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**

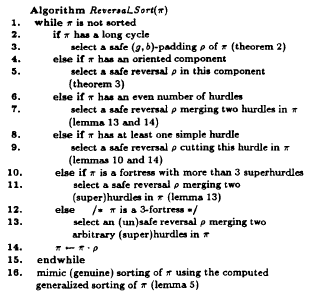
**Anexo II: Conjunto de Entradas14**

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.

Importantíssimos para a área, Hannehalli e Pevner [1] chegaram em 1995 ao seguinte algoritmo de ordenação por reversões:



**(Prosseguir com as explicações)**

1. **Restrições Conceituais do Modelo Implementado**

Dada a complexidade do problema tratado e do 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;

**(Prosseguir com as explicações)**

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;

**(Prosseguir com as explicações)**

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.

* 1. **Uma Visão *Top-Down* do Algoritmo**

**createRealityGraph();**

**createDesireGraph();**

**sortReversal();**

* 1. **Destrinchando o Algoritmo Passo-a-Passo**
  2. **A Complexidade Algorítmica**

1. **Conclusões**
2. **Trabalhos Futuros**
3. **Referências Bibliográficas**

[1] **–**

[2] **–**

[3] **–**

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