando operações de edição. A distância entre duas *strings* pode ser definida pela *edit distance* (ED), uma métrica de similaridade entre *strings* popularmente utilizada, sendo calculada como o número mínimo de operações de edição para transformar s_1 em s_2. Seja m e n os comprimentos das seqüências s_1 e s_2, a ED(s_1,s_2) e as operações de edição correspondentes podem ser determinadas com

complexidade de tempo e espaço da ordem de $O(mn)$, utilizando programação dinâmica (Korf, 2003).

Figura 3.2 Transformação da *string* GCACGTA para CTCCTT utilizando operações de edição de inserção (I), remoção (R) e substituição (S).

O emparelhamento das seqüências de bases apresentado na Figura 3.2 representa um alinhamento entre seqüências s_1 e s_2. Para a obtenção de s_2 a partir de s_1 é necessário a: (i) remoção do caractere G, (ii) inserção do caractere T, (iii) substituição do caractere A por C, (iv) substituição do caractere G por T e (v) remoção do caractere A. Os traços correspondem a caracteres não alinhados. Para cada par de caracteres é atribuída uma pontuação baseada em sua similaridade. O valor de um alinhamento é definido pela soma das pontuações de todos os pares de caracteres. O alinhamento global de duas seqüências é definido pelo alinhamento de valor máximo entre as seqüências, definir o alinhamento global de duas seqüências é o mesmo que calcular a ED entre elas. Já o alinhamento local de duas seqüências é definido pelo alinhamento de valor máximo de todas as suas subseqüências.

O cálculo da ED é muito custoso, tanto em termos de complexidade de tempo e de espaço (Kahveci, 2001) (Sun, 2004). Desta forma, foram propostas aproximações visando reduzir o custo de cálculo da ED.

Inicialmente, é proposta uma função de distância baseada em vetores de freqüência. O vetor de freqüência visa mapear uma *string* para um ponto em um espaço multidimensional. Para uma determinada *string* s do alfabeto $\sum = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$, e considerando n_i o número de ocorrências do caractere α_i em s para $1 \leq i \leq \sigma$, o vetor de freqüência de s é definido por $f(s) = [n_1, n_2, ..., n_\sigma]$. No contexto de seqüências de DNA, o alfabeto básico é $\sum = \{A, C, G, T\}$, considerando uma seqüência t = GCACGTA, então $f(t) = [2, 2, 2, 1]$, utilizando a ordem alfabética na construção do vetor de freqüência.

É possível destacar que o vetor de freqüência gera um ponto no espaço σ-dimensional, independente do tamanho da *string* a ser mapeada, o somatório das entradas do vetor de freqüência é independente do conteúdo da *string*, além de que todas as entradas são não-negativas.

Uma operação de edição sobre uma *string s* pode ocasionar um dos seguintes efeitos sobre $f(s)$, considerando $1 \leq i, j \leq \sigma$, e $i \neq j$:

1. $n_i = n_i + 1$;
2. $n_i = n_i - 1$;
3. $n_i = n_i + 1$ e $n_j = n_j - 1$;

O efeito ocasionado pelo caso 1 representa a inserção de α_i em *s*, o caso 2 representa a remoção de α_i de *s* e o caso 3 representa a substituição de α_i por α_j em *s*. Uma única operação de edição em uma *string* resulta em uma mudança limitada em seu vetor de freqüência correspondente. Com base neste fato, é definida uma nova função de distância para os vetores de freqüência denominada de *frequency distance* (FD_1). Seja *u* um ponto (i.e., vetor de freqüência) e dois pontos de um MBR (denominado *box*) no espaço σ-dimensional, $FD_1(u, box)$ é calculada pelo número mínimo de passos para obter *box* a partir de *u*, utilizando uma única operação de edição a cada passo.

```
/* u é um ponto  inteiro no espaço σ-dimensional e box possui dois pontos
(lower e higher) nesse espaço. */
Algoritmo FD1(u,box)
   1) posDistance := negDistance := 0
   2) Para i:=1 até σ faça
        Se u[i] > box.lower[i]
             Então posDistance := posDistance + u[i] – box.lower[i]
             Senão Se u[i] > box.higher[i]
                  Então negDistance := negDistance + u[i]– box.higher[i]
   3) Se posDistance > negDistance
        Então retorna posDistance
        Senão retorna negDistance
```

Figura 3.3 Cálculo de FD_1
Fonte: Kahveci; Singh (2001).

Sejam duas *strings* s_1 e s_2, temos que a *frequency distance* entre os vetores de freqüência das duas seqüências é um limite inferior para a *edit distance* entre as duas seqüências, $FD_1(f(s_1), box) \leq ED(s_1, s_2)$. A Figura 3.3 apresenta um algoritmo para o cálculo da $FD_1(u, box)$ para o ponto *u* e um MBR *box*, constituído de dois pontos (inferior e superior), no espaço σ-dimensional, o algoritmo original em Kahveci (2001) calculava a *frequency distance* de um ponto *u* para outro ponto *v*, mas em se tratando de que a pesquisa é realizada de um ponto *u* para os MBRs contidos na estrutura de indexação MRS, a *frequency distance* foi reformulada.

Uma forma de aprimorar a *frequency distance* é armazenar as freqüências locais dos caracteres na *string*, assim como suas freqüências globais. Para isto, Kahveci e Singh (Kahveci, 2001) definem a transformação *wavelet* de uma *string*.

A transformação *wavelet* possui dois coeficientes (*A* e *B*). Considerando uma *string s* e seu vetor de freqüência *f*(*s*), a transformação *wavelet* de *s* pode ser definida formalmente por:

- $A_{k,i} = f(s[2^k i : 2^k(i + 1) - 1])$, sendo $f(s)$ o vetor de freqüência da *string s* para um dado intervalo. Cada coeficiente $A_{k,i}$ corresponde a um vetor de freqüência de uma *string* de tamanho 2^k.
- $B_{k,i} = f(s[2^k i : 2^k i + 2^{k-1} - 1]) - f(s[2^k i + 2^{k-1} : 2^k i + 2^k - 1])$, onde cada coeficiente $B_{k,i}$ corresponde à diferença de dois vetores de freqüência de duas *substrings* consecutivas de *s* de tamanho 2^{k-1}.

Os dois coeficientes *wavelet* que serão armazenados pela estrutura de indexação são $A\log_2 n,0$ (primeiro coeficiente *wavelet*) e $B\log_2 n,0$ (segundo coeficiente *wavelet*), estes a partir de agora denominados simplesmente de *A* e *B*. A transformação *wavelet* pode ser considerada um ponto no espaço 2σ-dimensional, sendo $\Psi(s) = [A,B]$ a transformação *wavelet* para uma dada *string s*. Um exemplo de transformação *wavelet* é Ψ(GCACTCGA) = [[2, 3, 2, 1], [0, 1, 0, -1]].

Os passos para se obter $\Psi(s_j)$ a partir de $\Psi(s_i)$, sendo s_i e s_j *strings*, podem ser definidos a partir dos efeitos gerados na transformação *wavelet* por uma operação de edição. Considerando uma *string s* pertencente ao alfabeto $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$, seja $\Psi(s) = [A, B]$ o primeiro e o segundo coeficientes *wavelet* de *s*, e $A = [a_1, ..., a_\sigma]$ e $B = [b_1, ..., b_\sigma]$. Uma operação de edição ocasiona um dos seguintes efeitos em *A* e *B* para $1 \leq i,j \leq \sigma$, e $i \neq j$:

1. $a_i := a_i + 1, a_j := a_j - 1, b_i := b_i + 1, b_j := b_j - 1$
2. $a_i := a_i + 1, a_j := a_j - 1, b_i := b_i - 1, b_j := b_j + 1$
3. $a_i := a_i \pm 1, b_i := b_i \pm 1$
4. $a_i := a_i \pm 1, b_i := b_i + 1, b_j := b_j - 2$
5. $a_i := a_i \pm 1, b_i := b_i - 1, b_j := b_j + 2$

A ED entre s_i e s_j é no mínimo o número de passos no menor caminho para se obter $\Psi(s_j)$ a partir de $\Psi(s_i)$. Deste modo, foi definida uma nova função de aproximação, $FD_2(\Psi(s),$ *box*), apresentada na Figura 3.4, para calcular o número de passos no menor caminho no

espaço 2σ-dimensional entre um ponto e um *box* neste espaço, baseado nos cinco efeitos ocasionados pelas operações de edição sobre os coeficientes *A* e *B*.

```
/* Ψ(s) possui um ponto inteiro (dividido em a e b) no
espaço 2σ-dimensional e box possui dois pontos inteiros
(divididos em lowA, lowB, high A e highB) nesse espaço,
sendo estes os pontos que delimitam o MBR. */
Algoritmo FD2(Ψ(s), box)
   1) pos := neg := 0
   2) Para i:=1 até σ faça
        Se a[i] < box.lowA[i]
             Então pos += (box.lowA[i] – a[i] )
             Senão Se a[i] > box.highA[i]
                  Então neg +=  (a[i] – box.highA[i])
        Se b[i] < box.lowB[i]
             Então pos += (box.lowB[i] – b[i])
             Senão Se b[i] > box.highB[i]
                  Então neg += (b[i] – box.highB[i])
   3) Se pos < neg
        Então min = pos
        Senão min = neg
   4) Se min < |pos-neg|/2
        Então retorna|pos-neg|/2
        Senão retorna |pos-neg|/2 + (min-|pos-neg|/2)/2
```

Figura 3.4 Cálculo de FD_2
Fonte: Kahveci; Singh (2001).

A função de distância $FD_2(\Psi(s), box)$ é calculada utilizando o primeiro e o segundo coeficiente *wavelet* ao mesmo tempo, definindo um limite inferior para a *edit distance*. Entretanto, $FD_1(u, box)$ não é necessariamente menor que $FD_2(\Psi(s), box)$, sendo definido a *frequency distance* máxima (FD) entre s_i e *box* como:

$$FD(s_i, box) = \max \{FD1(f(si), box), FD2(\Psi(s), box)\}$$

3.3 A Estrutura de Indexação MRS

Seja $S = \{s_1, s_2, ..., s_d\}$ um banco de dados consistindo de *strings* potencialmente longas pertencentes ao alfabeto $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$ e $w_1 = 2^a$ o tamanho da menor *string* de

consulta (*query string*) possível, sendo *a* um número inteiro. A estrutura de indexação armazena um conjunto de árvores $T_{i,j}$, onde *i* varia de *a* até $a + l - 1$, e *j* varia de 1 até *d*. O parâmetro *l* corresponde ao número de resoluções disponíveis na estrutura de indexação, onde cada resolução corresponde a um tamanho 2^i utilizado para o mapeamento da *string* em pontos multidimensionais. A árvore $T_{i,j}$ é a estrutura de indexação para a j-ésima *string* correspondendo a um nível de resolução 2^i.

A árvore $T_{i,j}$ é obtida deslizando uma janela de cobertura de tamanho 2^i sobre a *string* s_j. Para cada possível posicionamento da janela, é computada a transformação *wavelet* da *substring* correspondente de s_j, de forma que cada *substring* é mapeada na estrutura de indexação como um ponto no espaço 2σ-dimensional. Para a *substring* inicial é criado um MBR englobando os coeficientes *wavelet* da *substring*. Este MBR é então estendido para englobar as transformações das próximas *c*-1 *substrings*, onde *c* corresponde à capacidade do MBR. Após a transformação das primeiras *c substrings*, um novo MBR é criado para englobar as próximas *c substrings*. Esse processo se repete até o final da *string* s_j. Para cada MBR são armazenados apenas os dois pontos que o delimitam junto com a posição inicial da primeira *substring* contida no MBR.

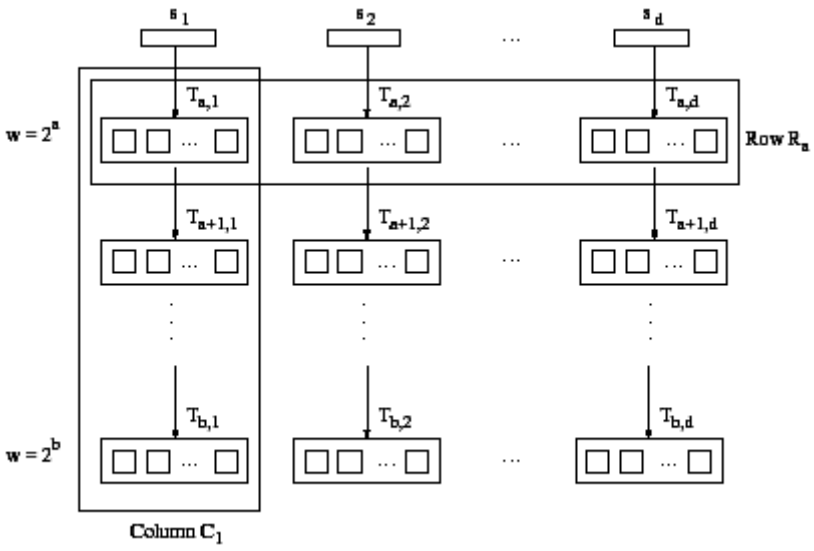


Figura 3.5: *Layout* da Estrutura de Indexação MRS
Fonte: Kahveci; Singh (2001).

A estrutura de indexação é composta por linhas e colunas, sendo que a *i*-ésima linha, representada por R_i, é o conjunto $R_i = \{T_{i,1}, ..., T_{i,d}\}$ correspondendo ao conjunto de todas as árvores de resolução 2^i. Da mesma forma, a *j*-ésima coluna, representada por C_j, é o conjunto $C_j = \{T_{a,j}, ..., T_{a+l-1,j}\}$ correspondendo ao conjunto de todas as árvores para a *j*-ésima *string*. O *layout* da estrutura de indexação MRS é apresentado na Figura 3.5.

Range Query

Uma *range query* pesquisa por todas as *substrings* do banco de dados que estejam a, no máximo, uma *edit distance r* de uma dada *string* de consulta *q*, onde *r* é o raio da pesquisa. A taxa de erro é definida por:

$$\varepsilon = \frac{r}{|q|}$$

Em um primeiro passo, a técnica de pesquisa particiona uma dada *string* de consulta de tamanho arbitrário em um número de subconsultas de várias resoluções disponíveis na estrutura de indexação. Então é realizada uma *partial range query* para cada uma destas subconsultas em suas linhas correspondentes na estrutura de indexação. Esse passo é denominado *partial range query* por calcular apenas a distância da transformação *wavelet* da *substring* de consulta aos MBRs, ao invés da distância da *string* de consulta às *substrings* contidas nos MBRs.

Dada uma *string* de consulta *q* de tamanho $k*2^a$, a técnica de particionamento escolhe o mais longo sufixo possível de *q*, de modo que seu tamanho seja igual a uma das resoluções disponíveis na estrutura de indexação, sendo a última *substring* de consulta. O restante da *string* é particionada recursivamente compondo um particionamento único, $q = q_1q_2...q_t$, com $|q_i| = 2^{c_i}$ e $a \leq c_1 < ... < c_{i+1} \leq ...c_t \leq a + l - 1$.

A pesquisa é realizada com base na taxa de erro ε passada como parâmetro e inicia-se realizando uma pesquisa de q_1 na linha R_{c1} da estrutura de indexação. Um conjunto de MBRs cujas distâncias (i.e., distância fornecida pela função FD que calcula a distância entre a transformação *wavelet* da *string* de consulta a um MBR) são inferiores à uma distância $r = \varepsilon \times |q|$ de q_1 é obtido. A cada iteração do algoritmo, o valor de r é refinado para cada MBR do conjunto resultante. A pesquisa prossegue para as subconsultas restantes em suas respectivas linhas da estrutura de indexação, reiniciando o valor r para cada subconsulta e o refinando a cada iteração subseqüente.

```
/* r = ε*|q| */
Algoritmo RANGE_SEARCH(q,r)
   1) Particiona a query string q em t partes como q1,q2,...,qt
   sendo |qi| = 2^ci e a ≤ c1 < ... < ci ≤ ci+1 ≤ ... ≤ ct ≤ a + l – 1. Dado que
   Tc1,j contém Mc1 MBRs.
   2) Para j:=1 até d faça /*processa cada string do BD */
     3) Para k:=1 até Mc1 faça /*processa cada MBR*/
        a) distance[k] := r
        b) Para i:=1 até t faça
        Se existe o MBR Bk na resolução de qi Então
             i) Se FD(qi, Bk) < distance[k]
                  Então insere Bk em Resci,j
             ii) distance[k] := max Bk∈Resci,j {distance[k] – FD(qi,B)}
   4) Ler páginas de disco correspondentes a Resct,j
   5) Executar pós-processamento para eliminar retornos falsos
```

Figura 3.6: Algoritmo de pesquisa *Range Query*
Fonte: Kahveci; Singh (2001).

O algoritmo para a pesquisa é apresentado na

Figura 3.6. No passo 1 é descrita a rotina de particionamento da *string* de consulta. Nos passos 2 e 3, todos os MBRs de todas as *strings* do banco de dados são pesquisados independentemente. Em 3*a* o vetor de distâncias é inicializado e em 3*b* as linhas da estrutura de indexação correspondentes às linhas das subconsultas q_i são pesquisadas para todas as *strings* e MBRs dos passos 2 e 3, armazenando em $Res_{ci,j}$ os MBRs cujas distâncias para q_i sejam inferiores que r e refinando seu valor a cada iteração. Após todas as linhas terem sido

pesquisadas, as páginas de disco correspondendo ao último conjunto resultante são lidas (Passo 4). E por último é realizado um pós-processamento para eliminar falsos candidatos obtidos na pesquisa (Passo 5).

Nearest Neighbour Query

A pesquisa *nearest neighbour query* pesquisa pelas *k substrings* do banco de dados que estejam mais próximas da *string* de consulta *s*.

Algoritmo NN_SEARCH(q, k)

1) Fase 1
   - B := O conjunto dos *k* mais próximos MBRs à consulta *q*.
   - r_1 := k^{th} menor *edit distance* para *strings* em B.

2) Fase 2
   - *RANGE_SEARCH(q, r_1)*
   - Retorne as *k strings* mais próximas no conjunto resposta.

Figura 3.7: Algoritmo de pesquisa *K-nearest Neighbour*
Fonte: Kahveci; Singh (2001).

O algoritmo de pesquisa *nearest neighbour query*, apresentado na Figura 3.7, é realizado em duas etapas. Na primeira etapa, os *k* mais próximos MBRs da estrutura de indexação são determinados por uma pesquisa na estrutura de indexação. Após a determinação dos *k* MBRs mais próximos, o algoritmo efetua a leitura das *substrings* contidas nesses MBRs, e encontra as *k* menores *edit distance* dessas *substrings* para a *string* de consulta. Esta distância é representada por r_1. No geral, apenas uma pequena porcentagem do banco de dados é processada nesta etapa. Na segunda etapa, é realizada uma *range query* utilizando r_1 como sendo o raio da pesquisa.

3.4 Considerações Finais

A estrutura de indexação MRS é uma estrutura compacta, apropriada para o uso em memória principal. Baseado no fato de que o crescimento exponencial dos dados biológicos é superior à disponibilidade de memória principal, podendo ocasionar a falta de memória principal para o armazenamento da estrutura de indexação MRS, assim necessitando o uso da memória secundária, foi realizada a implementação da estrutura de indexação MRS para a realização de testes de desempenho com um grande volume de dados. A implementação foi realizada de forma que a estrutura de indexação seja baseada em memória secundária com o auxílio de um *buffer-pool*. A implementação é descrita no próximo capítulo.

CAPÍTULO 5

TRABALHOS CORRELATOS

Este capítulo tem como objetivo descrever os principais trabalhos existentes atualmente na literatura, voltados à indexação de bancos de dados biológicos. Na seção 4.1 e 4.2 são apresentadas propostas para melhoramento da estrutura de indexação MRS, enquanto que na seção 4.3 e 4.4, são apresentadas propostas baseadas em árvores de sufixo, que é uma estrutura alternativa para indexação de *strings*.

4.1 Kahveci & Singh

Kahveci (2002, 2003) propõe uma técnica que integra a estrutura de indexação MRS e uma ferramenta de pesquisa de *substring* residente em memória, como por exemplo o BLAST. As associações encontradas entre a *string* de consulta e os MBRs presentes na estrutura de indexação são usadas para reduzir os pares de *substrings* encontrados no banco de dados, que não possuem regiões de similaridade, reduzindo significativamente o custo de E/S e processamento. Em Kahveci (2002), é proposta uma nova *frequency distance* para melhorar o desempenho da estrutura de indexação MRS.

Frequency Distance

Considere $S = \{f(s) \mid s \in \Sigma^*\}$ o conjunto de todos os possíveis vetores de freqüência das *strings* pertencentes ao alfabeto Σ, onde $|\Sigma| = \sigma$. Seja o conjunto S posicionado em um quadrante não-negativo de um espaço inteiro σ-dimensional. É definido um subconjunto S_w =

$\{f(s) \mid s \in \Sigma^*$ e $|s| = w\}$, onde w é um inteiro positivo. O conjunto S_w define todos os vetores de freqüência possíveis de *strings* de tamanho w.

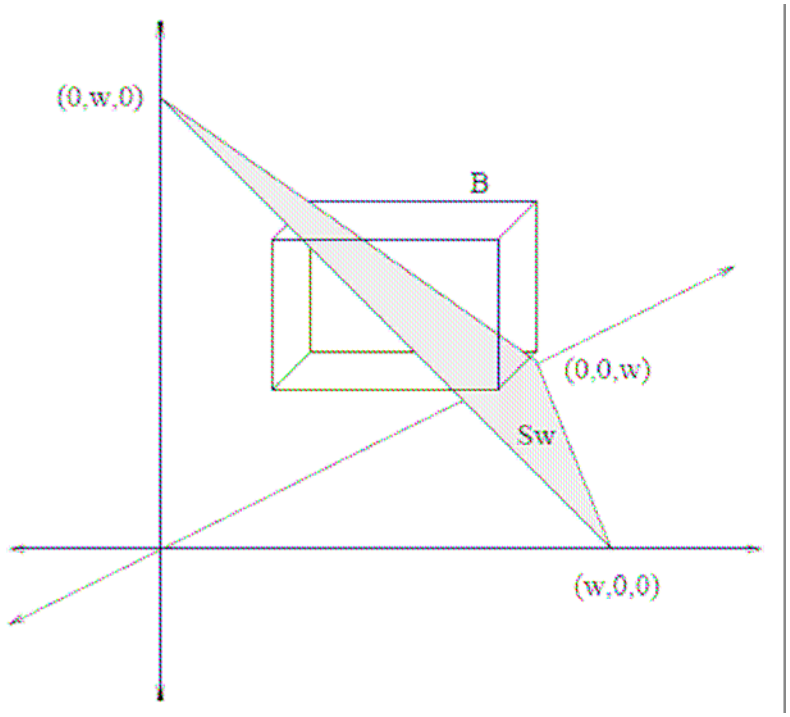


Figura 4.1: O espaço de vetor de freqüência *S*, um subespaço *Sw*, e um MBR *B* em *S* quando $|\Sigma| = 3$.

Considerando B um MBR de uma estrutura de indexação MRS para a resolução w. Seja q uma *string*, e s uma *string* de tamanho w de modo que $f(s) \in B$, então:

1. $FD(f(q), B) \leq FD(f(q), B \cap S_w)$
2. $FD(f(q), B \cap S_w) \leq EditDistance(q, s)$

Ou seja, $FD(f(q), B \cap S_w)$ é um limite inferior melhor que $FD(f(q), B)$, reduzindo potencialmente a pesquisa em uma grande porção do banco de dados. A Figura 4.2 apresenta o algoritmo para o cálculo da nova *frequency distance*.

/* v é um ponto inteiro σ-dimensional */
/* B é um *box* inteiro σ-dimensional de coordenadas inferior e superior $B.L$ e $B.H$ */
Algoritmo $FD_w(v, B)$

1. $inc := dec := sum := 0$
2. Para $i := 1$ até σ
   - Se $v[i] < B.L[i]$ então
     $inc \mathrel{+}= B.L[i] - v[i]$
     $sum \mathrel{+}= B.L[i]$
   - Senão se $B.H[i] < v[i]$ então
     $dec \mathrel{+}= v[i] - B.H[i]$
     $sum \mathrel{+}= B.H[i]$
   - Senão
     $sum \mathrel{+}= v[i]$
3. Se $sum < w$ então
   $dec \mathrel{+}= w - sum$
4. Senão se $w < sum$ então
   $inc \mathrel{+}= sum - w$
5. Retorne $max\{inc, dec\}$

Figura 4.2 – Algoritmo da nova *frequency distance*.

Extensão para Pontuação e Affine Gaps

A técnica também é estendida para considerar pontuações positivas para pares de caracteres iguais e uma penalidade caso sejam diferentes. Além de considerar o modelo de *affine gaps*, que atribui uma penalidade maior para a abertura de um *gap*, e uma outra penalidade para cada unidade de *gap*. O algoritmo para o cálculo da *frequency distance*, considerando *gaps* é apresentado na Figura 4.3.

/* v é um ponto inteiro σ-dimensional */
/* B é um *box* inteiro σ-dimensional de coordenadas inferior e superior $B.L$ e $B.H$ */
Algoritmo $FS_w(v, B)$

1. $inc := dec := sum := 0$
2. Para $i := 1$ até σ
   - Se $v[i] < B.L[i]$ então
     $inc \mathrel{+}= B.L[i] - v[i]$
     $sum \mathrel{+}= B.L[i]$
   - Senão se $B.H[i] < v[i]$ então
     $dec \mathrel{+}= v[i] - B.H[i]$
     $sum \mathrel{+}= B.H[i]$
   - Senão
     $sum \mathrel{+}= v[i]$
3. $ScoreInc = (min\{sum, w\} - inc) \times S_{match} + inc \times S_{mismatch}$
4. $ScoreDec = (min\{sum, w\} - dec) \times S_{match} + dec \times S_{mismatch}$
5. Se $sum < w$ então
   $ScoreInc \mathrel{+}= S_{gapopen} \times (sum - w - 1) + S_{gap\_extend}$
6. Senão se $w < sum$ então
   $ScoreDec \mathrel{+}= S_{gapopen} \times (w - sum - 1) + S_{gap\_extend}$
7. Retorne $min\{ScoreInc, ScoreDec\}$

Figura 4.3 – Algoritmo $FS_w(v, B)$.
Fonte: Kahveci (2002).

Pontuações Específicas de Alfabeto

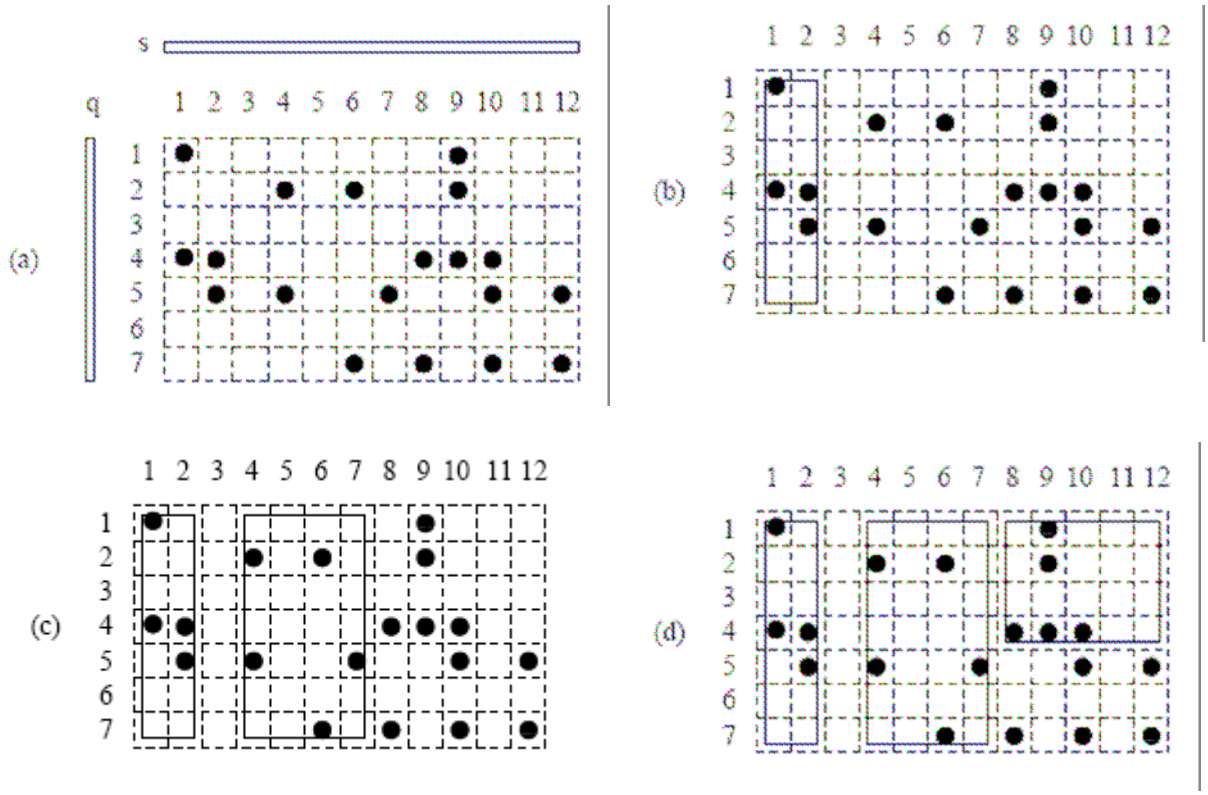
Em se tratando de pesquisa de similaridade em seqüências de proteínas, há a necessidade do uso de uma matriz de substituição para definir as pontuações dos alinhamentos. É possível estender o algoritmo apresentado na Figura 4.3 para encontrar um limite superior para a melhor pontuação. Para isto, após a quantidade de incrementos e decrementos para cada caractere ser determinado (Passo 2), o algoritmo interativamente encontra um caractere crescente e um decrescente que maximizem a pontuação.

Algoritmo MAP (*MAtch table based Pruning*)

O algoritmo MAP constrói uma tabela de emparelhamento (*Match Table*) booleana para uma *string* de consulta e uma *string* do banco de dados com o auxílio da estrutura de

indexação MRS. Cada entrada na *Match Table*, que corresponde a um par de *substring* consulta/banco de dados, é marcada como verdadeira se a correspondente *substring* de consulta e *substring* do banco de dados potencialmente contém padrões similares. Caso contrário, é marcada como falsa.

Após a computação da *Match Table*, tabelas *hash* sobre estas *strings* são construídas. Primeiramente, é encontrado o número de linhas e colunas marcadas através da projeção de todas as entradas verdadeiras da *Match Table*. Caso o número de colunas marcadas seja superior ao número de linhas marcadas, é escolhida uma fatia (*slice*) vertical da *Match Table* para a construção da tabela *hash* correspondente às linhas da fatia escolhida. Caso contrário, é escolhido uma fatia horizontal para a construção da tabela *hash* (Figura 4.4). O tamanho das fatias é restrito à quantidade de memória principal disponível, de forma que o tamanho das tabelas *hash* não serão maior que a quantidade de memória principal disponível.



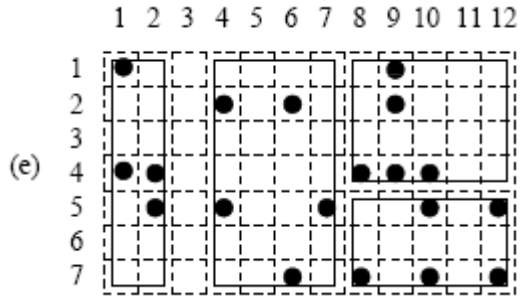


Figura 4.4 – Seleção de fatias (*slices*) para a construção de tabelas *hash*.
Fonte: Kahveci (2003).

Terminada a construção da tabela *hash* para uma determinada fatia da *Match Table*, as *substrings* marcadas da outra *string* são lidas seqüencialmente e alinhamentos exatos das *substrings* (sementes) de um tamanho pré-especificado (tamanho das *words*) são encontrados usando a tabela *hash*. Em seguida, as sementes são estendidas em ambas as direções para encontrar alinhamentos melhores. Ao final, os resultados são reportados em ordem decrescente de pontuação.

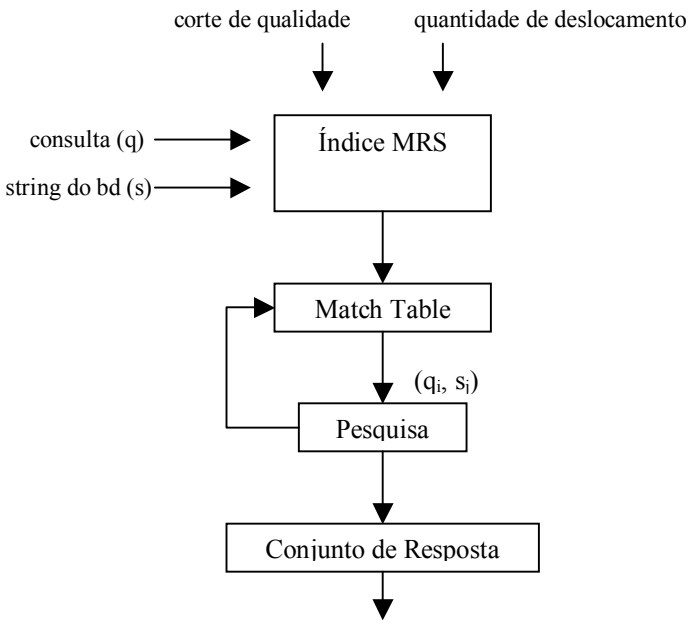


Figura 4.5 – Aplicação do MAP em conjunto com o BLAST.
Fonte: Kahveci (2003).

A Figura 4.5 apresenta o uso do algoritmo MAP em conjunto com o BLAST. O algoritmo MAP substitui a fase de *semeadura* do BLAST, que necessita a varredura completa do banco de dados. Com o uso deste novo algoritmo, a fase de *semeadura* é realizada sobre a estrutura de indexação MRS.

Resultados e Conclusões

Um exemplo de uso da estrutura de indexação MRS é apresentado em Kahveci (2003), além de ser apresentada uma nova *frequency distance*, que pode ser utilizada para melhorar o desempenho da pesquisa na estrutura de indexação MRS, reduzindo o espaço de busca da pesquisa no banco de dados.

Os testes de Kahveci (Kahveci, 2003) foram realizados com os cromossomos humanos 18 e 21, e também com o cromossomo 18 do rato, E.coli.K12 e E.coli.0157H7.EDL933 extraídos do NCBI. O algoritmo MAP foi comparado ao algoritmo da ferramenta BLAST, avaliando a qualidade das respostas obtidas e o desempenho dos algoritmos.

Durante os testes de qualidade das respostas obtidas, o algoritmo MAP obteve resultados similares ao do BLAST. O desvio entre as saídas do BLAST e as saídas do MAP foram de aproximadamente 5% para saídas com alta pontuação, e de 10 a 30% para saídas com baixa pontuação, assumindo um erro de 0.1. Para um erro de 0.25, o desvio é quase zero para saídas com alta pontuação, e 5% para saídas com baixa pontuação.

Nos testes de desempenho, o algoritmo MAP foi de 20 a 74 vezes mais rápido que o BLAST para a comparação dos cromossomos humanos 18 e 21 para tamanhos de memória 100, 200, 400, 800, 1600, 3200 e 6400 páginas.

O algoritmo MAP demonstrou ser um algoritmo eficiente com alto desempenho, retornando saídas de alta qualidade, próximas às saídas obtidas pelo BLAST. Além de executar bem com pequenas quantidades de memória principal.

4.2 Sun

Em (SUN, 2003), é proposta uma nova estrutura de indexação para pesquisa de similaridade de seqüências em bancos de dados de seqüências de DNA. Em relação à estrutura de indexação MRS, esta nova estrutura, denominada *CoMRI* (*Compressed Multi-Resolution Index*), substitui os conhecidos MBRs por VBRs (*Virtual Bounding Rectangles*), proporcionando uma redução considerável no custo de armazenamento.

Os MBRs são utilizados para representar um grupo de entradas no banco de dados, não necessitando armazenar todo e qualquer ponto de dado (SUM, 2003). Para cada MBR, basta apenas armazenar as coordenadas mínima e máxima, conseqüentemente o tamanho de um MBR em memória é a soma dos tamanhos dos dois vetores. Uma representação mais compacta é obtida através de VBRs, que são uma representação compacta baseada em quantização.

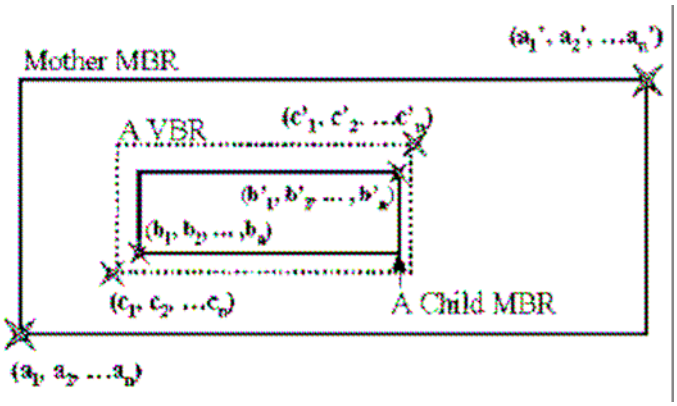


Figura 4.6 – Representação de Mother MBR, MBR e VBR.

Um VBR é criado através da representação de seu *box* em referência ao menor *box* contendo todos os pontos, denominado *Mother MBR*. Conforme visto na Figura 4.6, a

representação do *box* é mais compacta quando referenciado o *Mother MBR* do que todo o espaço.

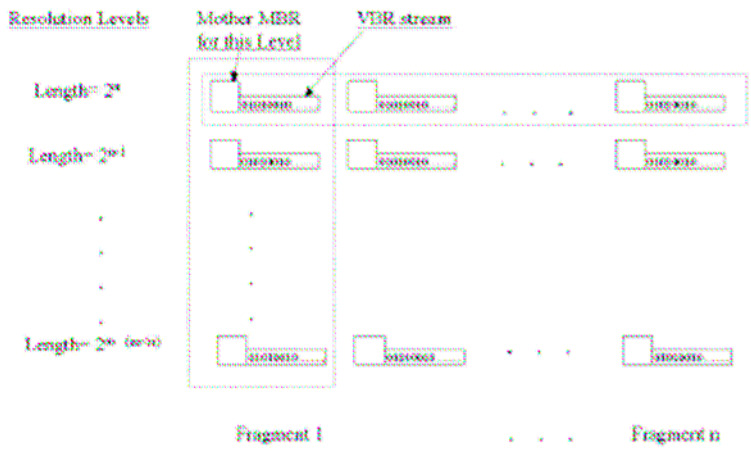


Figura 4.7: Visão Geral da Estrutura CoMRI

Uma visão geral da estrutura de indexação CoMRI é apresentada na Figura 4.7. Similar à estrutura de indexação MRS, cada coluna representa um fragmento do conjunto de dados e cada linha corresponde ao índice de uma determinada resolução. O diferencial se encontra no fato de que cada resolução contém um *Mother MBR* representado por seus pontos extremos e um VBR *stream*, representado como dois pontos quantizados com referência ao *Mother MBR*.

Resultados e Conclusões

Os VBRs propostos por Sun (2003) podem ser utilizados para substituir os MBRs que compõem a estrutura de indexação MRS, possivelmente otimizando o espaço ocupado pela estrutura de indexação.

Nos experimentos realizados por Sun (SUN, 2003) com os cromossomos humanos 18 e 22, mediu-se o desempenho da estrutura conforme a variação de parâmetros da estrutura, tais como: efeito do VBR no tamanho da estrutura de indexação, efeito da capacidade do

VBR, entre outros. Também foi realizado um teste comparativo entre a estrutura de indexação MRS e a CoMRI utilizando consultas retiradas aleatoriamente do cromossomo 12 de tamanhos entre 500 e 2000. Conforme demonstrado na Figura 5.7, a estrutura CoMRI executa 100 vezes mais rápido que a estrutura MRS, até mesmo para taxas de erro grandes.

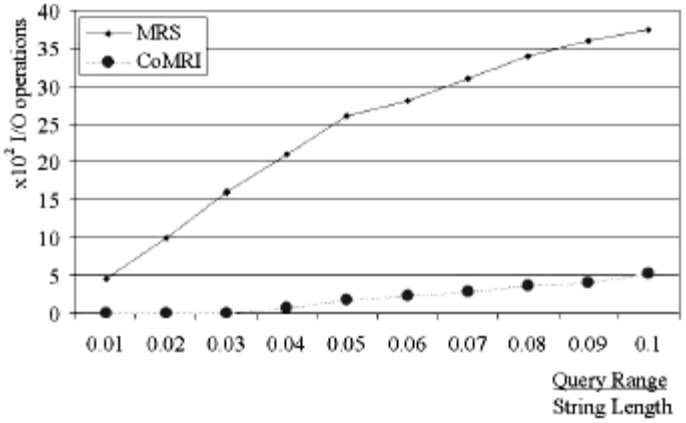


Figura 4.8 – Comparação de E/S para CoMRI e MRS.
Fonte: Sun (2003).

As vantagens apresentadas pela estrutura CoMRI se devem por motivos de otimizações aplicadas para a execução da *range query*, além de uma redução considerável no tamanho da estrutura de indexação devido ao uso de VBRs.

4.3 Hunt

Hunt (2000) (2002) explora o potencial das árvores de sufixo (*suffix trees*) no contexto de comparação aproximada de dados de *strings* não estruturados. Uma árvore de sufixo é um índice para todo os sufixos de uma determinada *string* (HUNT, 2000). Um exemplo de árvore de sufixo para a *string* ACATCTTA, com um caractere de término $ ao final, é apresentado na Figura 4.9. Uma árvore de sufixo indexando uma *string* de tamanho *n* possui *n* nós, um para cada sufixo.

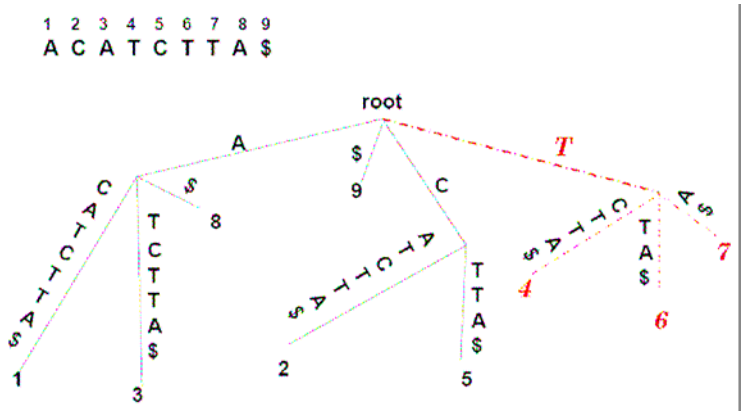


Figura 4.9 – Árvore de Sufixo para a *string* ACATCTTA.
Fonte: Hunt (2000).

A maioria das implementações de árvores de sufixo também utiliza o conceito de *suffix link*. Para cada nó interno, é criado um *suffix link* apontando a partir de um nó de indexação da árvore *aw* para um nó de indexação *w*, sendo que *aw* e *w* são traçados a partir da raiz e *a* é de tamanho 1. Um exemplo de árvore de sufixo com *suffix link* é apresentado na Figura 4.9.

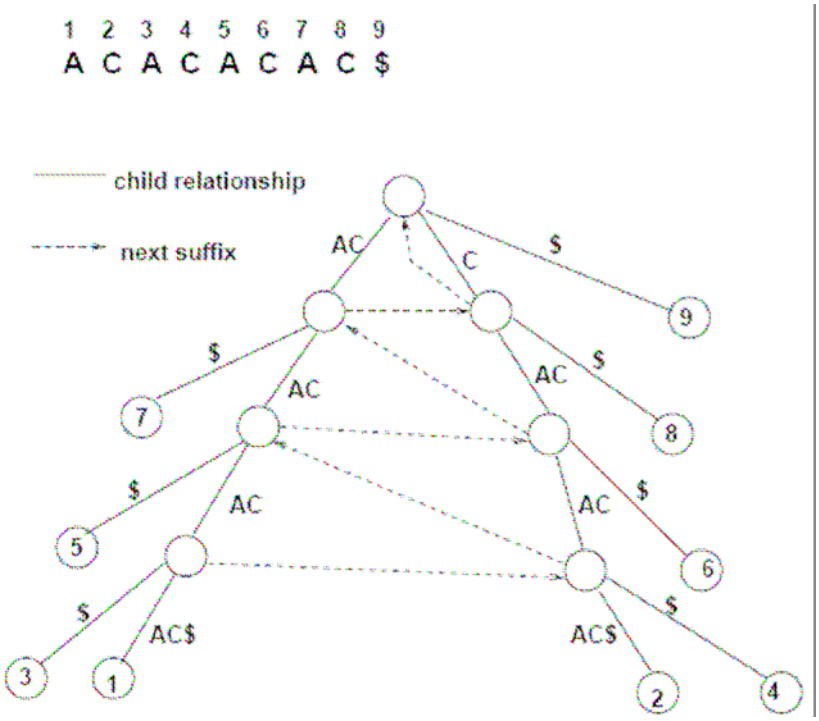


Figura 4.10 – Árvore de Sufixo e *links* em ACACACAC$
Fonte: Hunt (2002).

Em (HUNT, 2003), é proposta uma nova estrutura baseada em árvore de sufixo denominada *Suffix Sequoia*. Esta estrutura é derivada de uma árvore de sufixo, truncada em

uma determinada profundidade *d* com o auxílio de uma janela de tamanho *d* deslizada sobre o texto, e armazenada em memória como um mapa de *bit* e um vetor de posições de texto (Figura 4.11). O mapa de *bit* é armazenado em memória principal, representando a presença ou ausência de uma determinada *substring* curta em um texto. Já o vetor de posições é armazenado em memória secundária, podendo ser visto como um vetor de listas de índices de início de cada janela de texto.

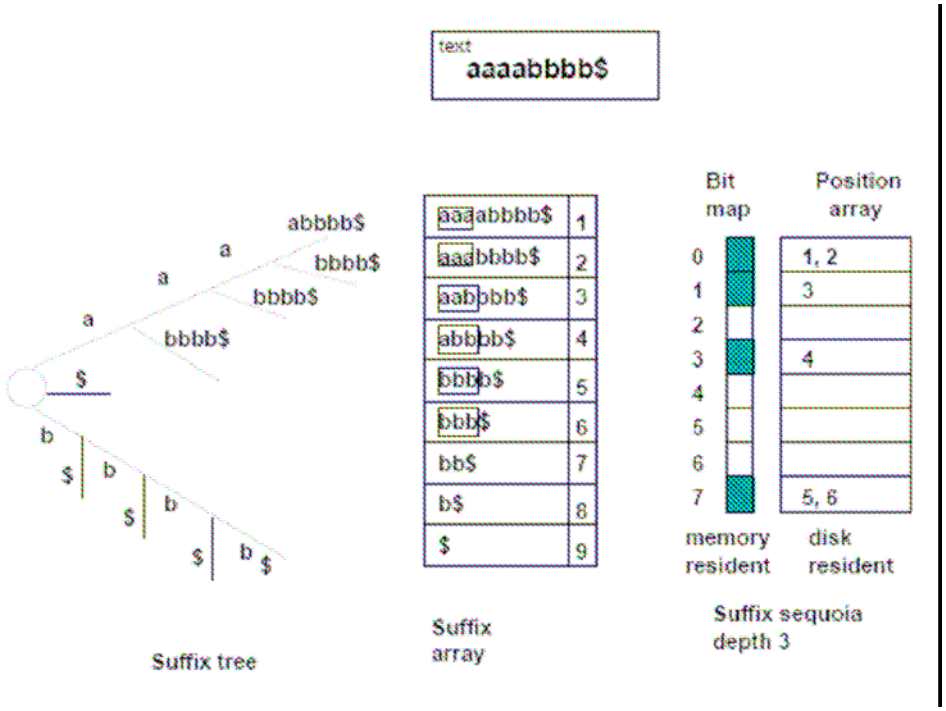


Figura 4.11 – Relacionamento entre árvore de sufixo, vetor de sufixo e *suffix sequoia*. Fonte: Hunt (2003).

Resultados e Conclusões

As árvores de sufixo representam uma alternativa para a indexação de dados de *string*, mas apresentam problemas relacionados ao tamanho da estrutura de indexação obtida que, em alguns casos, é superior ao próprio tamanho do banco de dados, além de que apresentam um bom desempenho para pesquisas exatas de *strings*, não ocorrendo o mesmo para pesquisas inexatas (i.e., pesquisa *range query*).

Os testes realizados por Hunt (2002) com seqüências de DNA dos cromossomos humanos 1, 21 e 22, *C.elegans*, e com seqüências de proteínas da *SWISS-PROT* demonstraram que a árvore de sufixo sem *suffix link* apresenta um grande potencial para a construção eficiente de grandes índices. A utilização de *suffix link* para a obtenção de um algoritmo de construção $O(n)$ é valido, mas requer um espaço adicional, além de gerar dificuldades para o carregamento em memória, com atualizações e leituras espalhadas.

Já os testes realizados na *suffix sequoia* (HUNT, 2003), que combina as características da árvore de sufixo com a eficiência com relação a espaço de um vetor, o índice é criado eficientemente e o tamanho é relacionado indiretamente ao tamanho dos dados armazenados. Os testes realizados com consultas contendo uma grande quantidade de repetições ou "palavras" muito similares, apresentaram muitos acessos a disco, resultado indesejado no uso de estruturas de indexação.

4.4 Neelapala

Neelapala (2004) apresenta uma estrutura de indexação baseada em uma compactação horizontal de uma *trie* denominada de SPINE (*String Processing INdexing Engine*). A estrutura de indexação SPINE consiste de um *backbone* formado por uma cadeia linear de nós representando a *string*, com os nós conectados por um conjunto de cortes para facilitar a passagem transversal sobre o *backbone* durante a fase de construção do índice e pesquisa. Todos os cortes recebem um rótulo durante o processo de construção com o intuito de prevenir falsos positivos quando for percorrido o índice durante a pesquisa.

Por ser baseada em árvores de sufixo, a estrutura de indexação SPINE possui, em um nível estrutural, as mesmas funcionalidades de uma árvore de sufixo, além de outras características como, por exemplo, o número de nós ser sempre igual ao tamanho da *string*. Após a construção do índice não é necessário acessar os dados, pois cada vértebra

corresponde a um caractere da *string*. Assim, a estrutura pode ser criada de maneira *online*, não sendo necessário um conhecimento prévio dos dados. Ademais, o fato da ordem de criação dos nós ser a mesma ordem lógica, a estrutura é *prefix-partitionable*, dado um prefixo da *string*, este prefixo corresponde ao fragmento inicial da estrutura.

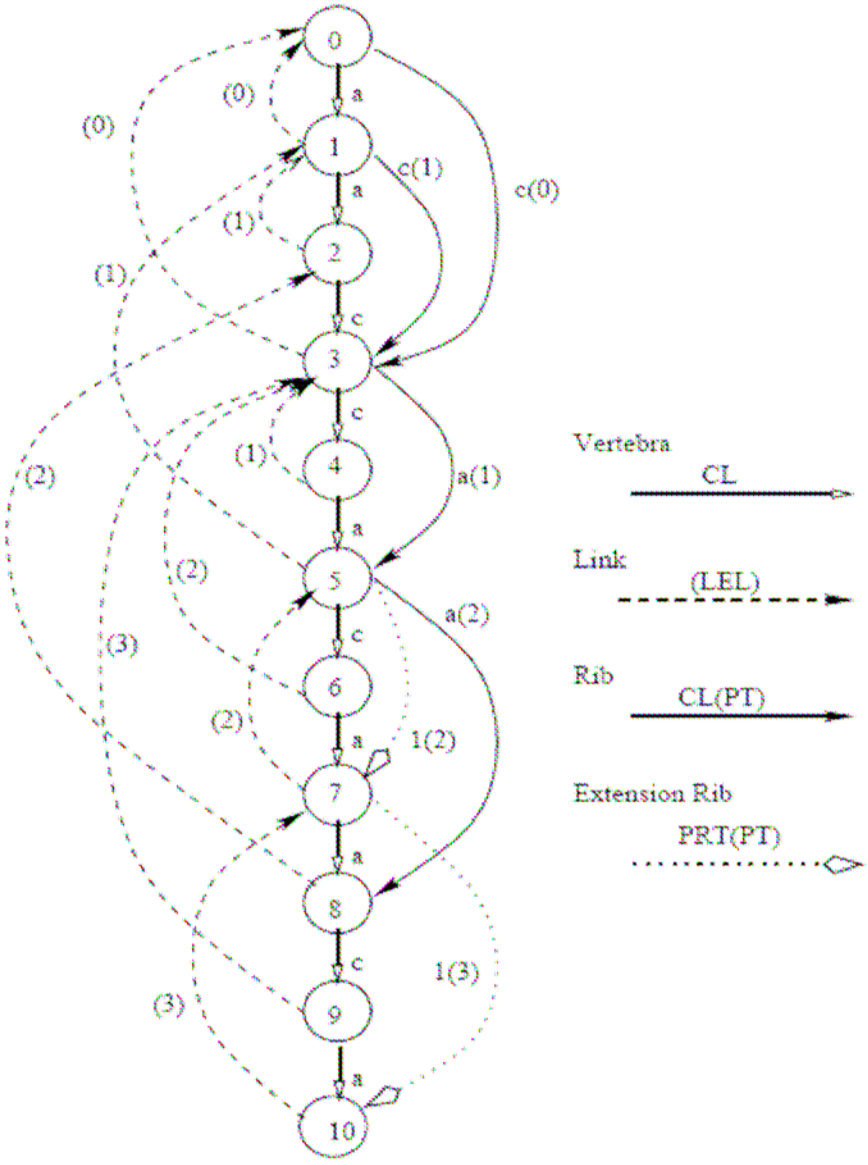


Figura 4.12 – Estrutura de Indexação SPINE (para *aaccacaaca*).
Fonte: Neelapala (2004).

A estrutura de indexação SPINE (Figura 4.12) é composta por:

1. Um *backbone* – componente central da estrutura – de nós conectados, denominados vértebras.

2. Um rótulo de caractere (*character label - CL*) associado a cada vértebra, com o valor do caractere da *string* a qual a vértebra corresponde.
3. *Links* registram, a cada nó, as informações sobre os sufixos *early-terminating* do nó.
4. *Ribs* são adicionados à estrutura representando a extensão de todos os sufixos ocasionados pela adição de um novo caractere da *string*.
5. *ExtRibs* são adicionados caso o nó corrente já possua um *rib* associado.

Resultados e Conclusões

A estrutura CoMRI é uma estrutura de indexação baseada em árvores de sufixo que busca melhorar o desempenho das árvores de indexação com a redução do tamanho ocupado pelo índice, reduzindo em 30% o espaço ocupado em memória secundária.

Em (NEELAPALA, 2004), testes são realizados com os cromossomos humanos 19 e 21, E.Coli e C.Elegans, para avaliar o desempenho da estrutura CoMRI. Para traçar um comparativo, os testes também são realizados com a árvore de sufixo, implementação retirada da ferramenta MUMmer. As métricas utilizadas para medir o desempenho foram: tempo para construção da estrutura de indexação e tempo de pesquisa no índice.

Na avaliação das estruturas em um ambiente de memória principal, a estrutura SPINE demonstrou ocupar em torno de 30% de espaço a menos que a árvore de sufixo, possibilitando a construção de um índice 30% maior que o permitido pelas árvores de sufixo, supondo que a árvore de sufixo tenha alcançado o limite da memória principal disponível. Quanto ao tempo de pesquisa no índice, SPINE gastou 30% a menos de tempo que as pesquisas realizadas na árvore de sufixo, fato este justificado pelo SPINE lidar com um número menor de sufixos.

Os testes de desempenho em memória secundária demonstraram um ganho no tempo de construção da estrutura SPINE de 50% em relação ao gasto pela árvore de sufixo. Por possuir nós de tamanho inferior, 30% do ganho de tempo diz respeito a este fato, os outros 20% correspondem a melhor localidade exibida pela estrutura SPINE. Já o tempo de pesquisa apresentado por SPINE é superior por um fator de 2 em relação às árvores de sufixo.

4.5 Considerações Finais

As estruturas de indexação baseadas em árvores de sufixo apresentam um melhor desempenho para pesquisas de *strings* exatas, mas sofrem com o fato de que o tamanho do índice é, muitas vezes, superior ao próprio tamanho do banco de dados, tornando-se impraticável para volumes elevados de dados. Diversos trabalhos buscam reduzir o espaço de armazenamento exigido por estas estruturas de indexação, tornando-se uma opção para melhorar o desempenho dos métodos de pesquisa de similaridade conhecidos, como o da ferramenta BLAST.

A estrutura de indexação MRS é uma opção para um índice armazenado em memória principal, devido ao baixo custo de armazenamento do índice. Vários trabalhos foram desenvolvidos buscando otimizar a função de distância utilizada pela estrutura de indexação, além de aplicações que utilizam a MRS para acelerar o processo de pesquisa, como realizado em conjunto com o BLAST. Alguns outros trabalhos sugerem conceitos como o VBR, que serve como opção a ser implementada em conjunto com a MRS.

CAPÍTULO 4

IMPLEMENTAÇÃO DA ESTRUTURA DE INDEXAÇÃO MRS

Neste capítulo é descrita a implementação da estrutura de indexação MRS, a qual foi realizada neste trabalho de mestrado utilizando a linguagem de programação Java. Esta implementação difere da implementação original da MRS no sentido que os dados biológicos são indexados em disco. Mais especificamente, parte da estrutura de indexação MRS se encontra em memória primária (i.e., em um *buffer-pool*) e o restante da estrutura se encontra em memória secundária, devido ao elevado volume de dados indexados. A implementação descrita neste capítulo será usada para avaliar o desempenho da MRS. Na seção 5.1 é apresentado o conceito sobre *buffer-pool*, enquanto que na seção 5.2 é apresentada a implementação da estrutura de indexação MRS. Na seção 5.3 são apresentadas algumas considerações finais a respeito da implementação realizada.

5.1 Descrição da Aplicação

No contexto biológico, é apresentado em (Kahveci, 2001) alguns testes de desempenho com a estrutura de indexação MRS no qual são utilizados 4 cromossomos humanos individualmente: cromossomo 2, cromossomo 18, cromossomo 21 e cromossomo 22. O cromossomo 18 possui 4 MB de caracteres, enquanto que os outros cromossomos possuem mais de 31 MB de caracteres cada. Os testes de desempenho realizados priorizavam os efeitos no desempenho da estrutura variando os parâmetros da estrutura de indexação (i.e., capacidade do *box* e tamanho da janela), e não prioritariamente a natureza dos dados.

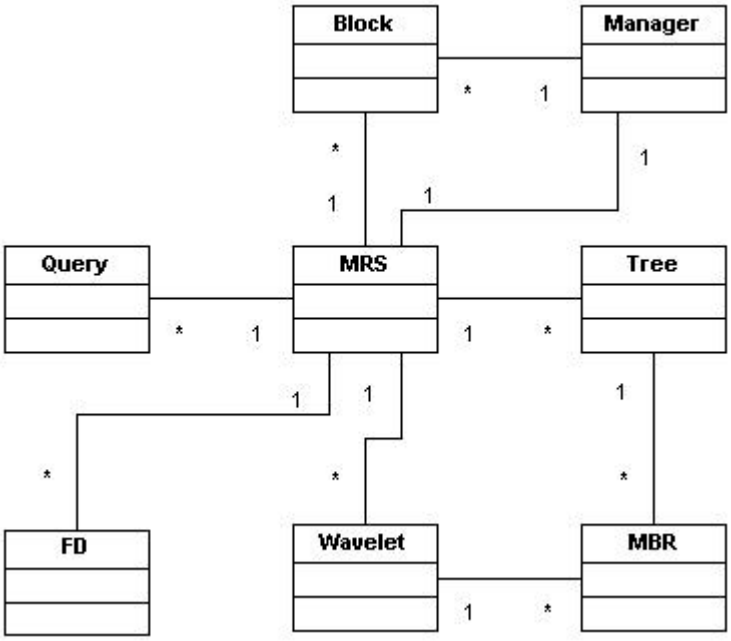


Figura 5.1: Diagrama de Classes da estrutura de indexação MRS.
Fonte: Nakano (2005).

Para a análise do desempenho da MRS foi realizada a implementação desta estrutura de indexação. A primeira versão da estrutura de indexação foi implementada por Kuroshu (2005) e utilizada por Nakano (2005). Esta implementação foi realizada utilizando a linguagem Java, oferecendo suporte para a construção da estrutura de indexação sobre uma determinada *string* (i.e. seqüência biológica) e consulta de uma determinada *string*.

O diagrama de classes desta referida implementação é apresentado na Figura 5.1. A classe principal é a classe *MRS*, responsável pela criação da estrutura de indexação MRS e pela realização das pesquisas *range query*. A classe *Manager* é responsável por quebrar as *strings* em blocos de 4KB, simulando uma página de disco. A estrutura de indexação é armazenada em um vetor bidimensional por um objeto da classe *Tree*, representando as linhas e colunas da estrutura de indexação, onde cada linha corresponde a uma determinada resolução (2^i) e cada coluna corresponde a uma *string* do banco de dados (bloco de 4096 *bytes*). A classe *Tree* é composta por objetos da classe *MBR*, que armazenam objetos da classe *Wavelet* conforme a definição da capacidade do *box*. Sobre cada *string* do banco de dados é

realizada a transformação *wavelet* de suas *substrings*, deslizando uma janela de tamanho determinado, sendo que para cada mapeamento é criado um objeto da classe *Wavelet*. Métodos da classe *FD* são utilizados para medir a distância entre a *string* de consulta e as *strings* mapeadas na estrutura de indexação MRS, sendo que para cada *string* de consulta é criado um objeto da classe *Query*.

A nova implementação realizada neste trabalho não modifica a parte central e funcional do programa, apenas o reorganiza, conforme apresentado no diagrama de classes da Figura 5.2. Isto foi necessário para adaptar a implementação para memória secundária. Toda parte de gerenciamento das *strings* do banco de dados é realizada pela classe *Manager*, responsável por carregar todas as *strings* do banco de dados, dividindo-as em blocos de 4KB, e retornando os blocos requisitados pela classe *MRS*. O resultado de uma pesquisa realizada sobre a estrutura de indexação é armazenado em uma lista de objetos da classe *SearchResult*, responsável por armazenar informações relacionadas ao MBR e a *string* do MBR cuja distância se encontra dentro do raio da pesquisa *range query*. Ademais, para otimização da fase de pesquisa, foi implementado um *buffer-pool*, representado pelas classes *BufferManager* e *BufferNode*.

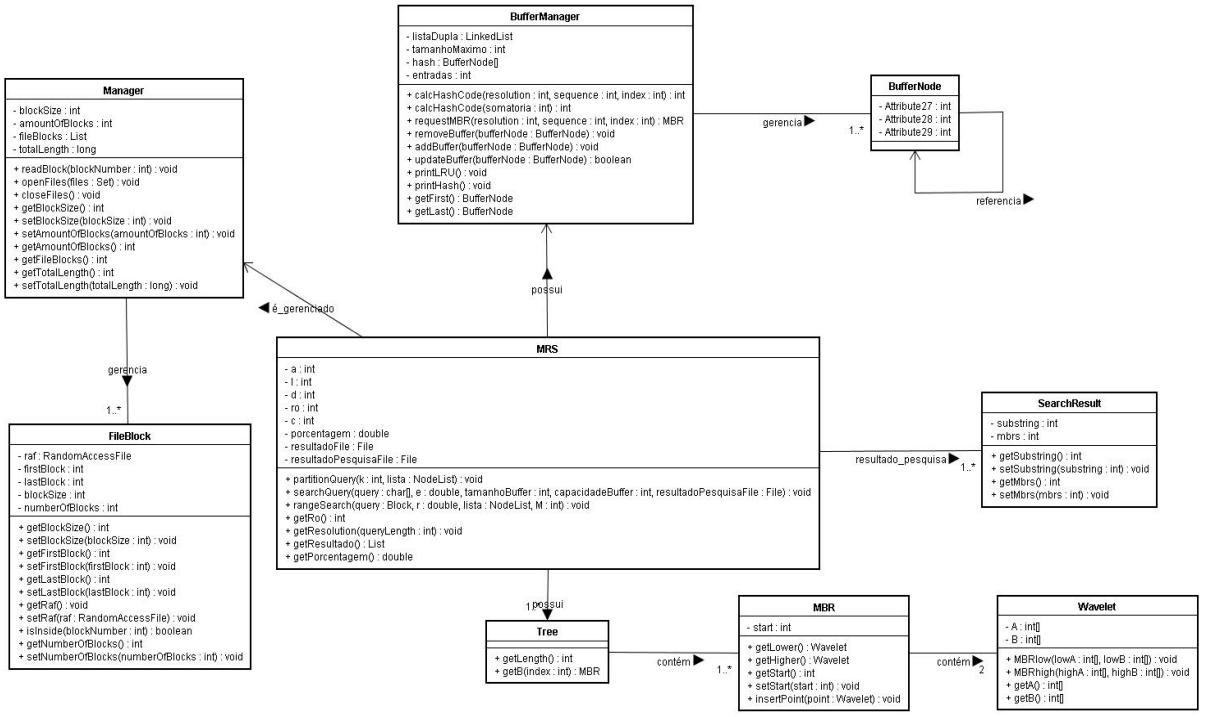


Figura 5.2: Diagrama de Classes da nova implementação da estrutura de indexação MRS

A implementação oferece suporte para o carregamento de diversas *strings* para a construção da estrutura de indexação, possibilitando o teste do desempenho da estrutura de indexação com um elevado volume de dados.

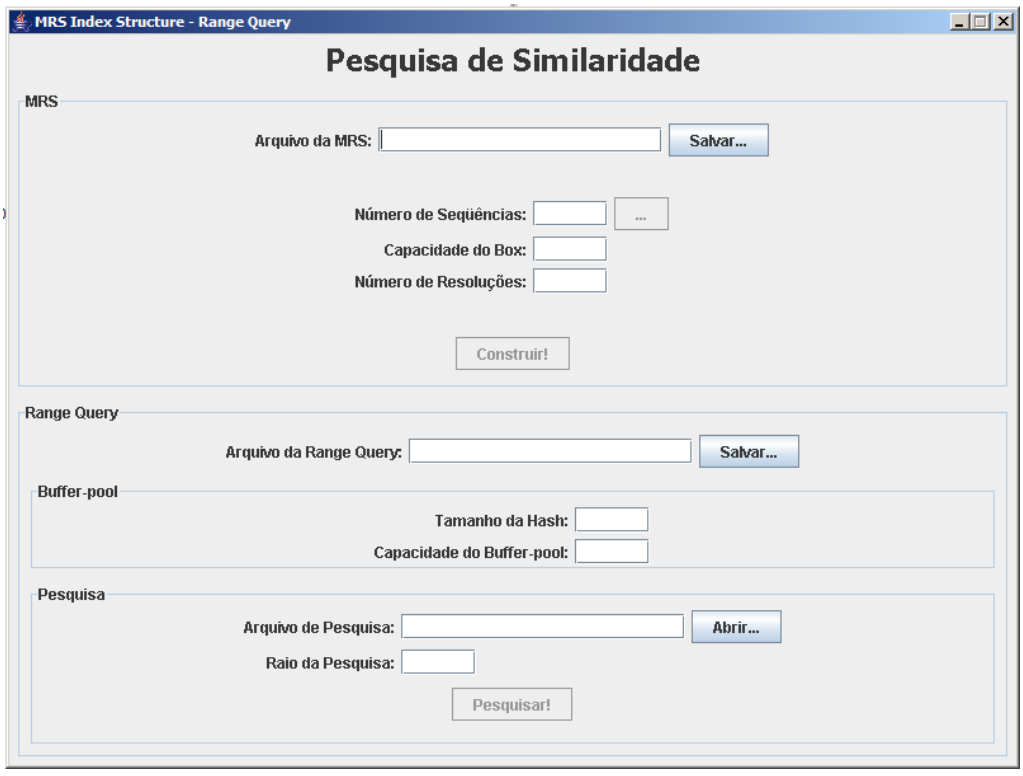


Figura 5.3: Interface Gráfica do programa MRS

Uma interface gráfica foi desenvolvida para melhorar a usabilidade da aplicação, que originalmente era executada em linha de comando. Apesar de que a aplicação em si não será utilizada diretamente por usuários finais, mas será inserido como módulo em ferramentas, como por exemplo, no BLAST, com o intuito de melhorar o desempenho das pesquisas. A interface gráfica principal é apresentada na Figura 5.3, na qual é possível identificar os seguintes passos: construção da estrutura de indexação e pesquisa na estrutura de indexação. Para efetuar as pesquisas na estrutura de indexação é necessário que esta já esteja construída.

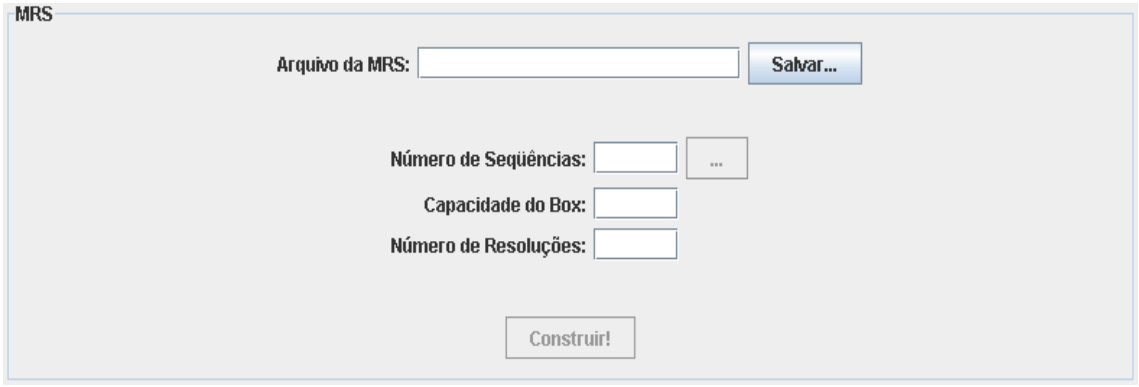


Figura 5.4: Interface Gráfica para Construção da Estrutura de Indexação MRS

A fase de construção da estrutura de indexação MRS é realizada após a definição dos seguintes parâmetros: número de seqüências, capacidade do *box* e número de resoluções disponíveis na estrutura de indexação. Após definir o número de seqüências a serem indexadas, é necessário carregá-las através da caixa de diálogo apresentada na Figura 5.5, através deste carregamento é possível indexar inúmeras seqüências biológicas, sendo que cada seqüência é quebrada em blocos de 4096 *bytes*.

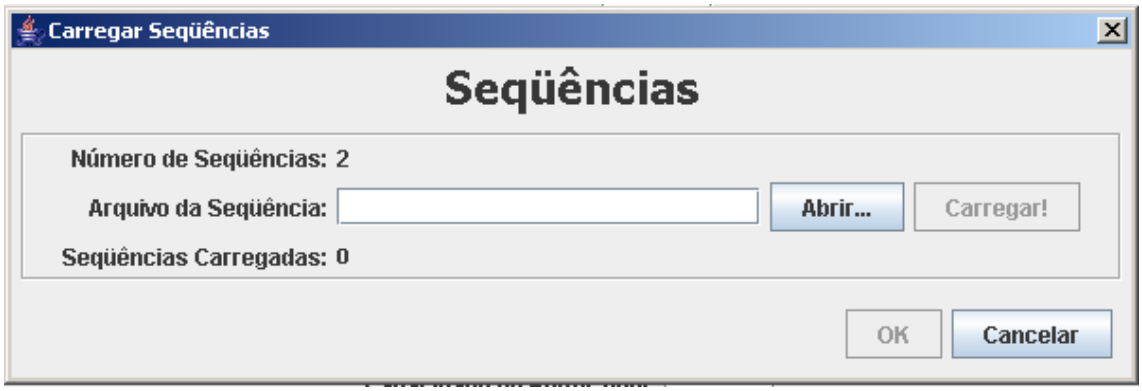


Figura 5.5: Interface Gráfica para Carregamento de Seqüências do Banco de Dados Biológicos

Com a conclusão da fase de construção da estrutura de indexação, é possível realizar a fase de pesquisa utilizando a técnica *range query*, apresentada na Figura 5.6 nesta é fornecida uma determinada seqüência de consulta, assim como o erro da pesquisa, que corresponde ao raio da pesquisa. Informações relacionadas ao *buffer-pool*, tamanho da *hash* e capacidade de entradas do *buffer-pool*, tipicamente em torno de 5% do total dos dados armazenados, também são fornecidas durante a fase de pesquisa com o intuito de otimizar o processo quando este necessitar acessar a memória secundária em busca de MBRs que não se encontrem em memória principal.

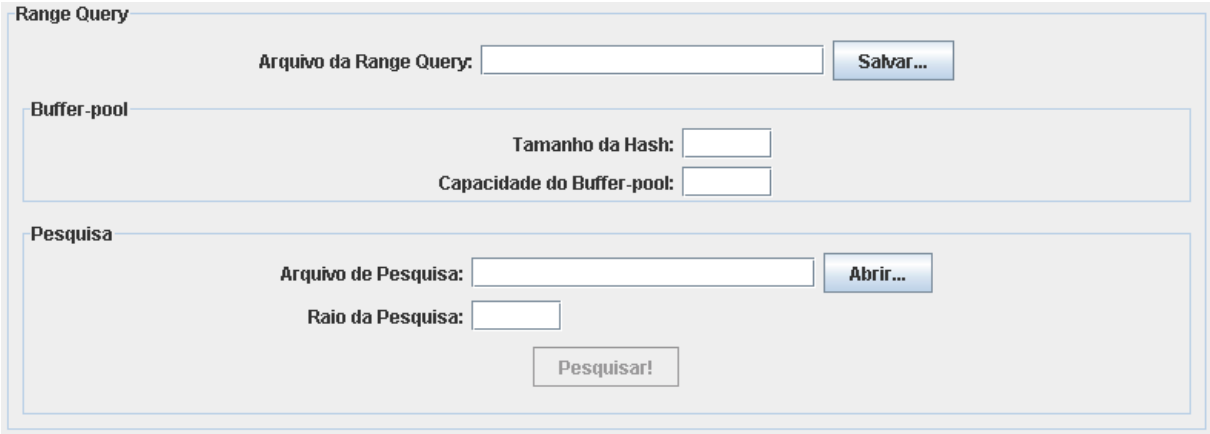


Figura 5.6: Interface Gráfica para o Processamento da Pesquisa *Range Query*

Para uma melhor análise dos resultados obtidos, dois arquivos são gerados. Um dos arquivos é responsável por armazenar informações relacionadas à estrutura de indexação construída: número de seqüências utilizadas para a construção da estrutura de indexação,

número de caracteres, nome dos arquivos de seqüência, número de blocos de 4096 *bytes* em que foram divididas as seqüências, número de dimensões utilizadas, número de resoluções disponíveis na estrutura, capacidade do *box*, horário de início e término da construção da estrutura de indexação. O outro arquivo é gerado para cada pesquisa *range query* realizada, armazenando informações relacionadas à pesquisa: nome do arquivo da estrutura de indexação, horário de início e término da pesquisa, *string* de consulta, tamanho da *string* de consulta, raio da pesquisa (erro), resultado da pesquisa (quantidade de MBRs que se encontram dentro do raio da pesquisa por bloco) e porcentagem de E/S que será necessário realizar para uma pesquisa completa sobre as *strings*.

5.2 Considerações Finais

A estrutura de indexação MRS é uma estrutura compacta, projetada para memória principal. A implementação da estrutura de indexação MRS possibilitou analisar o desempenho da estrutura de indexação com um elevado volume de dados, ocasionando o crescimento do tamanho do índice, de forma a extrapolar a quantidade de memória principal disponível, sendo que parte do índice foi armazenado em memória secundária.

Os testes de desempenho realizados com a estrutura de indexação MRS são apresentados no Capítulo 6. O código-fonte da implementação em Java utilizada para os testes de desempenho está listado no ANEXO A.

CAPÍTULO 6

TESTES DE DESEMPENHO

Neste capítulo são apresentados os testes de desempenho realizados com a implementação da estrutura de indexação MRS (descrita no Capítulo 5). Os dados são seqüências de nucleotídeos provenientes do Projeto Genoma Humano. Aplicando-se a estrutura de indexação MRS sobre um elevado volume de dados biológicos, composto por seqüências de DNA, armazenados em um banco de dados biológico, avalia-se o desempenho da estrutura de indexação quando esta é alocada em memória secundária.

Na seção 6.1 é apresentada uma introdução ao ambiente de teste utilizado, na seção 6.2 são apresentadas as características a serem avaliadas na estrutura de indexação MRS. Na seção 6.3 é descrito o conjunto de seqüências utilizadas para a realização de consultas, para então, na seção 6.4 serem apresentados os testes de desempenho realizados.

6.1 Introdução

Nakano (2005) propõe um ambiente de teste para a análise de estruturas de indexação para pesquisa de similaridade em bancos de dados biológicos, enfatizando a investigação do uso de métodos de acesso para realizar a pesquisa de similaridade em memória principal ou secundária.

No ambiente proposto por Nakano (2005), são considerados como fatores que possam afetar o desempenho de métodos de acesso para dados biológicos no suporte à pesquisa de similaridade:

- Tipo de dado
- Tamanho das seqüências
- Volume de dados
- Tamanho da consulta

As pesquisas de similaridade são constantemente realizadas com os diversos tipos de dados disponíveis. Exemplo disto é a ferramenta BLAST, que oferece implementações para pesquisas de similaridade entre: aminoácidos e proteínas (Blastp), nucleotídeos e nucleotídeos (Blastn e Tblastx), nucleotídeos e proteínas (Blastx), além de aminoácidos e proteínas (Tblastn).

Neste trabalho, seqüências de nucleotídeos foram utilizadas para a investigação do desempenho da estrutura de indexação MRS, em específico, seqüências de DNA provenientes do Genoma Humano (cromossomos humanos). O tamanho dos cromossomos humanos variam desde 33.000.000 pares de base correspondendo ao menor cromossomo humano até 246.000.000 pares de bases para o cromossomo mais longo.

Considerando o volume dos dados, o custo de armazenamento de seqüências biológicas é elevado. Um milhão de pares de bases (chamado de *megabase*) de dados de seqüência de DNA é equivalente à aproximadamente 1 *megabyte* de espaço de armazenamento de dados em um computador. Como o genoma humano possui aproximadamente 3 bilhões de pares de base, são necessários em torno de 3 *gigabytes* de espaço de armazenamento de dados em um computador.

Tabela 6.1: Volume dos cromossomos humanos em *mega* pares de base.

Cromossomo	*Mega* pares de bases (volume)
1	245
2	242
3	198
4	189
5	179
6	169
7	157
8	145
9	139
10	134
11	133
12	131
13	113
14	105
15	99
16	88
17	78
18	75
19	63
20	61
21	46
22	49
X	128
Y	57

Dos projetos genoma completamente seqüenciados, o genoma com maior quantidade de bases, disponibilizado pelo NCBI (*National Center for Biotechnology Information*), é o genoma humano (BENSON, 2006), servindo de referência para os demais genomas. Para nosso ambiente de teste, com base no tamanho da seqüência de DNA dos cromossomos (Tabela 6.1), estes foram divididos em três categorias (Tabela 6.2): volumes pequenos (tamanho inferior a 100.000.000 pares de base), volumes médios (tamanho entre 100.000.000 e 200.000.000) e volumes grandes (superiores a 200.000.000).

Categoria	Tamanho (pb)	Cromossomos
Pequeno	< 100.000.000	15, 16, 17, 18, 19, 20, 21, 22 e Y
Médio	≥ 100.000.000 e < 200.000.000	3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 e X
Grande	≥ 200.000.000	1 e 2

Tabela 6.2: Cromossomos divididos entre as categorias pequeno, médio e grande.

Com a intensificação do interesse em projetos genoma, necessita-se freqüentemente pesquisar similaridade em genomas já concluídos. Realizar a pesquisa sobre, por exemplo, um cromossomo humano é uma tarefa praticável com algoritmos em memória principal. Mas, considerando a pesquisa sobre um genoma completo, torna-se impraticável o uso destes algoritmos, devido ao elevado volume dos dados. Para o ambiente de teste, volumes pequenos, médios e grandes de dados foram utilizados, tendendo a aumentar a cada bateria de teste, de modo a ultrapassar o limite da memória principal disponível para a estrutura de indexação MRS, avaliando-se seu desempenho em memória secundária. Os tamanhos da consulta variaram entre 1.000, 2.000 e 4.000 pares de base.

6.2 Análise de Desempenho da Estrutura de Indexação MRS

A estrutura de indexação MRS é uma estrutura compacta, podendo ser armazenada em memória principal por computadores pessoais. Mas, existe um limite de memória principal para estes computadores, a tecnologia atual permite o uso em torno de 3 a 4GB de memória principal para computadores pessoais.

Os testes de desempenho realizados avaliaram o desempenho da estrutura de indexação MRS residindo em memória principal e em memória secundária. A estrutura de indexação MRS foi analisada segundo dois aspectos: construção da estrutura de indexação e pesquisa na estrutura de indexação.

Análise da Construção da Estrutura de Indexação MRS

Primeiramente, é necessário realizar a construção do índice sobre os dados a serem pesquisados. Uma estrutura de indexação apropriada sobre grandes seqüências pode levar muitas horas para ser construída, se tornando ineficiente construí-la para cada pesquisa (HUNT, 2001). Por esta razão, torna-se necessário avaliar o custo de tempo para a construção da estrutura de indexação MRS, que é um índice originalmente projetado para memória principal.

Para a análise de desempenho da estrutura de indexação MRS em memória secundária, faz-se necessária a avaliação do custo de armazenamento da estrutura de indexação. Com base no custo de armazenamento e na quantidade de memória principal disponível, é possível determinar se parte do índice se encontra armazenada em memória secundária.

Análise da Pesquisa Range Query sobre a Estrutura de Indexação MRS

A pesquisa *range query* é responsável por retornar todos os MBRs que se encontram dentro do raio da pesquisa. Para isto, é calculada a distância entre a seqüência de consulta e os MBRs da estrutura de indexação, e caso a distância calculada seja inferior ao limite (raio), o MBR é incluído na lista de MBRs resultantes. A pesquisa *range query* implementada não realiza o cálculo da distância entre a *string* de consulta e as *strings* contidas nos MBRs resultantes, para isto, é necessário calcular posteriormente a distância de cada *string* contida em cada MBR.

Para a análise de desempenho da estrutura de indexação MRS no processamento de consultas é avaliado o tempo para a conclusão da pesquisa.

6.3 Obtenção dos Dados Reais

As seqüências de DNA pertencentes aos cromossomos humanos foram obtidas através do endereço ftp://ftp.ensembl.org/pub/current_homo_sapiens/data/fasta/dna/. O Ensembl (BIRNEY, 2006) é um projeto conjunto entre o EMBL – *European Bioinformatics Institute* (EBI) e o *Wellcome Trust Sanger Institute* (WTSI) para desenvolver um sistema de *software* que produza e mantenha anotações automáticas sobre determinados genomas eucarióticos. Os genomas de 14 cordados estão atualmente disponíveis através do Ensembl, de mamíferos como Humano e Rato até o cordado "primitivo" *Ciona intestinalis*. Os genomas de três modelos chave eucariotos (levedura, mosca e minhoca) também são importados de seus respectivos banco de dados com o intuito de fornecer fácil integração de informação destes organismos com cordados. E por fim, um número limitado de genomas de insetos também é disponibilizado pelo Ensembl, dado sua participação no consórcio *Vectorbase*.

6.4 Testes de Desempenho

Os testes de desempenho foram divididos em 6 baterias de teste. Para cada bateria, foi submetido um conjunto de 100 consultas composto por: 30 seqüências sintéticas, 30 seqüências obtidas a partir de fragmentos de seqüências armazenadas no banco de dados biológicos, 30 seqüências geradas aleatoriamente (Tabela 6.3). Os tamanhos das seqüências variaram entre 1000, 2000 e 4000 pares de base (pb). Para completar 100 consultas, foram utilizadas 10 seqüências adicionais obtidas a partir de fragmentos de seqüências armazenadas no banco de dados biológicos, mas com tamanhos aleatórios variando entre 1000 e 4000 pares de base.

As seqüências sintéticas foram geradas com base nas características dos cromossomos, considerando a porcentagem das bases nitrogenadas na composição das seqüências. Apesar das seqüências serem formadas a partir de um alfabeto $\Sigma = \{A, C, G, T, N\}$, desconsideramos

o caractere N durante a geração das seqüências de consulta sintéticas. Assim, as seqüências de consulta são todas formadas pelo alfabeto $\Sigma = \{A, C, G, T\}$.

Conjunto de consultas				
	1000pb	**2000pb**	**4000pb**	**1000 – 4000pb**
Seqüências Sintéticas	10	10	10	
Seqüências Reais (Fragmentos)	10	10	10	10
Seqüências Aleatórias	10	10	10	

Tabela 6.3: Conjunto de consultas

Para todos os testes, foram utilizados os mesmos parâmetros para a construção da estrutura de indexação MRS, variando as seqüências armazenadas no banco de dados biológicos. A capacidade do *box* foi definida em 1000. Assim, cada MBR criado é capaz de armazenar, no máximo, 1000 pontos e o número de resoluções em 11 devido à restrição de implementação que utiliza blocos de 4KB, permitindo uma janela de tamanho máximo igual a 4KB. Os testes foram realizados em um computador AMD Turion™ 64 Mobile Processor 1.8 GHz com 1 GB de memória RAM.

As baterias de teste também foram divididas conforme o seguinte critério: baterias de teste com apenas um cromossomo no banco de dados biológico (baterias 1, 2 e 3) e baterias de teste com mais de um cromossomo no banco de dados biológico (baterias 4, 5 e 6).

Cada bateria de teste consiste de:

- *Bateria de Teste 1*: testes com banco de dados biológicos apenas com cromossomo de volume pequeno.

- *Bateria de Teste 2*: testes com banco de dados biológicos apenas com cromossomo de volume médio.
- *Bateria de Teste 3*: testes com banco de dados biológicos apenas com cromossomo de volume grande.
- *Bateria de Teste 4*: testes com banco de dados biológicos com cromossomos de volume pequeno.
- *Bateria de Teste 5*: testes com banco de dados biológicos com cromossomos de volume médio.
- *Bateria de Teste 6*: testes com banco de dados biológicos com cromossomos de volume grande.

Bateria de Teste 1

A bateria de teste 1 analisou o desempenho da estrutura de indexação MRS com um banco de dados biológicos composto por um cromossomo humano pertencente à categoria de volume pequeno. Para compor o banco de dados biológicos foi escolhido o cromossomo 20. As características do cromossomo 20 são apresentadas na Tabela 6.4.

	A%	**C%**	**G%**	**T%**	**N%**	**Tamanho**
Cromossomo 20	26.46398	20.99389	21.06046	26.78780	4.69385	62.435.904

Tabela 6.4: Características do Cromossomo 20

O índice MRS foi construído em 4,04 minutos ocupando um espaço de 435 MB. O tempo de resposta necessário para a realização da pesquisa *range query* com o conjunto de testes formado pelas 100 seqüências de consulta foi de 44,7 minutos.

Bateria de Teste 2

O banco de dados biológicos da bateria de teste 2 foi composto por um cromossomo humano pertencente à categoria de volume médio. O cromossomo escolhido para compor o banco de dados biológicos, cujas características são apresentadas na Tabela 6.5, foi o cromossomo 8 com 145M pares de base.

	A%	C%	G%	T%	N%	Tamanho
Cromossomo 8	29.1851	19.5797	19.5813	29.1502	2.5035	146.274.826

Tabela 6.5: Características do Cromossomo 8

O índice MRS foi construído em 9,47 minutos, ocupando um espaço de 983 MB. O tempo gasto para a realização do conjunto de testes foi de 108,8 minutos.

Bateria de Teste 3

A bateria de teste 3 utilizou um banco de dados biológicos composto por um cromossomo pertencente à categoria de volume grande. O cromossomo 2, com 242M pares de base, foi escolhido para compor o banco de dados biológicos (Tabela 6.6).

	A%	C%	G%	T%	N%	Tamanho
Cromossomo 2	29.2107	19.6758	19.6916	29.2643	2.157369	242.951.149

Tabela 6.6: Características do Cromossomo 2

O índice ocupou um tamanho de 1,5 GB, sendo construído em 15,4 minutos. O tempo gasto para a realização do conjunto de consultas foi de 261,8 minutos.

Análise das Baterias de Teste 1, 2 e 3

Conforme o Gráfico 6.1 mostra, o tempo gasto para a construção da estrutura de indexação para as baterias de teste 1, 2 e 3 seguiu um comportamento linear, onde o tempo gasto para a construção do índice foi proporcional ao volume dos dados. O crescimento do

tamanho do índice também apresentou um comportamento próximo ao linear (Gráfico 6.2), com o tamanho do índice quase dobrando a cada aumento de volume.

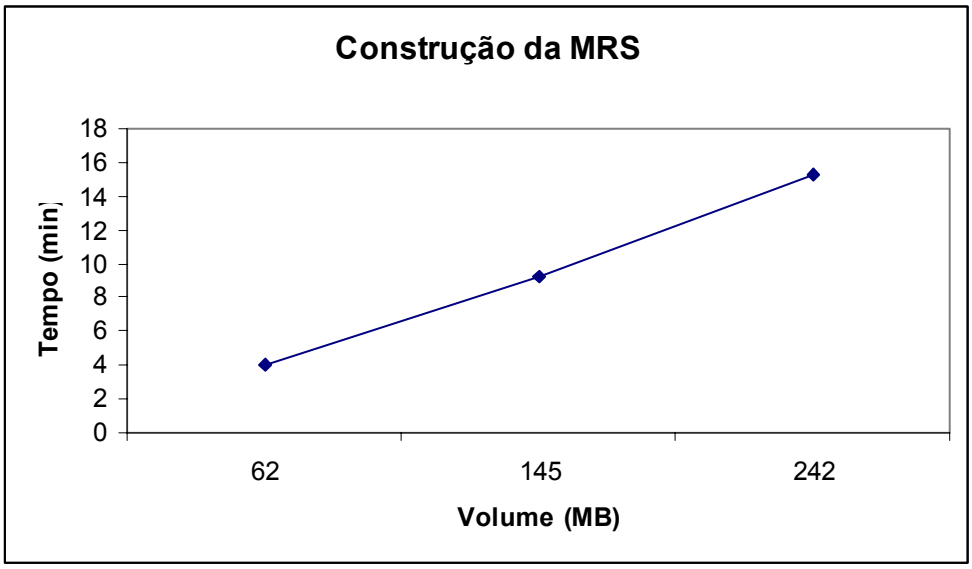


Gráfico 6.1: Custo de Tempo para a construção da MRS conforme o aumento do Volume.

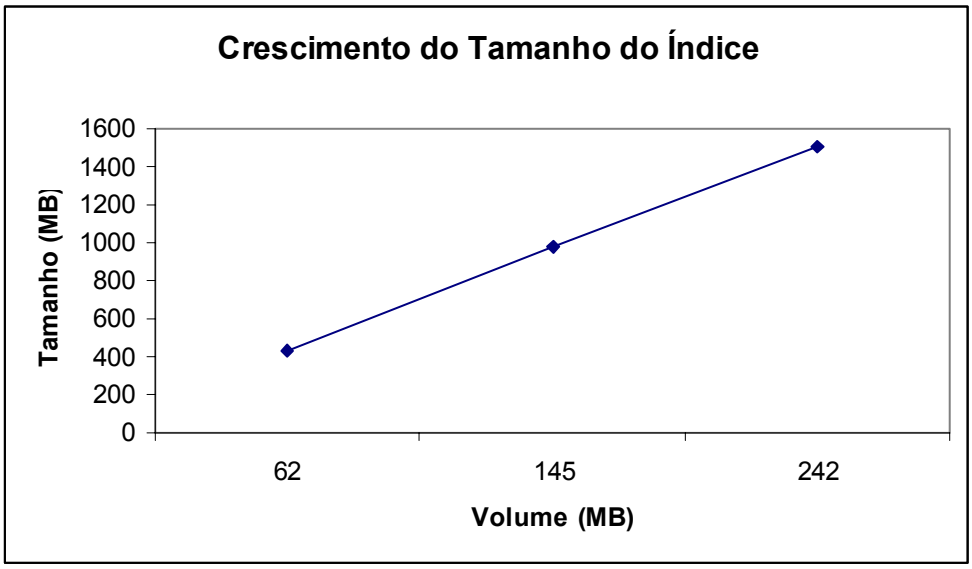


Gráfico 6.2: Custo de Armazenamento da MRS conforme o aumento do Volume.

O tempo de resposta do conjunto de consultas para a estrutura de indexação do cromossomo 2 foi superior ao tempo de resposta apresentando pela estrutura de indexação criada nos outros dois cromossomos, conforme demonstra o Gráfico 6.3. Para um aumento de aproximadamente 83MB no volume do BDB da bateria de teste 1 para a 2, houve um aumento de aproximados 64,1 minutos, enquanto que para um aumento do volume de 97MB entre a bateria de teste 2 e 3 ocorreu um aumento de aproximados 153 minutos.

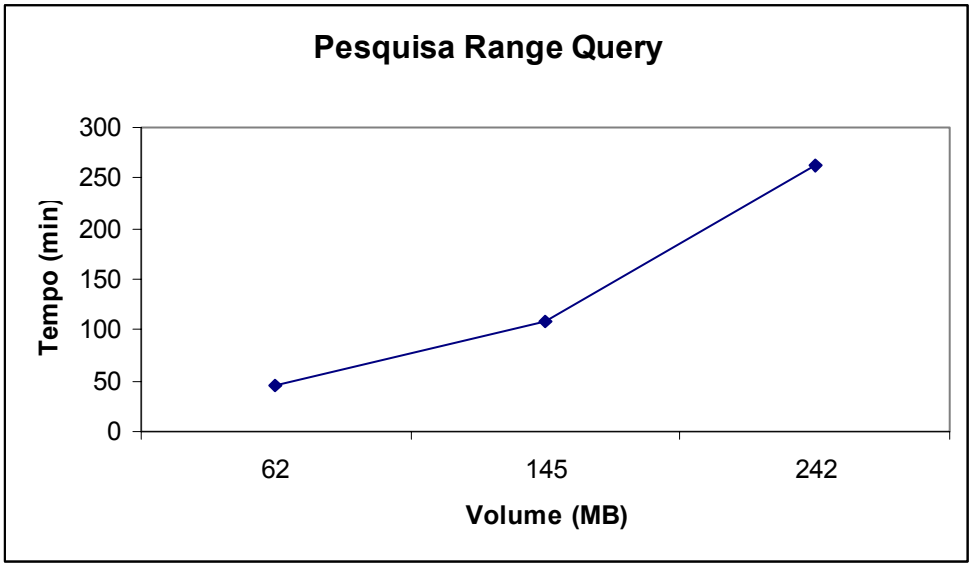


Gráfico 6.3: Custo de Tempo para a pesquisa *Range Query* conforme o aumento do Volume.

Bateria de Teste 4

A bateria de teste 4 utilizou um banco de dados biológicos composto por cromossomos pertencentes à categoria de volume pequeno. Durante a execução dos testes, o volume de dados do banco de dados biológicos foi elevado gradualmente, com a adição de cromossomos de pequeno volume, conforme demonstra o Gráfico 6.4.

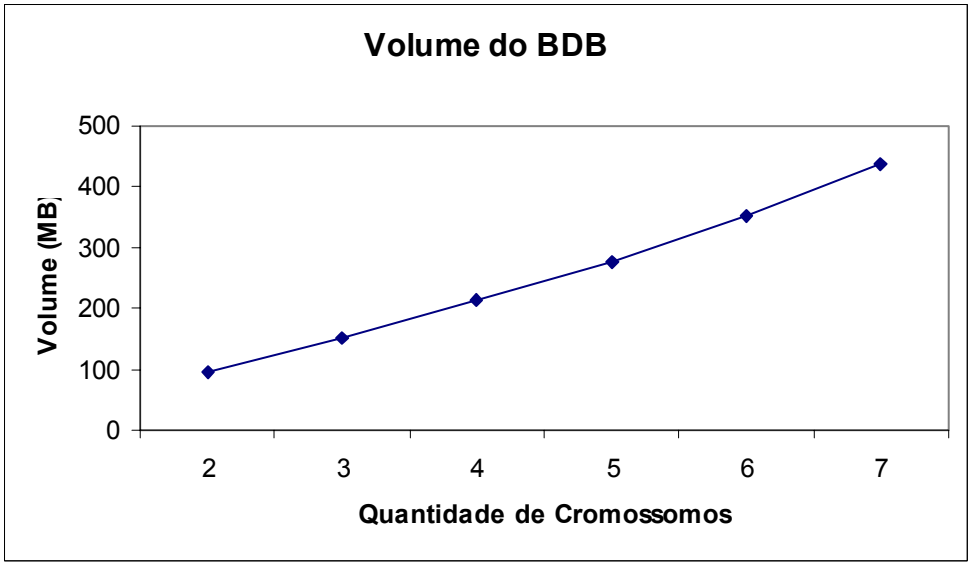


Gráfico 6.4: Crescimento do Volume do BDB com o aumento da quantidade de cromossomos.

Com base no tamanho das seqüências de DNA dos cromossomos, para cada aumento de volume foi selecionado o cromossomo de menor tamanho disponível para compor o banco de dados biológicos (Tabela 6.7).

Quantidade de Cromossomos	Cromossomos	Volume de Dados
2	21 e 22	95MB
3	21, 22 e Y	152MB
4	21, 22, Y e 20	213MB
5	21, 22, Y, 20 e 19	276MB
6	21, 22, Y, 20, 19 e 18	351MB
7	21, 22, Y, 20, 19, 18 e 17	439MB

Tabela 6.7: Cromossomos e Volume do BDB.

O tempo para a construção da estrutura de indexação MRS é apresentado no Gráfico 6.5. Com base nos tempos de construção e no volume do banco de dados biológicos, nota-se que o tempo aumentou aproximadamente de forma linear, seguindo o mesmo comportamento do volume do banco de dados biológicos.

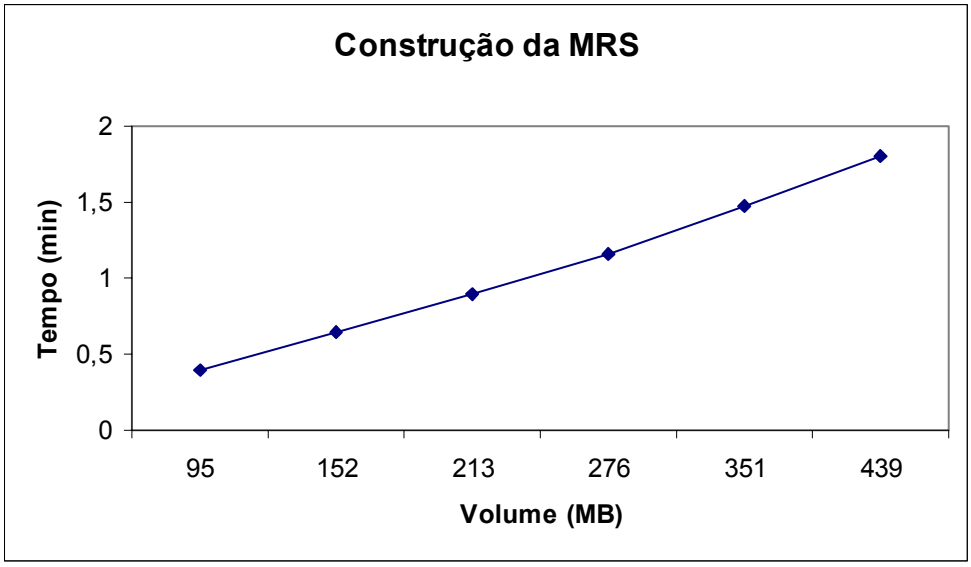


Gráfico 6.5: Custo de Tempo para a construção da MRS conforme aumento do Volume do BDB.

O crescimento do tamanho do índice, até um determinado volume de dados, foi proporcional ao volume do banco de dados biológicos, conforme demonstra o Gráfico 6.6. A partir do momento em que a quantidade de memória principal disponível não é mais suficiente para o armazenamento do índice, parte da estrutura de indexação é armazenada em memória secundária. Nota-se que a partir do volume de 276MB, não há uma grande variação no tamanho do índice.

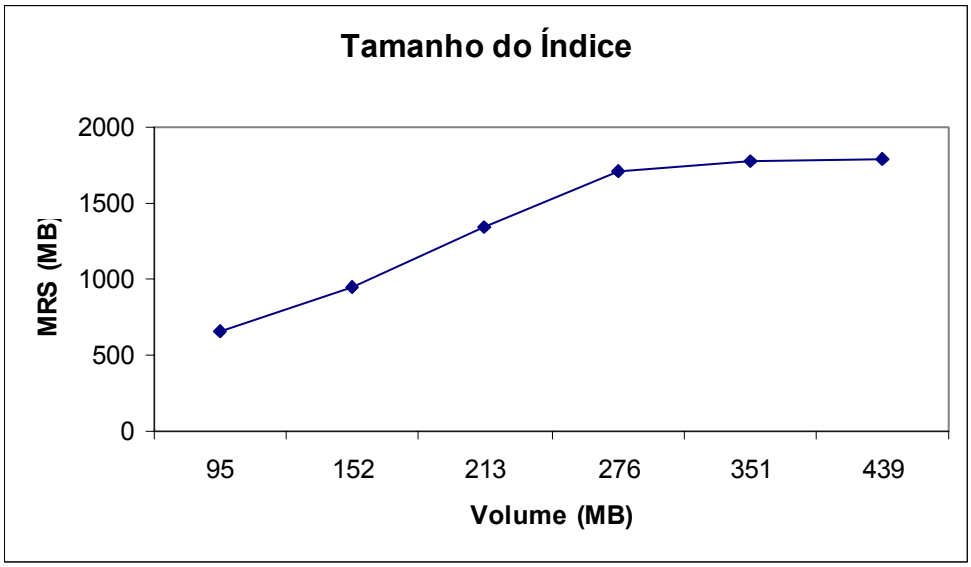


Gráfico 6.6: Custo de Armazenamento do índice com o aumento do Volume do BDB.

O desempenho da estrutura de indexação MRS no processamento da *range query* é apresentado no Gráfico 6.7 e Gráfico 6.8. No Gráfico 6.7 é apresentado o tempo de resposta do conjunto de consultas realizado sobre a estrutura de indexação MRS elevando o volume dos dados. Traçando um paralelo entre o tempo de resposta do conjunto de consultas e o tamanho do índice, nota-se que enquanto é possível armazenar o índice da estrutura de indexação em memória principal, o desempenho da pesquisa é linear, da mesma forma que o crescimento do volume dos dados. A partir do momento em que parte do índice da estrutura de indexação é armazenada em memória secundária, ocorre uma perda de desempenho da estrutura de indexação, elevando-se o tempo de resposta para a pesquisa. Fato percebido a partir do volume de 213MB armazenados no banco de dados biológicos. Quando o tamanho do índice da estrutura de indexação é muito superior à quantidade de memória principal,

necessitando o armazenamento de uma grande parte do índice em memória secundária, ocorre uma drástica queda no desempenho da pesquisa, ocasionando um elevado aumento no tempo de resposta do conjunto de consultas. Este aumento é evidenciado no Gráfico 6.8, onde para um banco de dados biológicos com volume igual à 351MB, grande parte da estrutura de indexação MRS é armazenada em memória secundária.

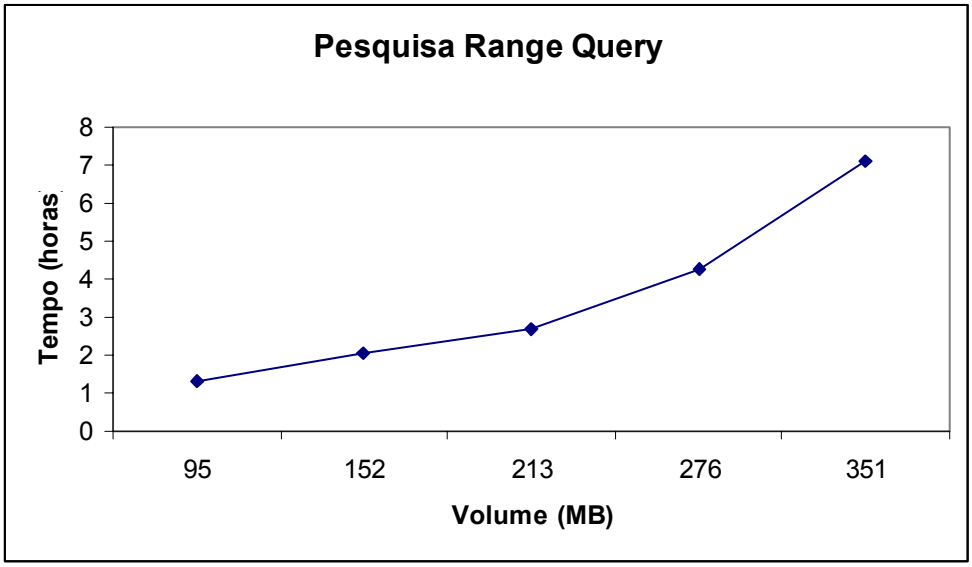


Gráfico 6.7: Pesquisa *Range Query* para o BDB composto por até 6 cromossomos.

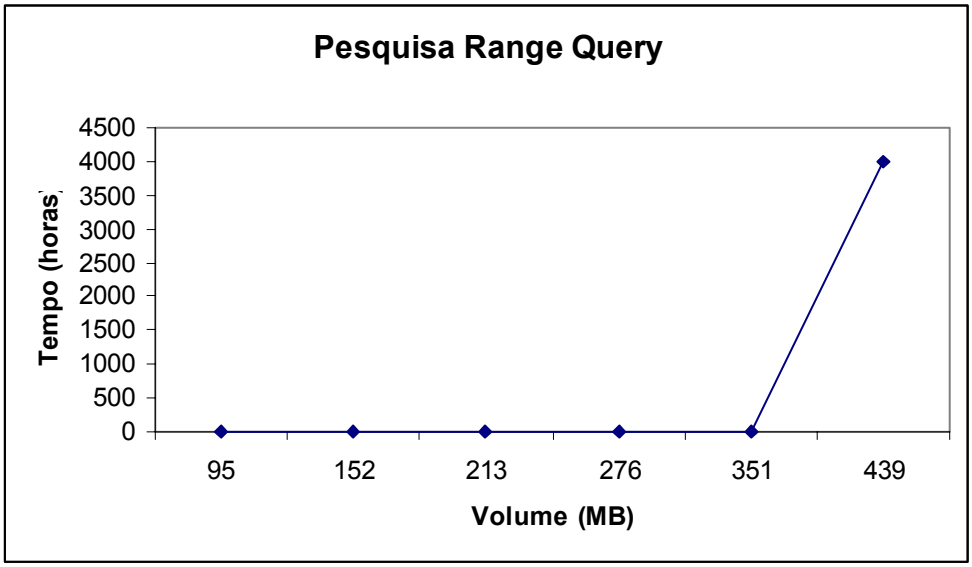


Gráfico 6.8: Pesquisa Range Query para o BDB composto pelos 7 cromossomos.

Bateria de Teste 5

A bateria de teste 5 avaliou o desempenho da estrutura de indexação MRS com um banco de dados biológicos composto por cromossomos humanos de volume médio. Da mesma maneira que a bateria de teste 4, o volume de dados do banco de dados biológicos foi elevado a cada conjunto de teste (Tabela 6.8), mas com seqüências de volume médio, diferentemente da bateria de teste anterior, que considerava seqüências de volume pequeno.

Quantidade de Cromossomos	Cromossomos	Volume de Dados
2	14 e 13	218MB
3	14, 13 e 12	349MB
4	14, 13, 12 e 11	482MB

Tabela 6.8: Cromossomos e Volume do BDB.

A taxa de crescimento do volume do banco de dados biológicos se manteve constante conforme demonstra o Gráfico 6.9. Para cada conjunto, o volume do banco de dados biológicos aumentou aproximadamente de 130 a 140MB.

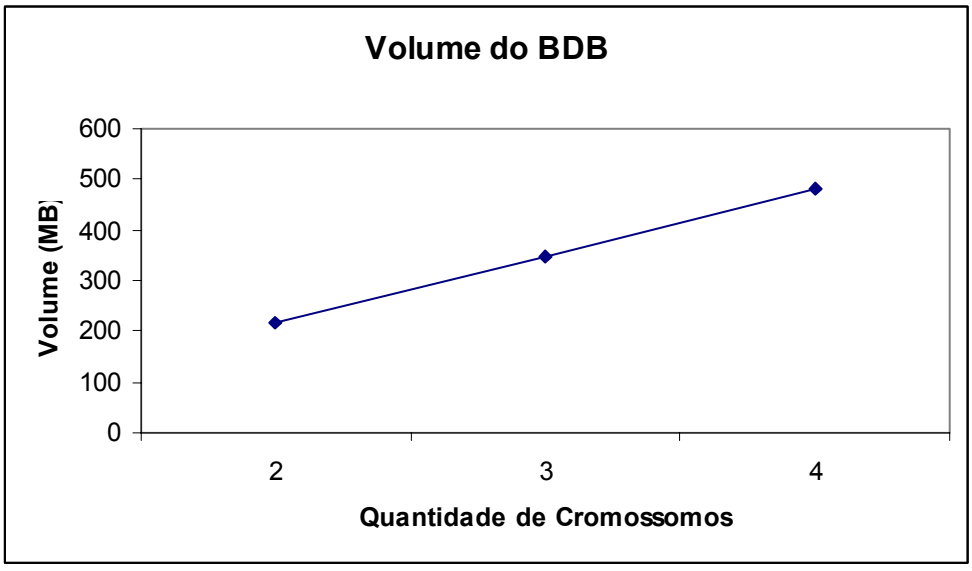


Gráfico 6.9: Crescimento do Volume do BDB com o aumento da quantidade de cromossomos.

O custo de tempo para a criação do índice da estrutura de indexação MRS apresentou um comportamento linear, com um aumento aproximado de 30 segundos no custo de tempo para mais ou menos 130MB de volume de dados, conforme demonstra o Gráfico 6.10.

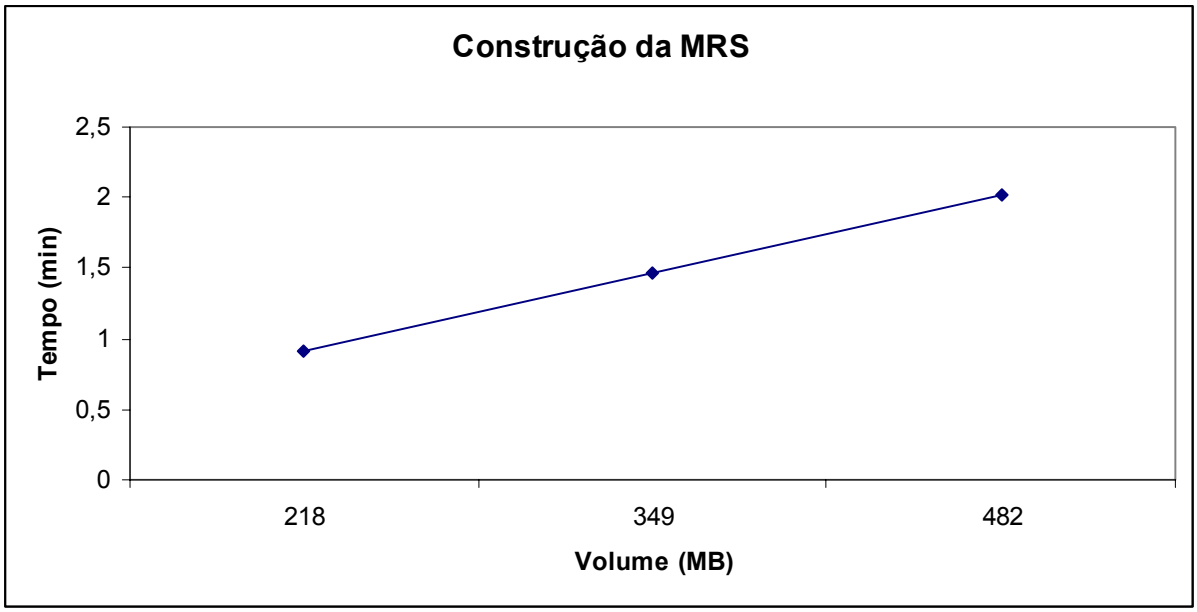


Gráfico 6.10: Custo de Tempo para a construção da MRS.

O crescimento do tamanho do índice, apresentado no Gráfico 6.11, apresenta o mesmo comportamento relatado na bateria de teste anterior. O tamanho do índice tende a estabilizar quando este está próximo ao limite de memória disponível, destacando-se a ação do *garbage collector* como provável motivo para o tamanho do índice estabilizar.

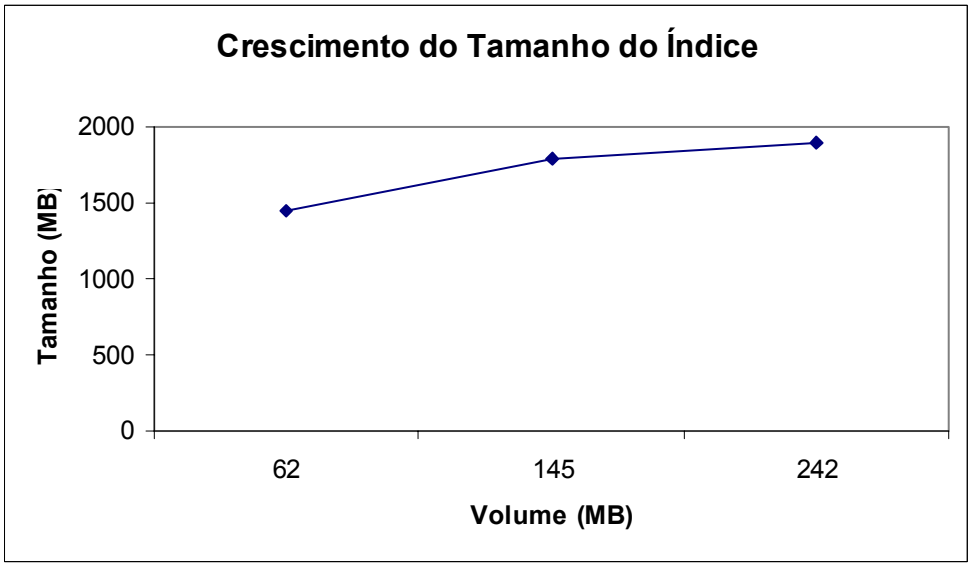


Gráfico 6.11: Custo de Armazenamento do índice com o aumento do Volume do BDB.

O desempenho da estrutura de indexação MRS no processamento da *range query* é apresentado no Gráfico 6.12 e Gráfico 6.13. Enquanto a maior parte do índice é armazenada em memória principal, são obtidos resultados proporcionais ao volume do banco de dados biológicos. Mas no momento em que o tamanho do índice é muito superior à quantidade de memória principal disponível, o desempenho é prejudicado, elevando drasticamente o tempo de resposta.

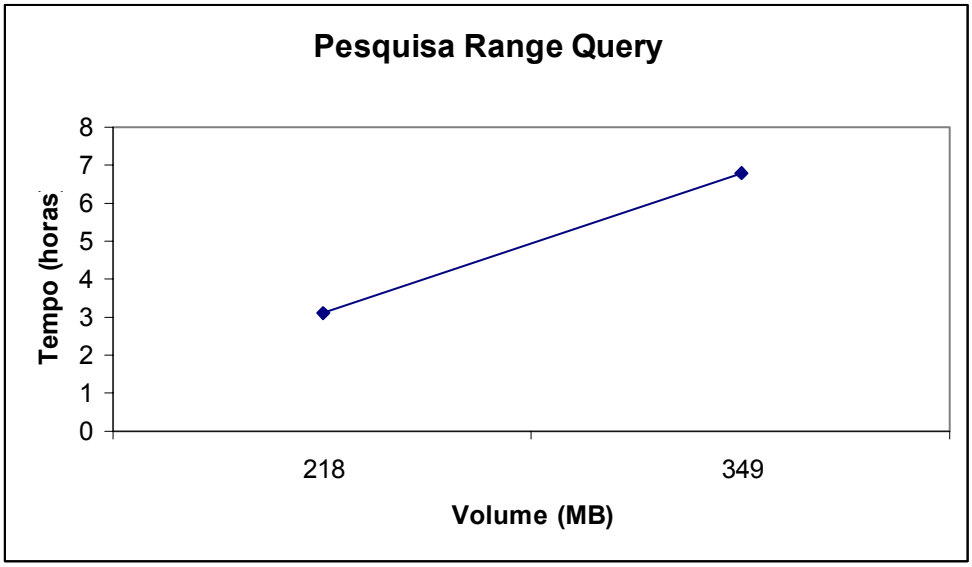


Gráfico 6.12: Pesquisa *Range Query* para o BDB composto por até 3 cromossomos.

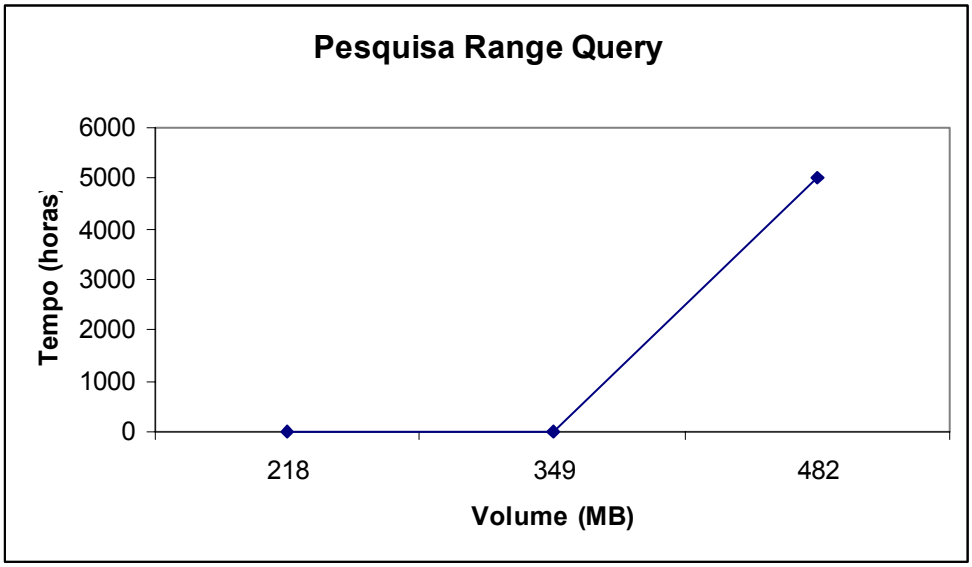


Gráfico 6.13: Pesquisa Range Query para o BDB composto pelos 3 cromossomos.

Comparação dos Resultados das Baterias 4 e 5

Os resultados dos testes de desempenho realizados nas baterias 4 e 5 são similares, como demonstra o Gráfico 6.14. Nos momentos em que o volume do banco de dados biológicos é similar, o tempo de resposta da pesquisa para o conjunto de consultas também é similar, ocorrendo o mesmo quando o tamanho do índice para as duas baterias de teste excedem a quantidade de memória principal disponível, ocasionando o armazenamento do índice em memória secundária.

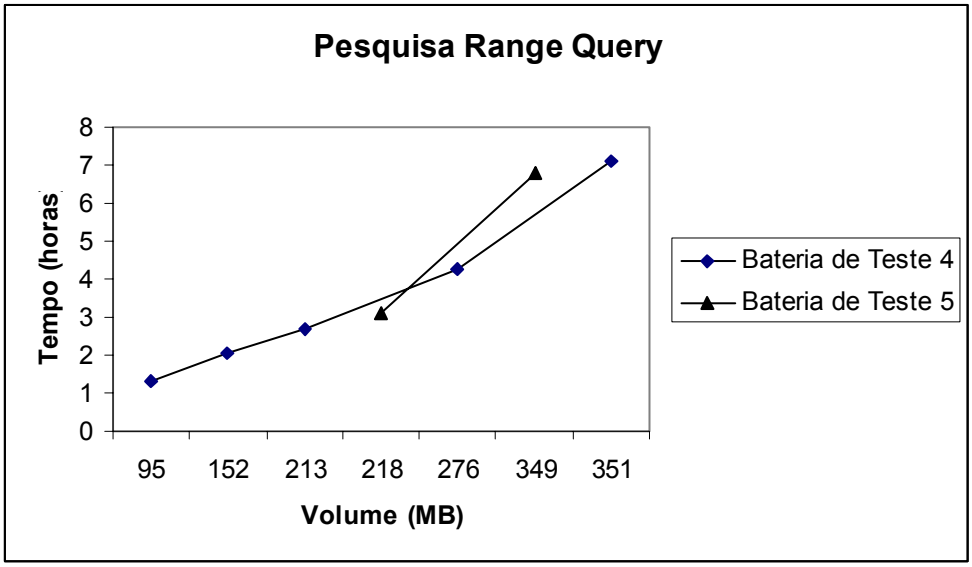


Gráfico 6.14: Custo de Tempo da pesquisa *Range Query* para as baterias de teste 4 e 5.

Bateria de Teste 6

A bateria de teste 6 envolveu um banco de dados biológicos composto por seqüências pertencentes à categoria de volume grande. A priori, o crescimento do volume do banco de dados biológicos da bateria de teste 6 aumentaria a cada conjunto de testes, adicionando uma nova seqüência pertencente à categoria de volume grande. Porém, como esta categoria possui apenas dois cromossomos, apenas um conjunto de teste foi realizado. O volume do banco de dados biológicos é apresentado na Tabela 6.9 e no Gráfico 6.15.

Quantidade de Cromossomos	Cromossomos	Volume de Dados
2	1 e 2	487MB

Tabela 6.9: Cromossomos e Volume do BDB.

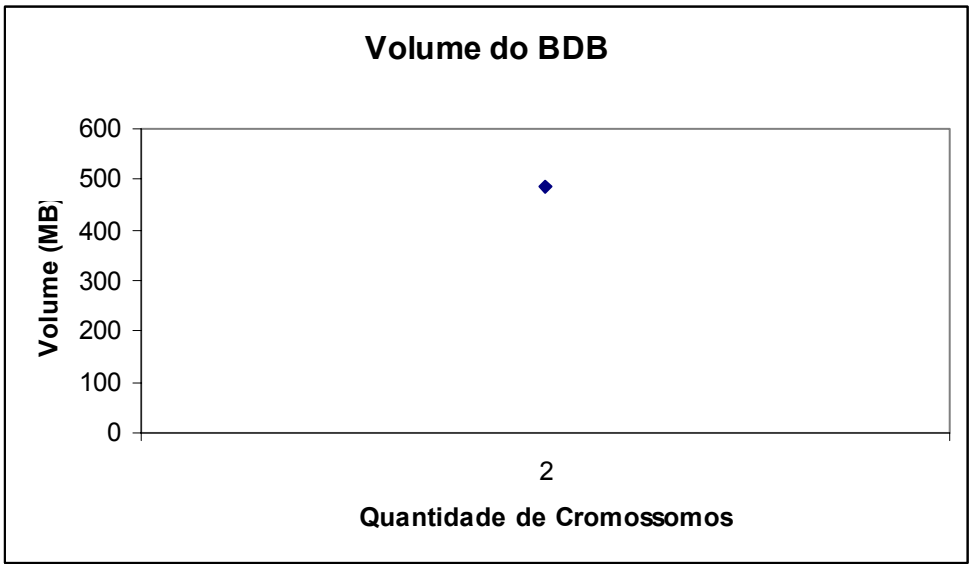


Gráfico 6.15: Volume do BDB conforme a quantidade de cromossomos armazenados no BDB.

A máquina virtual Java possui uma área de *heap* responsável por armazenar os objetos criados durante a execução da aplicação. Ao inicializar a máquina virtual, é possível estender esta área de acordo com a quantidade de memória principal disponível (parâmetro -Xmx). Mesmo estendendo a área de *heap* ao máximo para as configurações de *hardware* utilizadas, não foi possível criar o índice para o volume de dados existente no banco de dados biológicos, ocorrendo um erro na máquina virtual por falta de memória para a área de *heap* (*OutOfMemoryError*). O mesmo ocorreu para testes adicionais realizados nas baterias de teste 4 e 5 envolvendo um maior volume de dados no banco de dados biológicos.

O tamanho da estrutura de indexação MRS, conforme visto no Gráfico 6.2, Gráfico 6.6 e Gráfico 6.11, não condiz com a expectativa de que o índice da MRS ocupe apenas de 1% a 2% do volume do BD. Uma explicação plausível seria alguma falta de otimização da implementação estrutura de indexação MRS, ou até mesmo a questão de não ser previsível a ação do *garbage collector*. Ou ainda, uma análise mais profunda do comportamento da máquina virtual Java, relacionado aos tamanhos dos objetos criados durante a execução da aplicação.

6.5 Considerações Finais

O projeto da estrutura de indexação MRS é baseado em memória principal, considerando que o tamanho do índice construído na excede a quantidade de memória principal disponível. Mas com o elevado volume de dados disponibilizado em bancos de dados biológicos e a taxa de crescimento exponencial destes dados, é necessário considerar a possibilidade de que o tamanho do índice extrapole a quantidade de memória principal, fazendo-se necessário o uso da memória secundária.

Os resultados dos testes de desempenho para as baterias 1, 2 e 3, nos quais é possível armazenar o índice em memória principal, demonstraram que a estrutura de indexação MRS possui um desempenho linear, na qual o tamanho do índice e o tempo de resposta da pesquisa *range query* acompanham o aumento do volume dos dados. Mas para as baterias de teste 4, 5 e 6, foi possível identificar uma queda de desempenho na estrutura de indexação quando o tamanho desta ultrapassou o limite de memória principal, necessitando o acesso à memória secundária.

Testes de desempenho envolvendo volumes maiores de dados não foram possíveis de serem realizados devido a restrições da máquina virtual Java, que impossibilita alocar uma maior quantidade de espaço para o *heap*, área responsável por armazenar os objetos criados durante a execução da aplicação.

CAPÍTULO 7

CONCLUSÃO

Aplicações envolvendo dados de *string*, em geral, produzem uma grande quantidade de dados que são freqüentemente pesquisados. Um exemplo consiste na pesquisa de similaridade em bancos de dados biológicos. Diversos BDBs são responsáveis por armazenar dados de seqüências, constituídas por cadeias de caracteres (*strings*) pertencentes à um determinado alfabeto (ex: Σ = {A, C, G, T, N}), tendo seus volumes crescendo à uma taxa exponencial.

Atualmente, a maioria dos algoritmos de pesquisa de similaridade, como o BLAST, são algoritmos em memória principal que utilizam a técnica da varredura completa do banco de dados. O uso de um método de indexação eficiente pode direcionar a pesquisa a uma porção do BDB com elevado potencial de similaridade, evitando a varredura completa dos dados.

Dos diversos métodos estudados, a estrutura de indexação MRS se destacou pelo fato de que, diferentemente de técnicas baseadas em árvores de sufixo, o tamanho do índice gerado é pequeno, possibilitando seu armazenamento em memória principal.

A estrutura de indexação MRS é uma estrutura projetada para funcionar em memória principal. Porém, como a taxa de crescimento dos dados armazenados em bancos de dados é superior à quantidade de memória principal disponível, existe a preocupação de que, para um determinado banco de dados, não seja possível armazenar a MRS em memória principal.

Neste trabalho, a estrutura MRS foi adaptada para permitir a indexação de dados biológicos em memória secundária. Com o intuito de avaliar o desempenho da estrutura de indexação MRS em memória secundária, através de um ambiente de teste, foram realizados testes com dados extraídos do Genoma Humano. Durante os testes, o volume dos dados

armazenados no banco de dados biológicos foi aumentado de tal maneira que o índice criado para o banco de dados biológicos extrapolasse a quantidade de memória principal disponível, visando avaliar o desempenho em memória secundária da estrutura de indexação MRS considerando o uso do índice em computadores pessoais, os quais possuem restrições quanto à quantidade máxima de memória principal permitida.

Os resultados de desempenho demonstraram que, a partir do momento em que parte da estrutura de indexação MRS é armazenada em memória secundária, há uma queda no desempenho. Esta perda de desempenho, apesar de ser quase que imperceptível durante a fase de construção do índice, é evidenciado durante a realização das pesquisas *range query*.

A adaptação da estrutura de indexação MRS para memória secundária pode ser utilizada como um módulo para melhorar o desempenho de ferramentas que utilizam a varredura completa dos dados para a realização da pesquisa de similaridade, como é o caso do BLAST. O desempenho da MRS em memória secundária apresentou uma perda de desempenho para um volume elevado de dados, mas não é possível determinar se esta perda de desempenho torna a MRS impraticável, sendo necessário comparar a adaptação da MRS com as demais estruturas de indexação para memória secundária.

A linguagem Java foi utilizada para compatibilizar os resultados dos testes de desempenho com os obtidos por Nakano (2005), sendo que uma versão em C++ já foi implementada para a estrutura de indexação MRS em memória principal, restando implementar a adaptação da estrutura MRS para memória secundária.

Este trabalho buscou estudar os seguintes tópicos:

- Estudo e descrição de métodos de alinhamentos de seqüências;
- Estudo e descrição do problema de pesquisa de similaridade;
- Estudo e descrição da ferramenta BLAST;
- Estudo e descrição da estrutura de indexação MRS;

- Levantamento do estado da arte no tema estruturas de indexação para dados de *strings*;

Dentre as principais contribuições deste trabalho, pode-se citar:

- Adaptação da estrutura de indexação MRS para comportar inúmeras seqüências do banco de dados biológicos em memória secundária;
- Avaliação do desempenho da estrutura de indexação MRS em memória secundária;

Algumas extensões a este trabalho incluem:

- Comparação do desempenho da estrutura de indexação MRS com outros métodos de indexação, considerando um elevado volume de dados;
- Análise do desempenho da estrutura de indexação MRS para outros fatores, tal como o tamanho da consulta;
- Adaptação da estrutura de indexação MRS para construção com resoluções maiores;
- Substituição dos MBRs pelos VBRs proposto por (SUN, 2003) e realização de testes de desempenho;
- Revisão e otimização do processo de criação da estrutura de indexação MRS visando reduzir o custo de armazenamento; e
- Implementação da nova *Frequency Distance* proposta em (KAHVECI, 2002) e realização de testes de desempenho.
- Análise de desempenho da estrutura de indexação MRS implementada na linguagem C++.

REFERÊNCIAS BIBLIOGRÁFICAS

Altschul, S. F.; Gish, W.; Miller, W.; Myers, E. W.; Lipman, D. J. Basic Local Alignment Search Tool. Journal of Molecular Biology, Vol. 215, pag. 403-410, 1990.

Altschul, S. F.; Madden, T. L.; Schäffer, A. A.; Zhang, J.; Zhang, Z.; Miller, W.; Lipman, D. J. *Gap*ped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research, Vol. 25, No. 17, pag. 3389-3402, 1997.

Barker, W. C.; Garavelli, J. S.; Huang, H.; Mcgarvey, P. B.; Orcutt, B. C.; SRINIVASARAO, G. Y.; Xiao, C.; YEH, L. L.; Ledley, R. S.; Janda, J. F.; Pfeiffer, F.; Mewes, H.; Tsugita, A.; WU, C. The Protein Information Resource (PIR). Nucleic Acids Research, vol. 28, n. 1, pág. 41-44, 2000.

Barillot, E.; Pook, S.; Guyon, F.; Cussat-Blanc, C.; Viara, E.; Vaysseix, G. The HuGeMap Database: interconnection and visualization of human genome maps. Nucleic Acids Research, vol. 27, pág. 119-122, 1999.

Benson, D. A.; Karsch-Mizrachi, I.; LIPMAN, D. J.; OSTELL, J.; WHEELER, D. L. Genbank: update. Nucleic Acids Research, vol.32, D23-D26, 2004.

Benson, D. A.; Karsch-Mizrachi, I.; Lipman, D. J.; Ostell, J.; Wheeler, D. L. GenBank. Nucleic Acids Research, Vol. 33, 2005.

Benson, D. A.; Karsch-Mizrachi, I.; Lipman, D. J.; Ostell, J.; Wheeler, D. L. GenBank. Nucleic Acids Research, Vol. 34, 2006.

Birney, E; Andrews, D.; Caccamo, M; Chen, Y.; Clarke, L.; Coates, G.; Cox, T.; Cunningham, F.; Curwen, V.; Cutts, T.; Down, T.; Durbin, R.; Fernandez-Suarez, X. M.; Flicek, P.; Gräf, S.; Hammond, M.; Herrero, J.; Howe, K.; Iyer, V.; Jekosch, K.; Kähäri, A.; Kasprzyk, A.; Keefe, D.; Kokocinski, F.; Kulesha, E.; London, D.; Longden, I.; Melsopp, C.; Meidl, P.; Overduin, B.; Parker, A.; Proctor, G.; Prlic, A.; Rae, M.; Rios, D.; Redmond, S.; Schuster, M.; Sealy, I.; Searle, S.; Severin, J.; Slater, G.; Smedley, D.; Smith, J.; Stabenau, A.; Stalker, J.; Trevanion, S.; Ureta-Vidal, A.; Vogel, J.; White, S.; Woodwark, C.; Hubbard, T. J. P. Ensembl 2006. Nucleic Acids Res. 2006 Jan 1; 34 Database issue:D556-D561.

Brown, D. G.; Li, M.; Ma, B. Homology Search Methods. In *The Practical Bioinformatician*, ed: L. Wong. Singapore: World Scientific, 2004, 217-244.

Cameron, M.; Williams, H. E.; Cannane, A. Improved *Gap*ped Alignment in BLAST. IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB), vol.1, n.3, p. 116-129, Jul. 2004.

Ciferri, R. R. **Análise de Desempenho de Métodos de Acesso Multidimensionais Aplicada a Sistemas de Informações Geográficas.** Proposta de Tese de Doutorado, Universidade Federal de Pernambuco, 2002.

Dayhoff, M.; Schwartz, R.; Orcutt, B. A model of evolutionary change in proteins. Atlas of Protein Sequence and Structure, 5, 1978.

Eilbeck, K.; Lewis, S.; Mungall, C. J.; Yandell, M.; Ashburner, M. What is the Sequence Ontology? In the Computational Genomics Poster, 2003.

Fasman, K. H.; Letovsky, S. I.; Cottingham, R. W.; Kingsbury, D. T. Improvements to the GBDtm Human Genome Data Base. Nucleic Acids Research, vol. 24, n. 1, pág. 57-63, 1996.

Flybase Consortium. The FlyBase database of the Drosophila genome projects and community literature. Nucleic Acids Research, vol. 31, n. 1, pág. 172-175, 2003.

Folks, M. J. File Structures. Addison-Wesley Longman Publishing Co., Inc, Boston, USA, 2nd edition, 1999.

Galperin, M. Y. The Molecular Biology Database Collection: 2005 update. Nucleic Acids Research, vol. 33, n. D5-D24, 2005.

Gopalan, V.; Tan, T. W.; Lee, B. T. K.; Ranganathan, S. Xpro: database of eukaryotic protein-encoding genes. Nucleic Acids Research, vol. 32, D59-D63, 2004.

Guttman, A. **R-trees: a dynamic index structure for spatial searching.** June 1984 ACM SIGMOD Record, Proceedings of the 1984 ACM SIGMOD international conference on Management of data SIGMOD '84, Volume 14 Issue 2.

Harger, C.; Chen, G.; Farmer, A.; Huang, W.; Inman, J.; Kiphart, D.; Schilkey, F.; SKUPSKI, P.; Weller, J. The Genome Sequence DataBase. Nucleic Acids Research, vol. 28, n. 1, pág. 31-32, 2000.

Hunt, E. PJama Stores and Suffix Tree Indexing for Bioinformatics Applications. 2000.

Kahveci, T.; Singh, A. K. An Efficient Index Structure for *String* Databases. VLDB, Roma, Itália, pag. 351-360, 2001.

Korf, I.; Yandell, M.; Bedell, J. BLAST. O' Reilly & Associates, Inc. Gravenstein Highway North, Sebastopol, CA, July 2003.

Kuroshu, R. Indexação de Dados Biológicos. Universidade Estadual de Maringá, Trabalho de Graduação, 2005.

Letovsky, S. I.; Cottingham, R. W.; Porter, C. J.; Li, P. W. D. GDB: the Human Genome Database. Nucleic Acids Research, Vol. 26, No. 1, 1998.

Letunic, I.; Goodstadt, L.; Dickens, N. J.; Doerks, T.; Schultz, J.; Mott, R.; Ciccarelli, F.; Copley, R. R.; Ponting, C. P.; Bork, P. Recent improvements to the SMART domain-based sequence annotation resource. Nucleic Acids Research, vol. 30, n. 1, pág. 242-244, 2002.

Maidak, B. L.; Cole, J. R.; Parker, C. T.; JR; Garrity, G. M.; Larsen, N.; LI, B.; Lilburn, T. G.; Mcgaughey, M. J.; Olsen, G. J.;Overbeek, R.; Pramanik, S.; Schmidt, T. M.; Tiedje, J. M.;

Woese, C. R. A new version of the RDP (Ribosomal Database Project). Nucleic Acids Research, vol. 27, pág. 171-173, 1999.

Mewes, H. W.; Frishman, D.; Gruber, C.; Geier, B.; Haase, D.; Kaps, A.; Lemcke, K.; Mannhaupt, G.; Pfeiffer, F.; Schüller, C.; Weil, B. Mips: a database for genomes and protein sequences. Nucleic Acids Research, vol. 28, n. 1, pág. 37-40, 2000.

Miyazaki, S.; Sugawara, H.; Ikeo, K.; Gojobori, T.; Tateno, Y. DDBJ in the stream of various biological data. Nucleic Acids Research, vol. 32, D31-D34, 2004.

Nakano, M. Uma Abordagem Para a Pesquisa de Similaridade de Seqüências de Nucleotídeos e Proteínas. Universidade Estadual de Maringá, curso de Ciência da Computação, trabalho de graduação, abril de 2003.

Nakano, M. Pesquisa de Similaridade de Seqüências em Bancos de Dados Biológicos: Um Estudo Comparativo entre o Uso da Técnica de Varredura Completa e o Uso de Estruturas de Indexação. Universidade Estadual de Maringá, mestrado em Ciência da Computação, proposta de dissertação, agosto de 2004.

Nakano, M. Um Ambiente de Teste para Pesquisa de Similaridade em Bancos de Dados Biológicos. Universidade Estadual de Maringá, Dissertação de Mestrado, 2005.

Navarro, G.; Baeza-Yates, R.; Sutinen, E.; Tarhio, J. Indexing Methods for Approximate *String* Matching. Technical Report, University of Chile, 2000.

Needleman, S. B.; Wunsch, C. D. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. Journal of Molecular Biology, Vol. 48, 1970.

Okura, V. K. Bioinformática de Projetos Genoma de Bactérias. Universidade Estadual de Campinas, Instituto de Computação, dissertação de Mestrado, fevereiro de 2002.

Oliveira, R. S.; Carissimi, A. S.; Toscani, S. S. **Sistemas Operacionais.** Editora Sagra Luzzatto, Porto Alegre – Rio Grande do Sul, 2001.

Schulze-Kremer, S. Adding Semantics to Genome Databases: Towards an Ontology for Molecular Biology. In 5th International Conference on Intelligent Systems for Molecular Biology, AIII Press, Menlo Park, pág. 272-275, 1997.

Smith, T. F.; Waterman, M. S. Identification of Common Molecular Subsequences. Journal of Molecular Biology, Vol. 147, 1981.

Stevens, R.; Goble, C. A.; Bechhofer, S. Ontology-based Knowledge Representation for Bioinformatics. Journal Briefings in Bioinformatics, 2000.

Stoesser, G.; Baker, W.; Broek, A.; Camon, E.; Garcia-Pastor, M.; Kanz, C.;Kulikova, T.; Leinonen, R.; Lin, Q.; Lombard, V.; Lopez, R.; Redaschi, N.; Stoeh, P.; Tuli, M. A.;Tzouvara, K.; Vaughan, R. The EMBL Nucleotide Sequence Database. Nucleic Acids Research, vol. 30, n. 1, pág. 21-26, 2002.

Sun, H.; Ozturk, O.; Ferhatosmanoglu, H. CoMRI: A Compressed Mutli-Resolution Index Structure for Sequence Similarity Queries. IEEE Computer Society Bioinformatics Conference, Stanford, CA, pag. 553-558, agosto de 2003.

Tateno, Y.; Imanishi, T.; Miyazaki, S.; Fukami-Kobayashi, K.; Saitou, N.; Sugawara, H.; Gojobori, T. DNA Data Bank of Japan (DDBJ) for genome scale research in life science. Nucleic Acids Research, vol. 30, n. 1, pág. 27-30, 2002.

Wheeler, D. L.; Church, D. M.; Lash, A. E.; Leipe, D. D.; Madden, T. L.; Pontius, J. U.; Schuler, G. D.; Schrimi, L. M.; Tatusova, T. A.; Wagner, L.; Rapp, B. A. Database resources of the National Center for Biotechnology Information. Nucleic Acids Research, vol. 29, n. 1, pág. 11-16, 2001.

Yang, J.; Wang, W.; Yu, P. BASS: Approximate Search on Large *String* Databases. Proceedings, 16th International Conference on Scientific and Statistical Database Management, pag. 181-190, 2004.

ANEXOS

ANEXO A

Classe MRS

```
package mrs.model;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;

import mrs.model.bufferPool.BufferManager;
import mrs.model.bufferPool.BufferNode;
import mrs.model.bufferPool.MBRBufferNode;

public class MRS {
      Manager manager;
      //RangeSearch search;
      private int a;          //where 2^a is the shortest query string

      private int l;          //amount of resolutions

      private int d;          //amount of strings on DB

      private int ro;

      private int c;          //box capacity (must be pair)

      private Tree[][] T;

      private List<SearchResult> resultado;

      private double porcentagem;

      private BufferManager bufferManager;

      private File resultadoFile;

      private File resultadoPesquisaFile;

      public MRS(Set<File> files, int start, int resolutions, int box, File
resultadoFile) throws IOException {
            this.resultadoFile = resultadoFile;
            //characters = ro
            int blockSize = 4096;
            manager = new Manager(blockSize, files);

            a  = start;
            l  = resolutions;
            d  = manager.getAmountOfBlocks();
            ro = 5; //A, C, G, T, N
```

```
            c  = box;

            T = new Tree[l][d];

            PrintWriter out = null;
            try {
                  out = new PrintWriter(new
FileWriter(this.resultadoFile));
                  out.write("Número de Seqüências: " + files.size());
                  out.write("\nNúmero de Caracteres: " +
manager.getTotalLength());
                  out.write("\nSeqüências: ");
                  for(File f : files) {
                        out.write(f.getName() + "\n");
                  }
                  out.write("Número de Blocos: " + d);
                  out.write("\nRo: " + ro);
                  out.write("\nResolução: " + l);
                  out.write("\nBox capacity: " + c);
                  long tempoInicio = System.currentTimeMillis();
                  out.write("\nInício da construção: " + tempoInicio);

                  //for each string on DB
                  for(int j=0; j<d; j++) {
                        System.out.println("\nOn String " + (j + 1) + " of
" + d);
                        Block block = null;
                        block = manager.readBlock(j);
                        //for each windon size (resolution)
                        for(int i=0; i<l; i++) {
                              T[i][j] = new Tree((int)Math.pow(2,a+i), ro,
c, block);
                        }
                        block = null;
                  }
                  long tempoFim = System.currentTimeMillis();
                  out.write("\nTérmino da construção: " + tempoFim);
                  out.write("\nTempo Gasto: " + (tempoFim - tempoInicio));
            }
            catch(Exception ex) {
                  ex.printStackTrace();
            }
            finally {
                  if(out != null) {
                        out.flush();
                        out.close();
                  }
            }
            manager.closeFiles();

            //Eliminating references
            manager = null;
            out = null;
            Runtime.getRuntime().gc();
      }

      private void partitionQuery(int k, NodeList lista) {
            if(k > 0)
//                Para each resolution
                  for(int i=l-1; i>=0; i--) {
                        if(Math.pow(2,i)<=k) {
```

```
      lista.insertNode((int)(Math.pow(2,i)*Math.pow(2,a)));
                              System.out.print("Valor inserido na lista: ");

      System.out.println((int)(Math.pow(2,i)*Math.pow(2,a)));
                              partitionQuery(k-
(int)(Math.pow(2,i)*Math.pow(2,a)),lista);
                              return;
                        }
                  }
            else return;
      }

      public int getRo() {
            return ro;
      }

      public void searchQuery(char[] query, double e, int tamanhoBuffer,
int capacidadeBuffer, File resultadoPesquisaFile) {
//          Writing results to file
            this.resultadoPesquisaFile = resultadoPesquisaFile;
            bufferManager = new BufferManager(capacidadeBuffer,
tamanhoBuffer);
            if(Math.abs((double)query.length/Math.pow(2,a)) ==
(double)query.length/Math.pow(2,a)) {
                  //if the querys length is a multiple of 2^a
                  int k,M;
                  k = (int)query.length/(int)Math.pow(2,a);
                  NodeList lista = new NodeList();
                  partitionQuery(k,lista);
                  lista.printList();

                  M =
T[getResolution(lista.getFirst().getValue())][0].getLength();

                  Block q = new Block(query.length);
                  q.setBlock(query);
                  rangeSearch(q,e*query.length,lista,M);
            }
            else System.out.print("\nThe search can not be done!\n");

      }

      //    r = e*|query|
      public void rangeSearch(Block query, double r, NodeList lista, int M)
{
            PrintWriter out = null;
            try {
                  out = new PrintWriter(resultadoPesquisaFile);
                  out.write("Arquivo MRS: " + resultadoFile.getName());
                  long tempoInicio = System.currentTimeMillis();
                  out.write("\n-------------Range Query------------");
                  out.write("\nInício Range Query: " + tempoInicio);
                  System.out.println("Range Search...");
                  int i,j,k;
                  double distance[] = new double[M];
                  Node auxPartition;

                  NodeList Res[][] = new NodeList[lista.getT()][d];
                  Wavelet tmp;
```

```
                  for(i=0; i<lista.getT(); i++)
                        for(j=0; j<d; j++)
                              Res[i][j] = null;

                  for(j=0; j<d; j++) //process each sequence
//                      process each MBR
                        for(k=0; k<M; k++) {
                              System.out.println("Sequence " + j + " of " +
d + "- MBR " + k);
                              distance[k] = r;

                              auxPartition = lista.getFirst();
                              int start=0;
                              int end=auxPartition.getValue()-1; //index of
query string
                              for(i = 0; i < lista.getT(); i++) {
                                    int line =
getResolution(auxPartition.getValue());
//                                  if exists the MBR in this resolution
                                    if(k < T[line][j].getLength()) {
                                          FD fd = null;

                                          tmp = new
Wavelet(query.getRange(start,end),ro);
                                          MBR mbr =
bufferManager.requestMBR(line, j, k);
                                          if(mbr == null) {
                                                mbr = T[line][j].getB(k);
                                                BufferNode mbrNode = new
MBRBufferNode(line, j, k, mbr);

      bufferManager.addBuffer(mbrNode);
                                          }

                                          fd  = new FD(tmp, mbr, ro);
                                          if((fd.getDistance() <
distance[k])) {
                                                if(Res[i][j] == null)
                                                      Res[i][j] = new
NodeList();

                                                Res[i][j].insertNode(k);
//store the index of MBR
                                          }

                                          Node auxRes=null;
                                          if(Res[i][j]!=null)

                                                auxRes =
Res[i][j].getFirst();

                                          double max=0;
//                                        refine the distance
                                          while(auxRes!=null) {
                                                fd = new FD(tmp,
T[line][j].getB(auxRes.getValue()), ro);
                                                if((distance[k] -
fd.getDistance()) > max) {
                                                      max = distance[k] -
fd.getDistance();
                                                }
```

```
                                          auxRes = auxRes.getNext();
                                    }
                                    if(max!=0) {
                                          distance[k] = max;

                                    }
                                    fd = null;
                              }

                              auxPartition = auxPartition.getNext();
                              if(auxPartition!=null) {
                                    start = end+1;
                                    end += auxPartition.getValue();
                              }
                        }
                  }

            long tempoFim = System.currentTimeMillis();
            out.write("\nTérmino Range Query: " + tempoFim);
            out.write("\nTempo Gasto: " + (tempoFim - tempoInicio));
            out.write("\nQuery String: ");
            for(char c : query.getBlock()) {
                  out.write(c);
            }
            out.write("\nTamanho da Query: " +
query.getBlock().length);
            out.write("\nErro: " + r / query.getBlock().length);

            List<SearchResult> results = new
ArrayList<SearchResult>();
            int count=0;
            System.out.print(" *** Search Results: *** ");
            out.write("\n *** Search Results: *** ");
            for(j=0; j<d; j++) {
                  SearchResult searchResult = new SearchResult();
                  searchResult.setSubstring(j + 1);
                  System.out.print("(j=" + (j + 1) + " #MBRS=");
                  out.write("\n(j=" + (j + 1) + " #MBRS=");
                  if(Res[lista.getT()-1][j]==null) {
                        System.out.print(0);
                        out.write("" + 0);
                        searchResult.setMbrs(0);
                  }
                  else {
                        System.out.print(Res[lista.getT()-
1][j].getT());
                        out.write("" + Res[lista.getT()-1][j].getT());
                        searchResult.setMbrs(Res[lista.getT()-
1][j].getT());
                        count++;
                  }
                  System.out.print(") ");
                  out.write(") ");
                  results.add(searchResult);
            }
            System.out.print("\n -> Percentage i/o to do: " +
(int)(count*100)/d + "%\n");
            out.write("\n -> Percentage i/o to do: " +
(int)(count*100)/d + "%\n");
            resultado = results;
            porcentagem = (count*100)/d;
```

```
            }
            catch(Exception ex) {
                  ex.printStackTrace();
            }
            finally {
                  if(out != null) {
                        out.flush();
                        out.close();
                  }
            }
      }

      //return the index of window size=queryLength
      private int getResolution(int queryLength) {
            int i=0;
            while(queryLength > (int)Math.pow(2,a)) {
                  queryLength = queryLength/2;
                  i++;
            }
            return i;
      }

      public List<SearchResult> getResultado() {
            return resultado;
      }

      public double getPorcentagem() {
            return porcentagem;
      }
}
```

Classe FD

```
package mrs.model;

public class FD {
      int distance; //Frequency Distance between s1 and s2

      public FD(Wavelet s1, MBR B, int ro) {
            int fd1, fd2;
            fd1 = FD1(s1,B,ro);
            fd2 = FD2(s1,B,ro);
            if(fd1 > fd2)
                  distance = fd1;
            else distance = fd2;
      }

      private int FD1(Wavelet u, MBR v, int ro) {
            int posDistance,negDistance;

            posDistance = negDistance = 0;

            for(int i=0; i<ro; i++) {
                  int uA[] = new int[ro];
                  int vALower[] = new int[ro];
                  int vAHigher[] = new int[ro];

                  uA = u.getA();
                  vALower = v.getLower().getA();
                  vAHigher = v.getHigher().getA();
```

```
                  if(uA[i] < vALower[i])
                        posDistance += (vALower[i] - uA[i]);
                  else if(uA[i] > vAHigher[i])
                        negDistance -= (vAHigher[i] - uA[i]);
            }
            if(posDistance > negDistance)
                  return posDistance;
            return negDistance;
      }

      private int FD2(Wavelet s1, MBR box, int ro) {
            int pos=0,neg=0,min;
            int a1[] = new int[ro];
            int b1[] = new int[ro];
            int boxHighA[] = new int[ro];
            int boxHighB[] = new int[ro];
            int boxLowA[] = new int[ro];
            int boxLowB[] = new int[ro];
            a1 = s1.getA();
            b1 = s1.getB();
            boxHighA = box.getHigher().getA();
            boxHighB = box.getHigher().getB();
            boxLowA = box.getLower().getA();
            boxLowB = box.getLower().getB();
            for(int i=0; i<ro; i++) {
                  if(a1[i] < boxLowA[i])
                        pos += (boxLowA[i] - a1[i]);
                  else if(a1[i] > boxHighA[i])
                        neg -= (boxHighA[i] - a1[i]);

                  if(b1[i] < boxLowB[i])
                        pos += (boxLowB[i] - b1[i]);
                  else if(b1[i] > boxHighB[i])
                        neg -= (boxHighB[i] - b1[i]);
            }
            if(pos < neg)
                  min = pos;
            else min = neg;
            if(min < Math.abs(pos-neg)/2)
                  return (int)Math.ceil((pos-neg)/2);
            else return (int)Math.ceil(Math.abs(pos-neg)/2 + (min-
Math.abs(pos-neg)/2)/2);
      }

      public int getDistance() {
            return distance;
      }
}
```

Classe Tree

```
package mrs.model;

public class Tree {
      private MBR B[];

      //obtain the tree Tij
      public Tree(int resolution, int characters, int box, Block block) {
            //characters      = ro
          //box         = box capacity
            int length = block.getBlock().length;
```

```
            int amountMBR = (int)Math.ceil((double)(length-
resolution+1)/box);

            //Trying to solve negative amount in the last string
            if(amountMBR <= 0) {
                  System.out.println("Length = " + length);
                  System.out.println("Resolution = " + resolution);
                  System.out.println("Amount of MBRs = " + amountMBR);
                  amountMBR = 1;
            }

            int start=0;
            int end=resolution-1; //window size
            Wavelet point;
            point = new Wavelet(block.getRange(start, end), characters);
            int mid = (block.getRange(start, end)).length/2;
            B = new MBR[amountMBR];
            for(int i=0; i<amountMBR; i++) {
                  B[i] = new MBR(block.getRange(start,end), characters);
                  B[i].setStart(start);
                  //slide the window
                  for(int j=0; (j<(box-1)) && (end<(length-1)); j++) {
                        start++;
                        end++;
                        point.updateWavelet(block.getChar(start - 1),
block.getChar(mid), block.getChar(end), resolution);
                        B[i].insertPoint(point); //extend B to cover point
                        mid = (end - start)/2+start+1;
                  }
                  start++;
                  end++;
            }
            //cleaning references
            point = null;
      }
      public int getLength() {
            return B.length;
      }
      public MBR getB(int index) {
            return B[index];
      }
}
```

Classe MBR

```
package mrs.model;

public class MBR {
      private Wavelet lower;  //lower end point (A+B)

      private Wavelet higher; //higher end point (A+B)

      private int start; //start location of the first substring in MBR

      public MBR(char[] range, int ro) {
            lower  = new Wavelet(range,ro);
            higher = new Wavelet(range,ro);
      }
      public Wavelet getLower() {
            return lower;
      }
```

```
public Wavelet getHigher() {
	return higher;
}
public int getStart() {
	return start;
}
public void setStart(int start_location) {
	start = start_location;
}
public void insertPoint(Wavelet point) {
	lower.MBRlow(point.getA(), point.getB());
	higher.MBRhigh(point.getA(), point.getB());
}

@Override
public boolean equals(Object o) {
	if(o instanceof MBR) {
		MBR mbr = (MBR)o;
		Wavelet mbrLower = mbr.getLower();
		Wavelet mbrHigher = mbr.getHigher();
		boolean lowerOk = false;
		boolean higherOk = false;
		if(mbrLower.equals(this.lower)) {
			lowerOk = true;
		}
		if(mbrHigher.equals(this.higher)) {
			higherOk = true;
		}
		if(lowerOk && higherOk) {
			return true;
		}
		else {
			return false;
		}
	}
	else {
		return false;
	}
}

@Override
public int hashCode() {
	int[] lowerA = lower.getA();
	int[] lowerB = lower.getB();
	int[] higherA = higher.getA();
	int[] higherB = higher.getB();
	int soma = 0;
	for(int i = 0; i < lowerA.length; i++) {
		soma = soma + lowerA[i];
	}
	for(int i = 0; i < lowerB.length; i++) {
		soma = soma + lowerB[i];
	}
	for(int i = 0; i < higherA.length; i++) {
		soma = soma + higherA[i];
	}
	for(int i = 0; i < higherB.length; i++) {
		soma = soma + higherB[i];
	}
	return soma;
}
```

```
	@Override
	public String toString() {
		String string = "Higher = " + higher + "\n";
		string = string + "Lower = " + lower + "\n";
		string = string + "Start = " + start + "\n";
		return string;
	}
}
```

Classe Wavelet

```
package mrs.model;

import java.util.Arrays;

public class Wavelet {
	private int A[];

	private int B[];

	public Wavelet(char str[], int characters) {
		//characters == number of integer dimensions
		int secondHalf[] = new int[characters];
		A = new int[characters];
		B = new int[characters];

		if((str.length % 2) == 0) {
			for(int i=0; i<(str.length/2); i++){

				switch(str[i]) {
					case 'A': A[0]++;
							break;
					case 'C': A[1]++;

							break;
					case 'G': A[2]++;
							break;
					case 'N': A[3]++;
							break;
					case 'T': A[4]++;

							break;

				}
			}
			for(int i=str.length/2; i<str.length; i++) {

				switch(str[i]) {
					case 'A': secondHalf[0]++;
							break;
					case 'C': secondHalf[1]++;

							break;
					case 'G': secondHalf[2]++;
							break;
					case 'N': secondHalf[3]++;
							break;
					case 'T': secondHalf[4]++;
```

```
                                    break;

                        }
                  }
                  for(int i=0; i<characters; i++) {
                        B[i] = A[i] - secondHalf[i];
                        A[i] += secondHalf[i];
                  }
            }
      }

      public void MBRlow(int[] lowA, int[] lowB) {
            for(int i=0; i<A.length; i++) {
                  if(lowA[i] < A[i])
                        A[i] = lowA[i];
                  if(lowB[i] < B[i])
                        B[i] = lowB[i];
            }
      }
      public void MBRhigh(int[] highA, int[] highB) {
            for(int i=0; i<A.length; i++) {
                  if(highA[i] > A[i])
                        A[i] = highA[i];
                  if(highB[i] > B[i])
                        B[i] = highB[i];
            }
      }

      @Override
      public boolean equals(Object o) {
            if(o instanceof Wavelet) {
                  Wavelet wavelet = (Wavelet)o;
                  int[] waveletA = wavelet.getA();
                  int[] waveletB = wavelet.getB();
                  boolean aOk = true;
                  boolean bOk = true;
                  for(int i = 0; i < this.A.length; i++) {
                        if(A[i] != waveletA[i]) {
                              aOk = false;
                        }
                  }
                  for(int i = 0; i < this.B.length; i++) {
                        if(B[i] != waveletB[i]) {
                              bOk = false;
                        }
                  }

                  if(aOk && bOk) {
                        return true;
                  }
                  else {
                        return false;
                  }
            }
            else {
                  return false;
            }
      }

      @Override
      public int hashCode() {
```

```
		int soma = 0;
		for(int i = 0; i < A.length; i++) {
			soma = soma + A[i];
		}
		for(int i = 0; i < B.length; i++) {
			soma = soma + B[i];
		}
		return soma;
	}

	@Override
	public String toString() {
		String string = "A = [";
		for(int i = 0; i < A.length - 1; i++) {
			string = string + A[i] + ", ";
		}
		string = string + A[A.length - 1] + "]\n";
		string = string + "B = [";
		for(int i = 0; i < B.length - 1; i++) {
			string = string + B[i] + ", ";
		}
		string = string + B[B.length - 1] + "]\n";
		return string;
	}

	public int[] getA() {
		return A;
	}

	public int[] getB() {
		return B;
	}

	public void updateWavelet(char exclude, char mid, char insert, int
resolution) {
		if(resolution > 1) {
			switch(exclude) {
			case 'A':
				A[0]--;
				B[0]--;
				break;
			case 'C':
				A[1]--;
				B[1]--;
				break;
			case 'G':
				A[2]--;
				B[2]--;
				break;
			case 'T':
				A[3]--;
				B[3]--;
				break;
			case 'N':
				A[4]--;
				B[4]--;
				break;
			}

			switch(mid) {
			case 'A':
```

```
                B[0] += 2;
                break;
        case 'C':
                B[1] += 2;
                break;
        case 'G':
                B[2] += 2;
                break;
        case 'T':
                B[3] += 2;
                break;
        case 'N':
                B[4] += 2;
                break;
        }

        switch(insert) {
        case 'A':
                A[0]++;
                B[0]--;
                break;
        case 'C':
                A[1]++;
                B[1]--;
                break;
        case 'G':
                A[2]++;
                B[2]--;
                break;
        case 'T':
                A[3]++;
                B[3]--;
                break;
        case 'N':
                A[4]++;
                B[4]--;
                break;
        }
}
else {
        switch(exclude) {
        case 'A':
                A[0]--;
                B[0]++;
                break;
        case 'C':
                A[1]--;
                B[1]++;
                break;
        case 'G':
                A[2]--;
                B[2]++;
                break;
        case 'T':
                A[3]--;
                B[3]++;
                break;
        case 'N':
                A[4]--;
                B[4]++;
                break;
```

```
                }

                switch(insert) {
                case 'A':
                      A[0]++;
                      B[0]--;
                      break;
                case 'C':
                      A[1]++;
                      B[1]--;
                      break;
                case 'G':
                      A[2]++;
                      B[2]--;
                      break;
                case 'T':
                      A[3]++;
                      B[3]--;
                      break;
                case 'N':
                      A[4]++;
                      B[4]--;
                      break;
                }
          }
      }

      public void print() {
            System.out.print(Arrays.toString(A));
            System.out.println(" -> " + Arrays.toString(B));
      }
}
```

Classe Manager

```
package mrs.model;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;

import javax.swing.JOptionPane;

public class Manager {
      private int blockSize;

      private int amountOfBlocks;

      private List<FileBlock> fileBlocks = new ArrayList<FileBlock>();

      private long totalLength;

      public Manager(int size, Set<File> files) throws IOException {
            blockSize = size;
            if(!openFiles(files)) {
                  JOptionPane.showMessageDialog(null, "O arquivo do Genoma
está vazio!", "Erro", JOptionPane.ERROR_MESSAGE);
                  throw new IOException();
            }
```

```
	}

	///////////////methods for i/o management//////////////
	public Block readBlock(int blockNumber) throws IOException {
		if(blockNumber < amountOfBlocks) {
			Block b = null;
			b = new Block(blockSize);
			for(FileBlock fileBlock : fileBlocks) {
				if(fileBlock.isInside(blockNumber)) {
					int offset = blockNumber -
fileBlock.getFirstBlock();
					b.readBlock(fileBlock.getRaf(), offset *
blockSize);
					break;
				}
			}
			return b;
		}
		return null;
	}

	private boolean openFiles(Set<File> files) throws IOException {
		boolean filesOk = true;
		amountOfBlocks = 0;
		totalLength = 0;
		int firstBlock = 0;
		for(File file : files) {
			totalLength = totalLength + file.length();
			FileBlock fileBlock = new FileBlock(file, blockSize,
firstBlock);
			firstBlock = fileBlock.getLastBlock();
			System.out.println("Number of Blocks = " +
fileBlock.getNumberOfBlocks());
			amountOfBlocks = amountOfBlocks +
fileBlock.getNumberOfBlocks();
			fileBlocks.add(fileBlock);
		}
		if(fileBlocks.size() <= 0) {
			filesOk = false;
		}
		return filesOk;
	}

	public void closeFiles() throws IOException {
		for(FileBlock fileBlock : fileBlocks) {
			fileBlock.getRaf().close();
		}
	}

	public int getBlockSize() {
		return blockSize;
	}

	public void setBlockSize(int blockSize) {
		this.blockSize = blockSize;
	}

	public int getAmountOfBlocks() {
		return amountOfBlocks;
	}
```

```
	public void setAmountOfBlocks(int amountOfBlocks) {
		this.amountOfBlocks = amountOfBlocks;
	}

	public List<FileBlock> getFileBlocks() {
		return fileBlocks;
	}

	public long getTotalLength() {
		return totalLength;
	}

	public void setTotalLength(long totalLength) {
		this.totalLength = totalLength;
	}
}
```

Classe NodeList

```
package mrs.model;

import java.io.IOException;
import java.io.PrintWriter;

public class NodeList {
	private Node first;

	private int t; //amount of partitions

	public NodeList() {
		first = null;
		t = 0;
	}

	public void insertNode(int value) {
		if(first == null) {
			first = new Node(value);
		}
		else {
			Node tmp = new Node(value);
			tmp.setNext(first);
			first = tmp;
		}
		t++;
	}

	public Node getFirst() {
		return first;
	}

	public int getT() {
		return t;
	}

	public void printList() {
		Node tmp = first;
		System.out.print("\nPrinting the partition list:\n");
		while(tmp!=null) {
			System.out.print(tmp.getValue());
			System.out.print(" - ");
			tmp = tmp.getNext();
```

```
            }
      }

      public void writeList(PrintWriter out) throws IOException {
            Node tmp = first;
            while(tmp!=null) {
                  tmp = tmp.getNext();
                  if (tmp != null) {
                        out.write(" - ");
                  }
            }
      }
}
```

Classe Node

```
package mrs.model;

public class Node {
      private int value; //store the value of partitions size or an index
in the resulting set of MBRs (Range Search)

      private Node next;

      public Node(int a) {
            value = a;
            next = null;
      }
      public void setNext(Node n) {
            next = n;
      }

      public Node getNext() {
            return next;
      }

      public int getValue() {
            return value;
      }
}
```

Classe Block

```
package mrs.model;

import java.io.*;

public class Block {
      private char[] base;

      public Block(int size) { //size = size of buffer´s block
            base = new char[size];
            for(int i=0; i<size; i++)
                  base[i] = ' ';
      }
      //write the block in disc
      public void writeBlock(RandomAccessFile fd, long offset) throws
IOException {
            fd.seek(offset);
            for(int i=0; i<base.length; i++)
                  fd.writeByte((int)base[i]);
```

```
	}

	public void readBlock(RandomAccessFile fd) throws IOException {
		char tmp;
		for(int i = 0; i < base.length; i++) {
			tmp = (char)fd.readByte();
			base[i] = tmp;
		}
	}

	//read the block in disc and save in buffer
	public void readBlock(RandomAccessFile fd, long offset) throws
IOException {
		if((offset + base.length) > fd.length()) {
			fd.seek(offset);
			for(int i = 0; i < (fd.length() - offset); i++) {
				char tmp = (char)fd.readByte();
				base[i] = tmp;
			}
			for(int i = (int)(fd.length() - offset); i < base.length;
i++) {
				base[i] = 'X';
			}
		}
		else {
			char tmp;
			fd.seek(offset);
			for(int i=0; i<base.length; i++) {
				tmp = (char)fd.readByte();
				base[i] = tmp;
			}
		}
	}

	public char[] getBlock() {
		return base;
	}

	public void setBlock(char[] letter) {
		for(int i=0; i<letter.length; i++) {
			base[i] = letter[i];
		}
	}

	//get a range of the block
	public char[] getRange(int start, int end) {
		char temp[] = new char[end-start+1];
		if(!(start<0 || end>=base.length))
			for(int i=start,j=0; i<=end; i++,j++)
				temp[j] = base[i];
		else {
			System.out.print("\nError in method getRange!");
			System.exit(1);
		}
		return temp;
	}
	public char[] getBase() {
		return base;
	}

	public char getChar(int pos) {
```

```
            return base[pos];
      }
}
```

Classe SearchResult

```
package mrs.model;

public class SearchResult {
      private int substring;

      private int mbrs;

      public SearchResult() {
            super();
            substring = 0;
            mbrs = 0;
      }

      public SearchResult(int substring, int MBRs) {
            this.substring = substring;
            this.mbrs = MBRs;
      }

      public int getMbrs() {
            return mbrs;
      }

      public void setMbrs(int mbrs) {
            this.mbrs = mbrs;
      }

      public int getSubstring() {
            return substring;
      }

      public void setSubstring(int substring) {
            this.substring = substring;
      }
}
```

Classe BufferManager

```
package mrs.model.bufferPool;

import java.util.LinkedList;

import mrs.model.MBR;

public class BufferManager {
      private LinkedList<BufferNode> listaDupla = new
LinkedList<BufferNode>();

      private int tamanhoMaximo = 0;

      private BufferNode[] hash;

      private int entradas;

      public BufferManager() {
            tamanhoMaximo = 10;
```

```
		entradas = 10;
		hash = new BufferNode[10];
	}

	public BufferManager(int tamanhoMaximo, int entradas) {
		this.tamanhoMaximo = tamanhoMaximo;
		this.entradas = entradas;
		hash = new BufferNode[entradas];
	}

	public int calcHashCode(int resolution, int sequence, int index) {
		int hashCode = resolution + sequence + index;
		hashCode = (int)((hashCode/177)%entradas);
		return hashCode;
	}

	public int calcHashCode(int somatorio) {
		return (somatorio/177)%entradas;
	}

	public MBR requestMBR(int resolution, int sequence, int index) {
		int hashCode = calcHashCode(resolution, sequence, index);
		if(hash[hashCode] == null) {
			return null;
		}
		else {
			MBRBufferNode node = (MBRBufferNode)hash[hashCode];
			while(node != null) {
				boolean resolutionOk = false;
				boolean sequenceOk = false;
				boolean indexOk = false;
				if(node.getResolution() == resolution) {
					resolutionOk = true;
				}
				if(node.getSequence() == sequence) {
					sequenceOk = true;
				}
				if(node.getIndex() == index) {
					indexOk = true;
				}
				if(resolutionOk && sequenceOk && indexOk) {
					return node.getMbr();
				}
				node = (MBRBufferNode)node.getProximoSimples();
			}
			return null;
		}
	}

	public void removeBuffer(BufferNode bufferNode) {
		int posicao = calcHashCode(bufferNode.hashCode());
		if (hash[posicao].equals(bufferNode)) {
			hash[posicao] = hash[posicao].getProximoSimples();
		}
		else {
			BufferNode bufferAux = hash[posicao];
			while(!bufferAux.getProximoSimples().equals(bufferNode))
{
				bufferAux = bufferAux.getProximoSimples();
			}
```

```
	bufferAux.setProximoSimples(bufferAux.getProximoSimples().getProximoS
imples());
			}
			listaDupla.remove(bufferNode);
	}

	public void addBuffer(BufferNode bufferNode) {
			if(!updateBuffer(bufferNode)) {
				if(!(listaDupla.size() < tamanhoMaximo)) {
					removeBuffer(listaDupla.getLast());
				}
				listaDupla.addFirst(bufferNode);
				int posicao = calcHashCode(bufferNode.hashCode());
				if (hash[posicao] == null) {
					hash[posicao] = bufferNode;
				}
				else {
					bufferNode.setProximoSimples(hash[posicao]);
					hash[posicao] = bufferNode;
				}
			}
	}

	private boolean updateBuffer(BufferNode bufferNode) {
			boolean atualizou = false;
			int posicao = calcHashCode(bufferNode.hashCode());
			if(hash[posicao] == null) {
				return atualizou;
			}
			else {
				BufferNode nodeAux = hash[posicao];
				while(nodeAux != null) {
					if(bufferNode.equals(nodeAux)) {
						listaDupla.remove(bufferNode);
						listaDupla.addFirst(bufferNode);
						atualizou = true;
						return atualizou;
					}
					nodeAux = nodeAux.getProximoSimples();
				}
				return atualizou;
			}
	}

	public void printLRU() {
			for(BufferNode b : listaDupla) {
				System.out.println(b);
			}
	}

	public void printHash() {
			for(int i = 0; i < hash.length; i++) {
				if(hash[i] == null) {
					System.out.println("Entrada " + i + " vazia!");
				}
				else {
					System.out.println("Entrada " + i);
					MBRBufferNode node = (MBRBufferNode)hash[i];
					while(node != null) {
						System.out.println(node);
```

```
                              System.out.println(node.getMbr());
                              node =
(MBRBufferNode)node.getProximoSimples();
                        }
                  }
            }
      }

      public BufferNode getFirst() {
            return listaDupla.getFirst();
      }

      public BufferNode getLast() {
            return listaDupla.getLast();
      }

      public static void main(String[] args) {
            BufferManager bufferManager = new BufferManager(5, 10);

            for(int i = 1; i <= 10; i++) {
                  char[] seq = {'A', 'C', 'G', 'N', 'T'};
                  MBR mbr = new MBR(seq, 5);
                  mbr.setStart(i);
                  MBRBufferNode mbrBufferNode = new MBRBufferNode(i, i, i,
mbr);
                  bufferManager.addBuffer(mbrBufferNode);
            }

            char[] seq = {'A', 'C', 'G', 'N', 'T'};
            MBR mbr = new MBR(seq, 5);
            MBRBufferNode mbrBufferNode = new MBRBufferNode(8, 8, 8, mbr);
            bufferManager.addBuffer(mbrBufferNode);

            MBR mbr1 = new MBR(seq, 5);
            MBRBufferNode mbrBufferNode1 = new MBRBufferNode(7, 7, 7,
mbr1);
            bufferManager.addBuffer(mbrBufferNode1);

            System.out.println();
            bufferManager.printHash();

            System.out.println("MBR request: ");
            System.out.println(bufferManager.requestMBR(6, 6, 6));
      }
}
```

Classe Query

```
package mrs.model;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;

import javax.swing.JOptionPane;

public class Query {
      private Block query;

      private RandomAccessFile fd;
```

```
	public Query(File file) throws IOException {
		int size;
		if(openFile(file)==1) {
			size = (int)fd.length();
			System.out.print("\n---------->Size of query file:" +
size + "\n");
			query = new Block(size);
			query.readBlock(fd,0);
		}
		else JOptionPane.showMessageDialog( null,"Query File is
empty","Error",JOptionPane.YES_OPTION);
		closeFile();
	}

	public char[] getQuery() {
		return query.getBlock();
	}

	/////////////////methods for file control////////////////////
	private int openFile(File file) throws IOException {
		if ( file == null || file.getName().equals( "" ) ) {
			JOptionPane.showMessageDialog( null,"Invalid File
Name","Error",JOptionPane.YES_OPTION);
		}
		else {
			// open file
			fd = new RandomAccessFile(file,"r");
			if(fd.length() > 0)
				return 1;   //the file is not empty
		} // end else
		return 0;               //the file is empty
	} // end method openFile

	public void closeFile() throws IOException {
		fd.close();
	}
}
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo