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

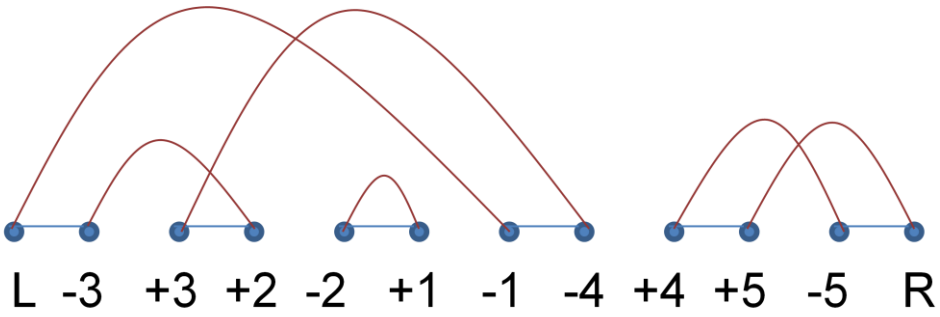
1. Introdução ao Problema de Ordenação por Reversão com Sinal

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

$$I(i)_{\alpha} = i$$

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

2. 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.

3. 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] e [3]).

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.

3.2 Uma Visão *Top-Down* do Algoritmo

Fluxo:

1- Pré-processamento da entrada

2- Criação das arestas de desejo

3- Criação das arestas de realidade

4- Procurar todas as componentes

5- Enquanto número de ciclos != n +1 :

    aplicar reversões, "making sure" que elas não criaram componentes ruins

6- Criar arquivo de saída, salvar output

3.3 Destrinchando o Algoritmo Passo-a-Passo

4- Procurar todas as componentes

• Numerar as arestas de realidade
• Caminhar pelas arestas
    • Encontrando ciclos
    • Definindo componentes (conjunto de ciclos)

• Componente será ruim se não houver ao menos duas arestas divergentes em um de seus ciclos

5- Enquanto número de ciclos != n +1 :
    aplicar reversões, “making sure” que elas não criaram componentes ruins

•Para cada componente, para cada ciclo dentro do componente:
    • se houver arestas divergentes:
        Reverter ! *(inverter essas arestas de realidade)*

• Se essa reversão não criou ciclos ruins: prosseguir.
• Caso contrário: testar outra possível reversão

Construção do DRD: O(n).

encontrar ciclos: O(n).

Determinar se um ciclo é bom ou ruim: O(n). Para todos os ciclos: $O(n^2)$

Selecinar duas arestas divergentes: $O(n^2)$

- Verificar que sua reversão não cria componentes ruins: $O(n^2)$

Logo, os dois itens combinados formam uma complexidade $O(n^4)$.

Devemos fazer essa ordenação até termos n ciclos. Logo, O(n) vezes.

Complexidade final: $O(n^5)$.

4. Conclusões

O algoritmo estudado, de Hannenhalli e Pevner, foi proposto publicamente em 1995 e, portanto, pode ser considerado um pouco obsoleto frente aos avanços que se sucederam, do ponto de vista da complexidade e da facilidade de implementação. No entanto, não seria justo abster os autores do crédito pela inovação que foi semeada com o algoritmo em questão. O algoritmo em si, inclusive, foi melhorado posteriormente pelo próprio Hannenhalli, muito embora tenhamos implementado a versão original.

Dito isso, é preciso destacar o valor didático desta versão original do algoritmo. Ele se baseia, afinal, nos diagrama de realidade e desejo, da maneira mais intuitiva e inteligível possível – até por isso foi a primeira forma pensada e também não-ótima.

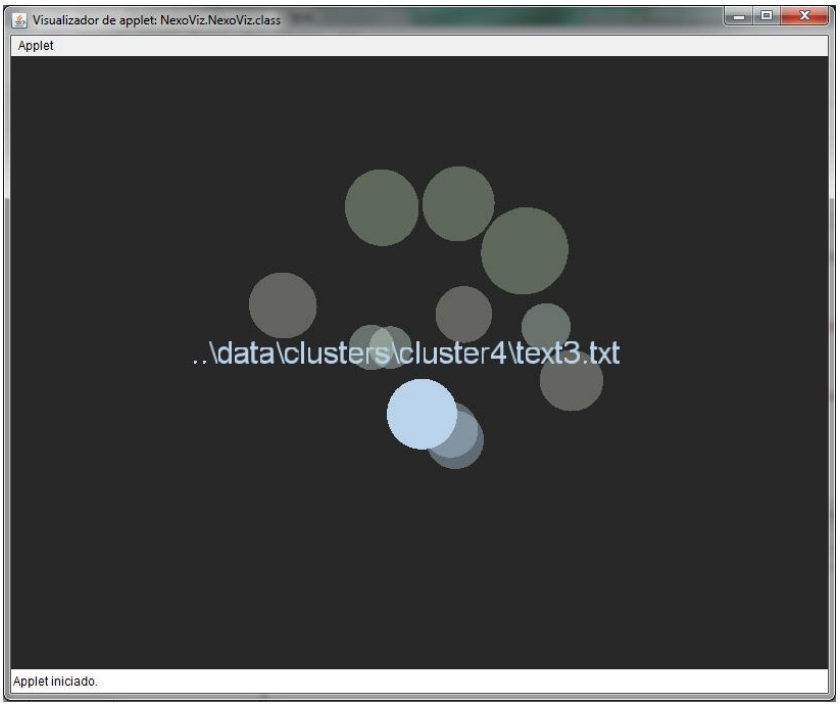
5. Trabalhos Futuros

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. [2]

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.



6. Referências Bibliográficas

[1] – Setubal, João, and Meidanis, João, *introduction to computational biology*, Ed. PWS Publishing Company.

[2] – **http://www.mcs.anl.gov/petsc/** – acessado em 23 de Maio de 2011.

[3] – Tanner E., Bergeron A. and Sagot M-F., *advances on sorting reversals*, Tech. report, Jan 2005.

7. Anexo I: Fontes

8. Anexo I: Conjunto de Entradas