Relatório do trabalho

**Ordenação de Genomas por Reversões com Sinal**

**[COS886] – Biologia Computacional**

Professoras Celina M. H. de Figueiredo e Marília Dias Vieira Braga.

Grupo 3: Fernando Chirigati, Rafael Dahis, Rafael Lopes e Victor Bursztyn.



Maio de 2011.

Sumário

**Introdução ao Problema de Ordenação por Reversão com Sinal3**

**Restrições Conceituais do Modelo Implementado4**

**A Implementação5**

As Estruturas de Dados5

Uma Visão *Top-Down* do Algoritmo6

Destrinchando o Algoritmo Passo-a-Passo7

A Complexidade Algorítmica10

**Conclusões11**

**Trabalhos Futuros12**

**Referências Bibliográficas13**

**Anexo I: Fontes14**

**Anexo II: Conjunto de Entradas14**

1. **Introdução ao Problema de Ordenação por Reversão com Sinal**

Um dos modelos possíveis para se estudar a evolução dos genomas é avaliá-la sob a luz das permutações de genes. Essa avaliação é feita computando-se a distância entre dois genomas – tal que chamamos esse campo de estudos de “comparação de genomas”.

A ordenação por reversão, nesse contexto, é definida como uma abordagem computacional para a comparação de genomas, baseada na ordem em que os genes aparecem e onde se entende por “distância” o número mínimo de reversões necessárias na transformação de um genoma-origem a um genoma-destino.

A operação de reversão é definida por “transformar uma permutação em outra ao reverter uma porção contígua da mesma, e ao mesmo tempo trocar os sinais dos elementos nela envolvidos.” [1]

Os genomas, por sua vez, são representados de acordo com dois conjuntos inicialmente distintos de arestas: um deles representando os elos a princípio existentes (a eles chamamos de “arestas de realidade”); e um segundo conjunto, a que chamamos de “arestas de desejo”, representando os elos que gostaríamos que existissem.

Esse conceito, em particular, tem uma importância vital para nosso algoritmo. Afinal, a partir das estruturas de dados criadas para se navegar por tais arcos é que muito do trabalho necessário em nossa solução é realizado. Veremos com maiores detalhes as estruturas e os passos que as abordam na seção 3. Implementação, mais adiante.

Vale destacar que, para nosso trabalho, sempre consideramos como genoma-destino – ou *target* – do processo de reversões, a chamada “*identidade”*. Isto é, a forma α em que

l(i)α = i

(E onde todas as orientações são mantidas da esquerda para a direita.)

1. **Restrições Conceituais do Modelo Implementado**

Dada a complexidade do problema tratado e o curto prazo disponível para a realização do trabalho, tornou-se necessário delimitar o escopo de nossa solução. Para isso, acrescentamos à base de entrada uma restrição sugerida pelas professoras Celina e Marília: consideramos que as entradas serão fornecidas sem que haja em cada uma delas, logo a princípio, uma componente ruim.

Vale a ressalva de que nosso algoritmo será capaz de identificar componentes ruins – esse é um dos passos intermediários na resolução do problema e, inclusive, na validação do conjunto de entradas. No entanto, a forma completa da solução, que aceitaria um conjunto de entradas sem restrições, implicaria na construção e no manuseio de estruturas de dados ainda mais complexas, bem como de conceitos que não foram ensinados em aula e que tampouco estão devidamente documentados na referência bibliográfica.

Com isso, essa restrição deve ser tratada com atenção durante os testes e execuções a serem realizados.

1. **A Implementação**

A implementação foi feita com a linguagem de programação C e o resultado, portanto, se encontra na forma procedural. Acreditamos que a abordagem de orientação a objetos talvez fosse mais interessante para o caso de uma solução completa, onde a mesma solução se apoiaria em conceitos e classificações de ciclos mais sofisticados (vide referências [1], [2] e [3]).

* 1. **As Estruturas de Dados**

Temos como principais estruturas globais ao algoritmo:

int n\_input;

O número de genes no input.

int n\_cycles;

O contador de ciclos.

int n\_components;

O contador de componentes.

int \*position;

Um vetor que aponta o índice, na sequência, em que o iésimo elemento aparece (não obstante o sinal).

int \*component\_id;

Armazena o id do componente a que pertence cada ciclo.

int \*n\_reality\_edges;

Armazena quantas arestas de realidade há em cada ciclo.

int \*\*cycle\_id;

É uma matriz [ (n\_input + 2) x 2 ] que armazena o id do ciclo a que as arestas em \*\*reality\_graph pertencem.

int \*\*components;

É um vetor com as boas componentes (a existência de uma má componente é um critério de saída do algoritmo, acusando, neste caso, erro na entrada).

signed int \*sequence;

Um vetor que armazena a sequência propriamente dita.

signed int \*\*reality\_graph;

É uma matriz [ (n\_input + 2) x 2 ] que armazena as arestas de realidade.

signed int \*\*desire\_graph;

É uma matriz [ (n\_input + 2) x 2 ] que armazena as arestas de desejo.

signed int \*\*\*cycles;

Armazena um vetor de ciclos na forma \*\*cycle, que, por sua vez, foram convencionados da seguinte forma:

cycle[i][0] : o primeiro nó da aresta;

cycle[i][1]: o segundo nó da aresta;

cycle[i][2]: a orientação da aresta (1 para da esquerda à direita e -1 para o contrário);

cycle[i][3]: o id do componente a que pertence o ciclo.

Diante das estruturas aqui indicadas, finalmente, vale a seguinte ressalva: optamos por não seguir a forma mais econômica e minimalista de se armazenar os dados do problema de ordenação por reversões com sinal. Ao invés, escolhemos manter estruturas que, se por um lado apresentam alguma redundância, por outro agilizam em muito a manipulação dos dados em determinados passos da solução. Acreditamos que isso é um “*trade-off*” bastante razoável, afinal.

* 1. **Uma Visão *Top-Down* do Algoritmo**
  2. **Destrinchando o Algoritmo Passo-a-Passo**
  3. **A Complexidade Algorítmica**

|  |  |
| --- | --- |
|  | **Complexidade O()** |
| **Complexidade O()** |
| **Complexidade O()** |
| **Complexidade O()** |
| **Complexidade O()**  **Complexidade O()** |
| **Complexidade O()** |
| De onde calculamos nossa complexidade total e final: O() |  |

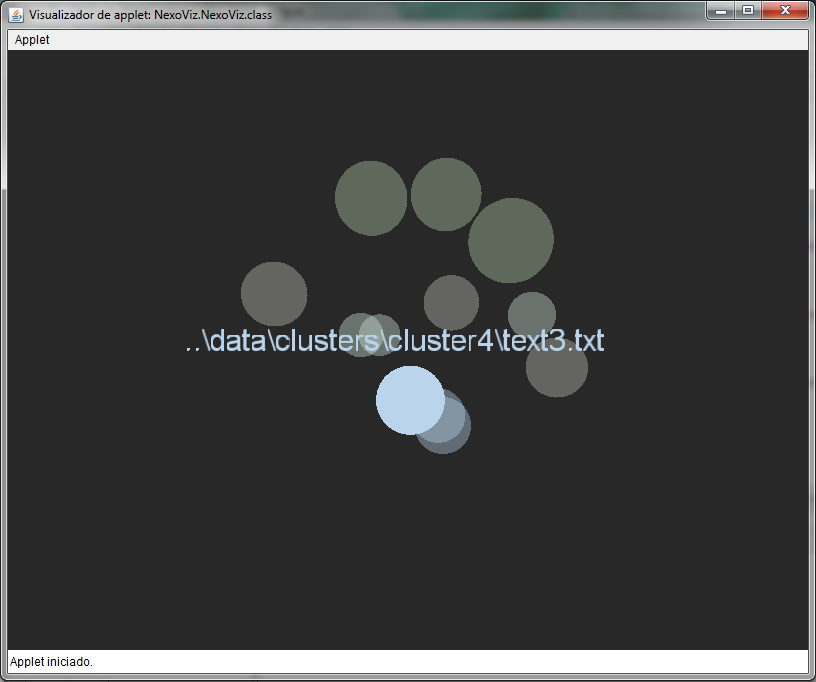
1. **Conclusões**
2. **Trabalhos Futuros**

Enxergamos na solução implementada possibilidades de melhora e, para tanto, mapeamos algumas ações que poderiam ser tomadas futuramente.

Como dito anteriormente, a solução exclui o tratamento de componentes ruins – por se tratar de uma particularidade fora das referências ensinadas e que talvez inviabilizasse a implementação, dado o curto prazo disponível. Dessa forma, uma primeira ação interessante seria a de incorporar a parte que falta do algoritmo, isto é, deixá-lo apto a executar sem restrições de entrada.

Além disso, consideramos uma evolução natural da solução o uso de uma biblioteca mais inteligente do ponto de vista da computação de matrizes. Enxergamos nas matrizes que representam as arestas de desejo e realidade uma grande esparsidade (uma elevada proporção de valores nulos), como costuma acontecer nas estruturas de dados que representam grafos não densos. Entendemos, portanto, que seria inteligente a utilização do formato *Compressed Sparse Rows* de representação de matrizes. Como conseqüência dessa medida, possivelmente o algoritmo seria capaz de lidar com entradas maiores, tenha vista que a redução de uso de memória para as matrizes de adjacência pode chegar a 40% do valor atualmente alocado. Mapeamos como biblioteca para ser empregada neste sentido o PETSc. [1]

Finalmente, uma incorporação bem-vinda – ainda que não seja estritamente necessária – seria a extensão do programa para que se permitam visualizações mais ricas. Entendemos que um dos valores a serem extraídos a partir deste estudo, não perdendo de vista a história da comparação de genomas (vide seção 1. Introdução), seria facilitar justamente a visualização da similaridade entre genomas.

****

1. **Referências Bibliográficas**

[1] **–**

[2] **–**

[3] **–**

1. **Anexo I: Fontes**
2. **Anexo I: Conjunto de Entradas**