Ingeniería de la Salud
Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

Práctica 5. Reordenación del Genoma.

Arrabalí Cañete, Carmen Lucía

---

1. Introducción y objetivos

Los reordenamientos del genoma describen cambios en la relación de enlace genético de grandes regiones cromosómicas, que implican inversiones, transposiciones, intercambios de bloques, deleciones, inserciones, fusiones y translocaciones, etc.

En este caso, el objetivo principal es estudiar e implementar los algoritmos de ordenación de permutaciones como son los algoritmos `prefix sort`, algoritmo de fuerza bruta, y `break point reversal`, algoritmo voraz.

1.1. Contexto Biológico del problema

Los genes son secuencias de nucleótidos que codifican la síntesis de material bioquímico (ya sea ARN o proteínas). Los procesos biológicos pueden dar lugar a mutaciones en el genoma. Algunas son locales y afectan a un solo nucleótido (sustituciones inserciones, deleciones) mientras que otras pueden ser no locales y afectar a largos tramos de la secuencia (inversiones, transposiciones, translocaciones). Las mutaciones no locales son raras, pero se acumulan a lo largo de la evolutiva. Por lo tanto, son un buen indicador de la distancia evolutiva entre especies.

2. Planteamiento del problema. El genoma como permutación

Se deja que el orden de los genes en un organismo se represente como una permutación con signo. Denominando a las permutaciones como $\pi = [\pi_1\pi_2...\pi_n]$

También hay que tener en cuenta las restricciones en cuanto a la orientación de los genes:

- Si la orientación de los genes no es importante, cada $\pi_i \in \{1, 2, ..., n\}$ es un número entero positivo no repetido.
- Si la orientación del gen es importante, cada π_i puede ser positivo o negativo, y $|\pi_i| \in \{1, 2, ..., n\}$ es un número entero positivo no repetido positivo no repetido.

2.1. Reversal

Una inversión $\rho(i, j)$ es una operación que afecta a la parte del genoma entre las posiciones i y j (ambas inclusive), invirtiendo este segmento.

2.2. Minimal Reversal Distance

Sean π y $\pi^{'}$ dos organismos. La distancia de inversión entre es el número mínimo de inversiones necesarias para transformar una de las permutaciones en la otra. Se puede suponer, por ejemplo, que una de las permutaciones, $\pi^{'}$ es la permutación de identidad positiva, es decir, $\pi^{'}_{i} = 1$, $1 \leqslant i \leqslant n$.

3. Configuración del equipo

Se ha realizado la implementación en un equipo con un sistema operativo Windows 10 Home 64 bits, con un procesador Intel(R) Core (TM) i7-6500U CPU @ 2.50GHz 2.59 GHz (4 CPUs) y un disco SSD de 480GB y 12GB de RAM. La versión Java corresponde a la número 17.

4. Implemetación

4.1. Clase PrefixSort.java

En el método principal, *protected void _run(Permutation l)*, se implementa PrefixSort, el cual se encarga de ordenar la secuencia, ver apartado 4.1.1, recibe como parámetro una permutación, cuya longitud tiene que ser mayor que la unidad. Por otro lado, la permutación como tal no puede ser la identidad (método ya creado) y cuya posición i no puede ser i+1.

Se le aplica el método `Reversal` y se incrementa el número de operaciones totales para así saber cual es el total de permutaciones que realiza el algoritmo.

4.1.1. Método *protected void _run(Permutation l)* de la clase *PrefixSort.java*

```
1  protected void _run(Permutation l) {
2    int n = l.size();
3    for(int i=0; i<n; i++) {
4      if(n > 1 && !l.isIdentity() && l.get(i) != i+1) {
5        int j = l.indexOf(i+1);
6        Reversal p = new Reversal(i, j);
7        p.apply(l);
8        numOperations++;
9      }
10   }
11 }
```

4.2. Clase BreakpointReversalAlgorithm.java

En el primer método de la implementación, *protected List¡SequenceRearrangementOperation¿ getCandidates(Permutation p)* (ver apartado 4.2.1) recibe como parámetro los candidatos posibles que pueden realizar una permutación y son aquellos que entre si mismo y el siguiente valor de la cadena de estudio haya más de 1 de diferencia. En el caso de que no sea así, se considera como no candidato puesto que ya estaría ordenado. En primer lugar se encuentran los puntos de ruptura y luego se evalúan para comprobar que es un candidato factible.

En el segundo método de esta clase, *protected double getOperationQuality(Permutation p, SequenceRearrangementOperation op)*, se trata de buscar, entre todos los candidatos posibles, el que sea de mayor calidad. Esto quiere

decir que, en el caso de tener una permutación que consiga eliminar 2 breakpoints y otra permutación que elimine 1 solo, se cogerá el que consiga eliminar los 2 puesto que será de mayor calidad.

4.2.1. Método *protected List¡SequenceRearrangementOperation¿getCandidates(Permutation p)* de la clase *BreakpointReversalAlgorithm.java*

```
protected List<SequenceRearrangementOperation> getCandidates(Permutation p) {
  int n = p.size();
  List<SequenceRearrangementOperation> ops = new LinkedList<SequenceRearrangementOperation
  >();

  // 1) Find where the breakpoints are.
  int[] breakpoints = new int[n];
  int numBP = 0;
  for (int i = 0; i < n-1 ; i++) {
    if (Math.abs(p.get(i)-p.get(i+1)) > 1) {
      breakpoints[numBP] = i;
      numBP++;
    }
  }

  // 2) Candidates are reversals between breakpoints.
  for (int i = 0; i < numBP; i++) {
    for (int j = i+1; j < numBP; j++) {
      ops.add(new Reversal(breakpoints[i]+1,breakpoints[j]));
    }
  }
  return ops;
}

```

4.2.2. Método *protected double getOperationQuality(Permutation p, SequenceRearrangementOperation op)* de la clase *BreakpointReversalAlgorithm.java*

```
protected double getOperationQuality(Permutation p, SequenceRearrangementOperation op) {
  Reversal r = (Reversal) op;
  int result = 0;

  if (Math.abs(p.get(r.getStart()-1)  - p.get(r.getFinish()) ) == 1) {
    result++;
  }
  if (Math.abs(p.get(r.getFinish()+1)  - p.get(r.getStart()) ) == 1) {
    result++;
  }
  return (double) result;
}

```

5. Resultados y Conclusiones

Para poder analizar correctamente los resultados, el método main recibe en los argumentos de entrada, varios modos de ejecución entre los que se encuentra *−r <algorithm> <init_length> <doubling> <num_test>* el cual genera un archivo con permutaciones aleatorias con distintas longitudes. La longitud inicial de las permutaciones

viene dada por init_length, y se irá aumentando la longitud tantas veces como indique el parámetro doubling. Todas las permutaciones se ordenarán con el algoritmo que se pasa como primer parámetro.

Al utilizar ese modo de ejecución, se genera un archivo con los diferentes tiempos de ejecución, el cual se llama un archivo *PrefixSortrearrangement.txt* y *BreakpointReversalrearrangement.txt*

Por otro lado, entre los recursos aportados en el Campus Virtual se tiene un archivo llamado *analyze.R* que contiene un script el cual se encarga de generar una gráfica para cada algoritmo haciendo uso de los archivos .txt generados anteriormente con el modo de ejecución −r.

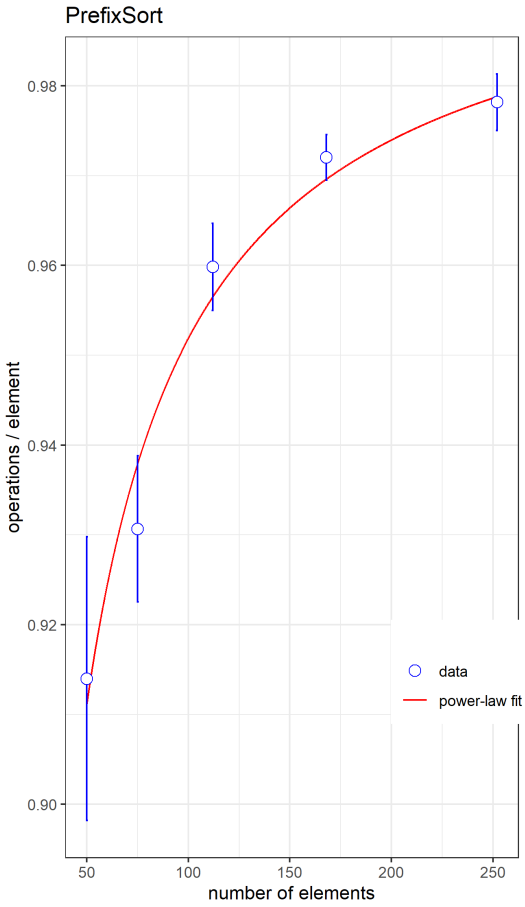

Figura 1: Resultado de la ejecución del algoritmo PrefixSort con los parámetros utilizados.

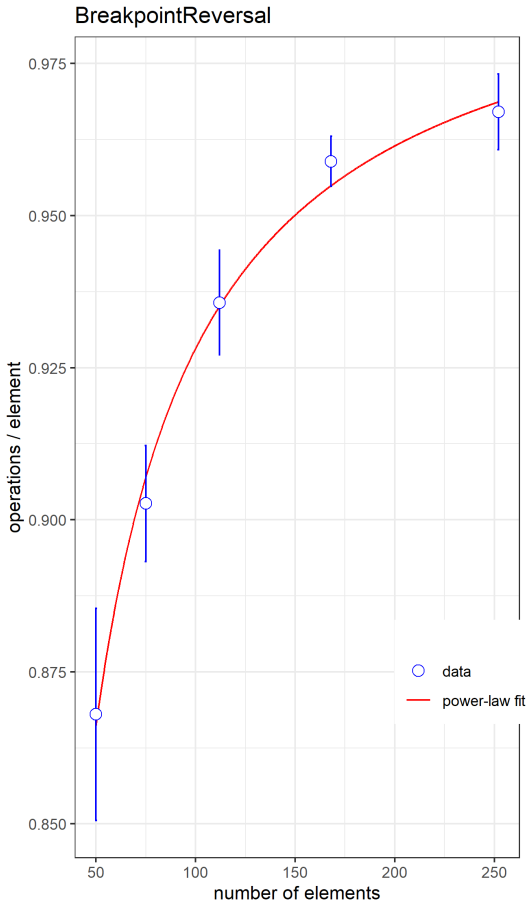

Figura 2: Resultado de la ejecución del algoritmo BreakpointReversal con los parámetros utilizados.

Para poder generar un archivo con unos datos asequibles para el algoritmo y para el equipo que se utiliza, se han dado los siguientes valores:

- <**algoritmo**>: PrefixSort o BreakPointReversal
- <**doubling**>: 5
- <**init_length**>: 50
- <**num_test**>: 10

Como se puede observar en las figuras 1 y 2, cuanto mayor sea el número de elementos de la cadena con la que se quiere obtener el reordenamiento, el número de operaciones que tiene que realizar el algoritmo es mayor, por lo que tiene un coste más elevado.