Teoría de algoritmos 1 – 75.29

Trabajo Práctico Nº: 2

Integrantes:		
Padrón	**Nombre y Apellido**	**Email**
83.989	Alejo Vinjoy	avinjoy@gmail.com
84.623	Santiago Nicolás Risaro Sesar	nrisaro@gmail.com

Para uso de la cátedra
Primera entrega
Corrector
Observaciones
Segunda entrega
Corrector
Observaciones

Contenido

Enunciado

75.29 TEORÍA DE ALGORITMOS

Trabajo Práctico Nº 2

Distancia de edición

Fecha de entrega: 6 de noviembre de 2013

Para convertir una cadena de caracteres x [1..m] en otra cadena y [1..n] , se pueden realizar distintas operaciones. La meta consiste, en dadas las cadenas x e y , encontrar una serie de transformaciones para cambiar x en y , para lo cual debemos emplear un vector z (cuyo tamaño asumimos que es suficiente para almacenar todos los caracteres necesarios) en el que almacenaremos los resultados parciales. Al inicio z está vacío y al final se cumple que $z[j] = y[j]$ *para* $j = 1,2, \ldots, n$. Se deben examinar todos los caracteres de x , para lo cual se mantienen los índices i en x y j en z.

En un principio $i = j = 1$ y al final $i = m + 1$. En cada paso se aplica alguna de las siguientes seis operaciones (transformaciones):

Copiar: copia un carácter de x a z . Esto es: $z[j] = x[i]$ e incrementa los índices i y j
Reemplazar: reemplaza un carácter de x por otro carácter c . Esto es: $z[j] = c$ e incrementa los índices i y j
Borrar: borra un carácter de x incrementando i y sin mover j
Insertar: inserta un carácter c en z . Esto es: $z[j] = c$ e incrementa j sin mover i
Intercambiar: intercambia los próximos dos caracteres copiándolos de x a z pero en orden inverso. Esto es: $z[j] = x[i + 1]$ y $z[j + 1] = x[i]$ e incrementa los índices de la siguiente manera: $i = i + 2$ y $j = j + 2$
Terminar: elimina los caracteres restantes de x haciendo $i = m + 1$. Esta operación descarta todos los caracteres de x que todavía no se analizaron. Es la última operación se aplica si hace falta.

Ejemplo: Se transforma la cadena *algoritmo* en *altruista* usando la siguiente secuencia de operaciones donde los caracteres subrayados corresponden a $x[i]$ y $z[j]$ después de la operación.

Operación	***x***	***z***
Cadenas iniciales	*a lgoritmo*	
copiar	*a l goritmo*	*a*
copiar	*al g oritmo*	*al*
reemplazar por t	*alg o ritmo*	*alt*
borrar	*algo r itmo*	*alt*
copiar	*algor i tmo*	*altr*
insertar u	*algor i tmo*	*altru*
insertar i	*algor i tmo*	*altrui*
insertar s	*algor i tmo*	*altruis*
borrar	*algori t mo*	*altruis*
copiar	*algorit m o*	*altruist*
reemplazar por a	*algoritm o*	*altruista*
terminar	*algoritmo*	*altruista*

La secuencia de operaciones para transformar *algoritmo* en *altruista* no es única.

Cada operación tiene un costo asociado. El costo de cada operación depende de la aplicación específica, pero asumimos que dicho costo es conocido y constante y también asumimos que el costo individual de las operaciones copiar y reemplazar son menores que los costos de las combinaciones de borrar e insertar, ya que de otra manera no convendría nunca usar copiar y reemplazar.

El costo de una determinada secuencia es la suma de los costos individuales de cada operación utilizada en la secuencia. En el ejemplo anterior:

4.costo (*copiar*)+ 2.costo (*reemplazar*)+2.costo (*borrar*)+3.costo (*insertar*)+*costo* (*terminar*)

Se pide:

1. Dadas dos cadenas *x* [1..m] e *y* [1..n] y el costo de cada una de las operaciones. Escribir un programa (programación dinámica) que calcule la *distancia de edición* siendo esta la secuencia de menor costo que permita transformar *x* en *y* . Analizar la complejidad en tiempo y espacio de la solución implementada.
2. Explicar que como utilizar el programa de distancia de edición del punto 1 utilizando un subconjunto de las operaciones copiar, reemplazar, borrar, insertar, intercambiar y terminar para resolver el problema de la alineación de secuencias de ADN con el método dado.

Estrategia de Resolución

La estrategia de resolución se divide en dos partes. Por un lado existe un módulo que se encarga de cargar los costos asociados e inicializar el módulo de resolución.

A su vez el módulo de resolución contiene los pasos para el algoritmo de programación dinámica, que, previa inicialización, va a iterar guardando en cada celda el costo de operación, la celda anterior al paso en el que nos encontramos y el tipo de operación que realizó. De esta forma además de obtener la distancia de edición, obtenemos la serie de pasos realizados para llegar a esa distancia.

Como último paso, se vuelve al inicio desde el final hacia atrás, obteniendo las columnas y filas de las celdas que se guardan en una pila. Esa pila a su vez se vacía para obtener la serie de operaciones en una forma ordenada.

Análisis de Orden

Comentado en los métodos del código, sin embargo hacemos una mención breve al método principal del cálculo distancia de edición (calcularDistanciaEdicion) tiene una complejidad del orden de O (m x n) + O(n) donde m es la longitud de la palabra inicial y n es la longitud de la palabra final.

El costo O(m x n) viene de cargar la matriz, el O(n) surge de aplicar las operaciones para transformar una palabra en la otra.

Adicionalmente tenemos el costo de leer el archivo de costos, esto será tan costoso como líneas tenga el archivo.

Indicaciones para la ejecución

Se asume que en la PC donde va a ejecutarse el programa se encuentra instalada y correctamente configurada una versión del JRE. Este trabajo práctico fue desarrollado utilizando la versión 1.6 de java y testeado con el JRE oficial, provisto por Oracle. Si bien no fue probado con otra configuración, el trabajo debería ejecutarse sin problemas con máquinas virtuales versión 1.7 y/o no oficiales. Debido a los cambios en la API entre la versión 1.5 y 1.6 de java no hay garantías de que pueda ejecutarse correctamente con versiones anteriores a 1.6.

Para la compilación se utilizó Maven 3.

Se asume que existe una carpeta $HOME/tda y que en ella se han descargado los archivos a utilizar.

Ejecución

1. Posicionarse en el directorio donde se encuentra el archivo tdatp2.jar

```
cd $HOME/tda
```

2. Ejecutar el programa

```
java -jar tdatp2.jar <INICIO> <FIN> [<ARCHIVO>]
```

<INICIO> Es la palabra de la que se parte.

<FIN> Es la palabra a la que se quiere llegar

<ARCHIVO> es un parámetro opcional con la ruta del archivo que contiene la definición de los costos. Si no se provee este parámetro el programa buscará un archivo llamado “costos.txt” en la carpeta $HOME/tda

Compilación

1. Posicionarse en la carpeta donde se descargará el código fuente

```
cd $HOME/tda
```

2. Descargar una copia del código fuente

```
svn checkout
http://teoria-de-algoritmos-2013-2.googlecode.com/svn/tags/Entrega_2_2013_11_6/TeoriaDeAlgoritmos
```

3. Poscionarse en la carpeta recién descargada

```
cd TeoriaDeAlgoritmos
```

4. Ejecutar maven

```
mvn clean package -Ptp2
```

5. El ejecutable se encuentra en la carpeta target

Alineación de ADN

Para este caso no logramos obtener una configuración de los datos de entrada que diera como resultado la salida esperada sin alterar el algoritmo específicamente para estos casos.

Lo que pudimos hacer es definir una serie de costos para las operaciones (costosADN.txt) que priorizan las operaciones de modo tal que, realizando una traducción entre las operaciones y los costos de la alineación, tal como están especificados en el enunciado, podemos llegar al resultado esperado.

La traducción mencionada es la siguiente:

La operación **Reemplazar** equivale a **-**

La operación **Copiar** equivale a **+**

Las operaciones **Insertar** y **Borrar** equivalen a *****

De esta manera, si ejecutamos el TP con las secuencias del enunciado obtenemos una serie de operaciones (Reemplazar, Copiar, Insertar y Borrar) que, traducidas con la referencia anterior nos dan un costo de -4, tal como lo indica el enunciado, pero el costo de edición al que llegamos no es -4.

Conclusiones

Hemos visto de forma práctica el concepto de programación dinámica que consiste en dividir un problema grande en una serie de subproblemas que se pueden resolver mejor y encontrar una solución óptima. Esa solución óptima se basa en el estado actual y no en los estados anteriores. Mediante el uso de una tabla se puede evitar resolver los subproblemas más de una vez.

Comprobamos además que el orden temporal es en general de O(n x m). Tanto el orden temporal como espacial variará a partir de la longitud de las palabras a las que le aplicamos la distancia de edición.

Respecto a la distancia con el A.D.N., no hemos encontrado una combinación satisfactoria de costos que diera el mismo resultado sin sacrificar el algoritmo original. Pero hemos logrado una solución "de compromiso" que nos acerca al resultado esperado.

Código Fuente

El código fuente completo puede ser accedido en
http://code.google.com/p/teoria-de-algoritmos-2013-2/source/browse/#svn%2Ftags%2FEntrega_2_2013_11_6%2FTeoriaDeAlgoritmos

CostoOperacion.java

```
public class CostoOperacion {

   public enum TipoOperacion{

         COPIAR(1) {

               @Override

               public void ejecutar(String palabraInicio, String palabraFin,

                           StringBuilder palabraResultante, int i, int j) {

                     palabraResultante.setCharAt(j, palabraInicio.charAt(i));

                     System.out.println("Copiar " + palabraInicio.charAt(i));

               }

               @Override

               public int nuevoI(int i) {

                     return i + 1;

               }

               @Override

               public int nuevoJ(int j) {

                     return j + 1;

               }

         },

         REEMPLAZAR(1) {

               @Override
```

```
		public void ejecutar(String palabraInicio, String palabraFin,
				StringBuilder palabraResultante, int i, int j) {
			palabraResultante.setCharAt(j, palabraFin.charAt(j));
			System.out.println("Reemplazar " + palabraInicio.charAt(i) + " " +
palabraFin.charAt(j));
		}
		@Override
		public int nuevoI(int i) {
			return i + 1;
		}
		@Override
		public int nuevoJ(int j) {
			return j + 1;
		}
	},
	BORRAR(1) {
		@Override
		public void ejecutar(String palabraInicio, String palabraFin, StringBuilder
palabraResultante, int i, int j) {
			System.out.println("Borrar " + palabraInicio.charAt(i));
		}
		@Override
		public int nuevoI(int i) {
			return i + 1;
		}
		@Override
```

```
		public int nuevoJ(int j) {
			return j;
		}
	},
	INSERTAR(1) {
		@Override
		public void ejecutar(String palabraInicio, String palabraFin, StringBuilder
palabraResultante, int i, int j) {
			palabraResultante.setCharAt(j, palabraFin.charAt(j));
			System.out.println("Insertar " + palabraFin.charAt(j));
		}
		@Override
		public int nuevoI(int i) {
			return i;
		}
		@Override
		public int nuevoJ(int j) {
			return j + 1;
		}
	},
	TERMINAR(1) {
		@Override
		public void ejecutar(String palabraInicio, String palabraFin, StringBuilder
palabraResultante, int i, int j) {
			System.out.println("Terminar");
		}
```

```
		@Override
		public int nuevoI(int i) {
			return i + 1;
		}
		@Override
		public int nuevoJ(int j) {
			return j;
		}
	},
	INTERCAMBIAR(1) {
		@Override
		public void ejecutar(String palabraInicio, String palabraFin, StringBuilder
palabraResultante, int i, int j) {
			palabraResultante.setCharAt(j, palabraInicio.charAt(i + 1));
			palabraResultante.setCharAt(j + 1, palabraInicio.charAt(i));
			System.out.println("Intercambiar " + palabraInicio.charAt(i) + " " +
palabraInicio.charAt(i + 1));
		}
		@Override
		public int nuevoI(int i) {
			return i + 2;
		}
		@Override
		public int nuevoJ(int j) {
			return j + 2;
		}
```

```
        };

        private Integer costo;

        /**

         * Aplica la operación sobre la palabra resultante tomando los datos de la palabra de inicio o
de fin.

         * Realiza operaciones considerando los strings como arrays, dado que cuenta con los
subíndices para

         * acceder a cada posición de cada string el costo es O(1).

         */

        public abstract void ejecutar(String palabraInicio, String palabraFin, StringBuilder
palabraResultante, int i, int j);

        public abstract int nuevoI(int i);

        public abstract int nuevoJ(int j);

    }

    private int costo;

    private int filaAnterior;

    private int colAnterior;

    private TipoOperacion op;

}
```

CargadorCostos.java

```
public class CargadorCostos {

   /**
    * Lee el archivo linea a linea y guarda en memoria el costo de cada operación
    * El costo de ejecutar esta operación es proporcional a la cantidad de lineas
    * del archivo O(L)
    */
   public void cargar(BufferedReader reader) throws IOException {
         String linea = reader.readLine();
         while(linea != null) {
               String[] lineaCosto = linea.split(":");
               String operacion = lineaCosto[0].trim();
               String costo = lineaCosto[1].trim();
               TipoOperacion.valueOf(operacion.toUpperCase()).setCosto(Integer.valueOf(costo));
               linea = reader.readLine();
         }
   }
}
```

Distancia.java

```
public class Distancia {
   private final DistanciaEdicion distanciaEd;
   /**
    * Carga los costos O(L) y ejecuta el programa O(mxn + n)
    */
   public static void main(String[] args) throws IOException {
            System.out.println("Teoría de algoritmos - TP 2");
            System.out.println("Autores:");
            System.out.println("Alejo Vinjoy - 83.989");
            System.out.println("Santiago Nicolas Risaro Sesar - 84.623");
            if (args.length > 0) {
                     String palabraInicial= args[0];
                     String palabraFinal = args[1];
                     BufferedReader archivo = leerArchivo(args);
                     CargadorCostos cargador = new CargadorCostos();
                     cargador.cargar(archivo);
                     DistanciaEdicion distanciaEdicion = new DistanciaEdicion(palabraInicial,palabraFinal);
                     new Distancia(distanciaEdicion).ejecutar();
            } else {
                     System.err.println("Se debe ingresar la palabra inicial, la palabra final y el nombre
del archivo de costos");
                     System.err.println("O solo las palabras inciales y finales");
                     System.err.println("i.e.: java -jar Distancia.jar Algoritmo Altruista");
            }
```

```
    }
    private static BufferedReader leerArchivo(String[] args) throws FileNotFoundException {
            String rutaArchivo = "costos.txt";
            if (args.length > 2) {
                    rutaArchivo = args[2];
            }
            return new BufferedReader(new FileReader(rutaArchivo));
    }
    public void ejecutar() throws IOException {
            int costo = distanciaEd.calcularDistanciaEdicion();
            System.out.println();
            System.out.println("Distancia de Edicion: "+costo);
    }
}
```

DistanciaEdicion.java

```
public class DistanciaEdicion {
   private CostoOperacion[][] distance;
   private String palabraInicio;
   private String palabraFin;
   private Stack<CostoOperacion> resultado;
   private int costoFinal;
   private String palabraResultante;
   /**
    * Implementación de http://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance
    * Construye una matriz de mxn, siendo m la cantidad de letras de la palabra de inicio
    * y n la cantidad de letras de la palabra a la que se quiere llegar.
    * Luego itera por cada celda de la matriz decidiendo cual es la operación óptima, esta
    * decisión tiene costo O(1), por lo que el costo de esta parte es O(mxn).
    * Por último debe aplicar las operaciones que detectó como óptimas, esto será tan costos
    * como la palabra resultante O(n).
    * El costo total es O(mxn + n)
    */
   public int calcularDistanciaEdicion() {
            CostoOperacion casoBase;
            int i, j;
            cargarPrimeraColumna();
            cargarPrimeraFila();
            for(i = 1; i < distance.length; i++) {
                     for (j = 1; j < distance[i].length; j++) {
```

```
                        casoBase = casoBase(i, j);

                        CostoOperacion eraseCost = new CostoOperacion(i-1, j,
distance[i-1][j].getCosto() + BORRAR.getCosto(), BORRAR);

                        CostoOperacion insertCost=new CostoOperacion(i, j-1,
distance[i][j-1].getCosto() + INSERTAR.getCosto(), INSERTAR);

                        CostoOperacion menorCosto = minimoCosto(insertCost, eraseCost,
casoBase);

                        distance[i][j]=menorCosto;

                        if(sePuedeIntercambiar(i, j)) {

                                intercambiar(i, j, menorCosto);

                        }

                }

        }

        costoFinal = distance[palabraInicio.length()][palabraFin.length()].getCosto();

        recolectarOperaciones();

        aplicarOperaciones();

        return costoFinal;

  }

  private void cargarPrimeraColumna() {

        for (int k = 0; k < distance.length; k++) {

                distance[k][0] = new CostoOperacion((k-1 > 0)? k-1:0,0,(k-1 >= 0)?
k*BORRAR.getCosto():0,BORRAR);

        }

  }

  private void cargarPrimeraFila() {

        for (int k = 0; k < distance[0].length; k++) {

                distance[0][k] = new CostoOperacion(0,(k-1 > 0)? k-1:0,(k-1 >= 0)?
```

```
k*INSERTAR.getCosto():0,INSERTAR);
            }
    }
    private CostoOperacion casoBase(int i, int j) {
            CostoOperacion casoBase;
            if (palabraInicio.charAt(i - 1) == palabraFin.charAt(j - 1)) {
                    casoBase = new
CostoOperacion(i-1,j-1,distance[i-1][j-1].getCosto()+COPIAR.getCosto(),COPIAR);
            } else {
                    casoBase = new
CostoOperacion(i-1,j-1,distance[i-1][j-1].getCosto()+REEMPLAZAR.getCosto(),REEMPLAZAR);
            }
            return casoBase;
    }
    private CostoOperacion minimoCosto(CostoOperacion a, CostoOperacion b, CostoOperacion c) {
            int costoA= a.getCosto();
            int costoB= b.getCosto();
            int costoC= c.getCosto();
            if (costoA <= costoB && costoA <= costoC)
                    return a;
            if (costoB <= costoA && costoB <= costoC)
                    return b;
            return c;
    }
    private boolean sePuedeIntercambiar(int i, int j) {
            return i > 1 && j > 1 &&
```

```
            palabraInicio.charAt(i - 1) == palabraFin.charAt(j - 2) &&
            palabraInicio.charAt(i - 2) == palabraFin.charAt(j - 1);
    }
    private void intercambiar(int i, int j, CostoOperacion menorCosto) {
            CostoOperacion swapCost=new CostoOperacion(i-2,j-2,distance[i-2][j-2].getCosto() +
INTERCAMBIAR.getCosto(),INTERCAMBIAR);
            distance[i][j] = (distance[i][j].getCosto() >  swapCost.getCosto()) ? swapCost : menorCosto;
    }
    /**
     * Guardo el resultado de las operaciones en una pila antes de devolver el costo, recorriendo la
matriz.
     */
    private void recolectarOperaciones() {
            int fila=palabraInicio.length();
            int col=palabraFin.length();
            do {
                    CostoOperacion cOp= distance[fila][col];
                    resultado.push(cOp);
                    fila=cOp.getFilaAnterior();
                    col=cOp.getColAnterior();
            }
            while (fila > 0 || col > 0);
    }
    private void aplicarOperaciones() {
            int i = 0;
            int j = 0;
```

```
        StringBuilder palabraResultante = inicializarResultado();
        boolean terminado = false;
        while (!resultado.isEmpty() && !terminado) {
                CostoOperacion operacion = resultado.pop();
                TipoOperacion comando = operacion.getOp();
                comando.ejecutar(palabraInicio, palabraFin, palabraResultante, i, j);
                i = comando.nuevoI(i);
                j = comando.nuevoJ(j);
                terminado = verificarSiTermina(i, j, palabraResultante);
        }
        this.palabraResultante = palabraResultante.toString().trim();
}
private StringBuilder inicializarResultado() {
        char[] chars;
        if (palabraFin.length() >= palabraInicio.length()) {
                chars = new char[palabraFin.length()];
        } else {
                chars = new char[palabraInicio.length()];
        }
        Arrays.fill(chars, ' ');
        return new StringBuilder(new String(chars));
}
private boolean verificarSiTermina(int i, int j, StringBuilder palabraResultante) {
        boolean terminado = false;
        if (palabraResultante.toString().trim().equals(palabraFin)) {
```

```
                int diferencia = palabraInicio.length() - palabraResultante.toString().trim().length();
                if (diferencia > 0) {
                        int costoBorrar = diferencia * BORRAR.getCosto();
                        if (TERMINAR.getCosto() <= costoBorrar) {
                                TERMINAR.ejecutar(palabraInicio, palabraFin, palabraResultante, i, j);
                                costoFinal -= costoBorrar;
                                costoFinal += BORRAR.getCosto();
                                terminado = true;
                        }
                }
        }
        return terminado;
    }
}
```