Teoría de algoritmos 1 - 75.29

Trabajo Práctico N°: 2

Integrantes:

Padrón	Nombre y Apellido	Email
92216	Alejandro Pernín	ale.pernin@gmail.com

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

Indice

Enunciado

Distancia de edición

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 , ... , 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.

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.
El programa debe tomar como parámetros las dos cadenas y el nombre de un archivo con el costo de las operaciones. El formato de cada línea de este archivo es el siguiente:
<operación>: <costo>
Ejemplo de invocación:
tdatp2 algoritmo altruista costos.txt
La salida debe ser por pantalla y debe mostrar una línea por cada operación, indicando los caracteres que se insertan o reemplazan. Una línea en blanco y el costo total de

la secuencia.

El problema de la distancia de edición, tal como está planteado en este TP generaliza el problema de alinear dos secuencias de ADN. Existen varios métodos para medir la similitud entre dos secuencias de ADN alineándolas. Uno de esos métodos consiste en insertar espacios en posiciones arbitrarias en ambas secuencias (incluyendo al final) tal que las secuencias resultantes tengan la misma longitud pero que no tengan espacios en la misma posición. Entonces se le asigna un puntaje a cada posición de la siguiente manera:

• +1 si x ' [j]=y ' [j] y ninguno es un espacio
• −1 si x ' [j]≠y ' [j] y ninguno es un espacio
• −2 si x ' [j]o y ' [j]es un espacio

El puntaje total de la alineación es la suma de los puntaje de cada posición. Por ejemplo dadas las secuencias x=GATCGGCAT e y=CAATGTGAATC , una alineación posible es:
G A T C G G C A T
C A A T G T G A A T C
- * + + * + * + - + + *

Donde + indica un puntaje de +1 en esa posición, - indica un puntaje de -1 y * indica un puntaje de -2. Tal que el puntaje total de la alineación es $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$
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

Para resolver el problema planteado se tuvo en cuenta las siguientes hipótesis:

- El costo individual de copiar y reemplazar, son menores que los costos de las combinaciones de borrar e insertar.
- Conozco soluciones parciales anteriores del problema.
- El problema es case sensitive, por lo cual mayúsculas y minúsculas serán diferenciadas.

El problema se atacó desde un punto de vista dinámico utilizando el enfoque de memorización, es decir que se van guardando en memoria soluciones previas a un problema menor, cuya solución considero es óptima.

Para ello se utilizó un diccionario que irá alojando las soluciones previas para tenerlas disponibles para ensamblar soluciones futuras. El concepto de solución es:

- Solución(x)=Solución(x-1)+solucionador(x)

Es decir que la solución hasta un carácter se compone de la solución hasta el carácter anterior más una solución para ese único carácter.

Si al consultar el diccionario la solución no se encuentra, se calcula.

Para analizar cuál es una solución para cada carácter se analizan diversas opciones:

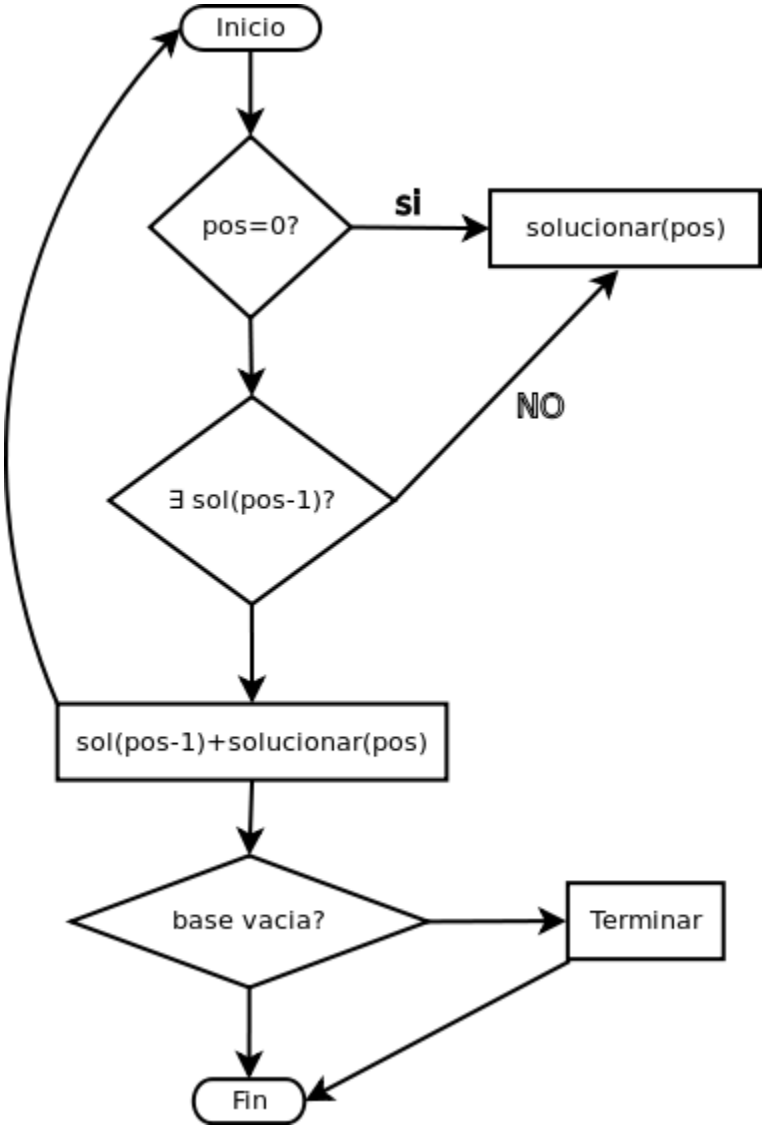
- Si no hay más elementos en la base, no hay otra opción más que insertar un carácter.
- Se analizan las diferentes posibilidades para obtener un carácter y se elije acorde a cuál tiene un menor costo.
- En caso de que se efectue un intercambio, se aloja en dos posiciones del diccionario para que en la consulta recursiva, no se vuelva a calcular la obtención de un carácter que ya se obtuvo mediante el intercambio

Para resolver el problema de la alineación de ADNs utilizando un subconjunto de las operaciones ya definidas se plantean las siguientes opciones:

- Con un subprograma realizar las inserciones de espacios requeridas.
- Modificar las operaciones para incluir los espacios en la toma de decisiones (abajo se explicará)
- Adecuar los costos a los puntajes requeridos.

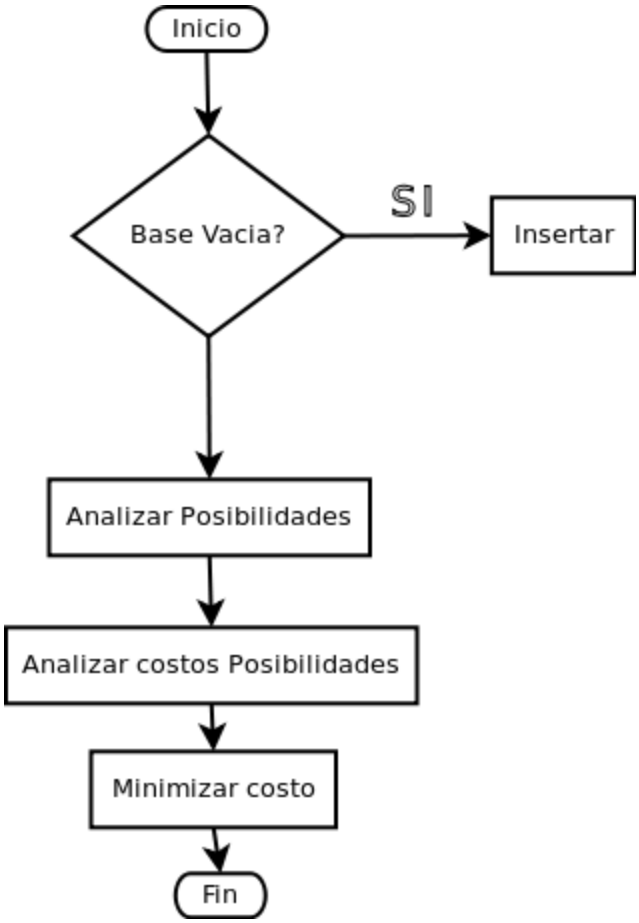
De esta manera alternando las posiciones en las cuales se insertan los espacios, se puede hallar la alineación máxima.
El programa en esta instancia reconoce los espacios como caracteres, pero no los incluye en la toma de decisiones. Por ejemplo si con ‘h la’ queremos obtener ‘h la’ (dependiendo de los costos), se van a copiar todos los caracteres. Por lo que no hace diferencia de si es una letra o un espacio, situación que a efectos de la alineación es necesario discriminar.

Diagramas de flujo:
Función Solucion:



Para verlo más grande: http://aleperno.com.ar/public/tda/diagrama1.png

Funcion Solucionar:



Para verlo más grande: http://aleperno.com.ar/public/tda/diagrama2.png

Análisis de orden

A continuación se hará un análisis de los órdenes de las principales funciones, se omiten las que no revisten complejidad (se encuentran los ordenes descriptos también en el código).

- Solution(): Una cota es O(base * objetivo), sin embargo este caso representa en el cual para cada letra a obtener del objetivo, se requiere hacer una análisis de todas las letras disponibles en la base. Se puede decir que en una ejecución normal, el orden es menor.
- Solve(): O(base), este es el caso de que se requiera hacer un análisis de todos los caracteres incluidos en la base para dar con una solución. De otro es O(1).

Compilación y ejecución

Al ser Python un lenguaje interpretado, no requiere compilación, directamente se ejecuta el programa mediante la llamada al intérprete con los parámetros necesarios.

La línea de ejecución se indica a continuación:

python tdatp2 <palabra base> <palabra objetivo> costos.txt

o

./tdatp2 <palabra base> <palabra objetivo> costos.txt

También se incluye un script en bash que ejecuta diferentes pruebas, para ejecutarlas simplemente mediante consola ejecutar

./tests.sh

Conclusiones

Se diseñó e implementó un algoritmo para analizar las operaciones necesarias para transformar una palabra en otra, con un análisis de costos.

Se utilizó estructuras que permitan almacenar en memoria las soluciones parciales óptimas para ser utilizadas a futuro; en cada construcción de soluciones se analizaron todas las posibilidades (dentro de las hipótesis) y sus respectivos costos, eligiendo aquella que sea de menor costo.

El programa resultante en el peor de los casos trabaja en un orden O(base * objetivo), pero en casos muy particulares; por lo que se podría afirmar que en un caso cotidiano con una disposición coherente de costos (que afectan la decisión) y las palabras a utilizar, el algoritmo trabaja en un orden menor.

Asimismo bajo este esquema, es fácilmente escalable el algoritmo a otro problema en el cual con una misma base, se analicen las transformaciones a más de una palabra, ya que poseemos soluciones de los subproblemas

Código fuente

Tda2tp.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys

class Cost(object):
    """Se utiliza el patron singleton que fue modificado
    para que no se llame a init en cada llamado a la instancia"""

    _instance = None
    _init = False

    """O(1)"""
    def __new__(cls,*args, **kwargs):
        if not cls._instance:
            cls._instance = super(Cost, cls).__new__(cls)
        return cls._instance

    """O(1)"""
    def __init__(self, file_source = None):
        if self._init :      return  #Avoids __init__ being called on every access.
        self.dict = {}
        if file_source is not None:
            self.load_file(file_source)
        self._init = True

    """O(#Operaciones) en este caso O(6)"""
    def load_file(self, file_source):
        #print "el archivo se llama %s \n" % file_source
        _file = open(file_source)
        for line in _file:
            line = line.replace('\n','').split(':')
            self.dict[line[0]]=int(line[1])

    """O(1)"""
    def costo(self,op):
        return self.dict[op]

"""TODAS LAS OPERACIONES SON O(1)"""
class Copiar():
    def __init__(self,char):
        self.char = char
```

```
    def __str__(self):
        s = "copiar %s" % self.char
        return s

class Reemplazar():
    def __init__(self,char1,char2):
        self.char1=char1
        self.char2=char2

    def __str__(self):
        s = "reemplazar %s %s" %(self.char1,self.char2)
        return s

class Borrar():
    def __init__(self,char):
        self.char=char

    def __str__(self):
        s = "borrar %s" % self.char
        return s

class Insertar():
    def __init__(self,char):
        self.char=char

    def __str__(self):
        s = "insertar %s" % self.char
        return s

class Intercambiar():
    def __init__(self,char1,char2):
        self.char1=char1
        self.char2=char2

    def __str__(self):
        s = "intercambiar %s %s" % (self.char1 , self.char2)

        return s

class Terminar():
    def __str__(self):
        return "terminar"


class Problem():
    """O(1)"""
    def __init__(self,pal1,pal2):
        self.base = pal1
```

```
        self.posbase = 0
        self.objective = pal2
        self.res=[]
        self.mem={}
        self.cost=0

    """O(1)"""
    def eob(self):
        return self.posbase == len(self.base)

    """O(1)"""
    def verBase(self):
        return self.base[self.posbase]

    """O(1)"""
    def verSigBase(self):
        try:
            return self.base[self.posbase+1]
        except IndexError:
            return None

    """O(1)"""
    def verObj(self,pos):
        return self.objective[pos]

    """O(1)"""
    def verSigObj(self,pos):
        try:
            return self.objective[pos+1]
        except IndexError:
            return None

    """O(1)"""
    def copiar(self):
        c = Copiar(self.verBase())
        self.posbase += 1
        self.cost += Cost().costo('copiar')
        return [c]

    """O(1)"""
    def insertar(self,pos):
        char = self.objective[pos]
        i = Insertar(char)
        self.cost += Cost().costo('insertar')
        return [i]

    """O(1)"""
    def terminar(self):
```

```
        t = Terminar()
        self.cost += Cost().costo('terminar')
        return [t]

    """O(1)"""
    def intercambiar(self):
        x = self.verBase()
        y = self.verSigBase()
        i = Intercambiar(x,y)
        self.cost += Cost().costo('intercambiar')
        self.posbase += 2
        return [i]

    """O(1)"""
    def reemplazar(self,pos):
        r = Reemplazar(self.verBase(),self.verObj(pos))
        self.cost += Cost().costo('reemplazar')
        self.posbase += 1
        return [r]

    """O(1)"""
    def borrar(self):
        b = Borrar(self.verBase())
        self.posbase += 1
        self.cost += Cost().costo('borrar')
        return [b]

    """O(1)"""
    def checkintercambio(self,pos):
        """Evalua si es viable un intercambio"""
        if pos < len(self.objective):
            """Evaluo caso de intercambio"""
            b1 = self.verBase()
            b2= self.verSigBase()
            o1 = self.objective[pos]
            o2 = self.verSigObj(pos)
            if (not b2 is None and not o2 is None) and (b1 == o2) and (b2 == o1):
                return True
        return False

    """
    El peor caso es que me encuentre al principio del string y que no haya ningun
    elemento cuya copia sea posible por lo que debe recorrer todo el string base
    O(#base)
    """
    def distcopy(self,pos):
        """Mide la distancia al proximo elemento que se pueda copiar"""
        dist = 0
```

```
        aux = self.posbase #Guardo la posicion original
        while not self.eob():
            if self.verBase()==self.verObj(pos):
                dist = self.posbase - aux
            self.posbase += 1
        self.posbase = aux #Vuelvo a colocarlo en su posicion original
        return dist

    """
    Idem anterior, el peor caso es que no haya ningun elemento cuyo intercambio
    sea posible.
    O(#base)
    """
    def distinter(self,pos):
        """Mide la distancia al proximo intercambio(de ser posible)"""
        dist = 0
        aux = self.posbase
        while not self.eob():
            if self.checkintercambio(pos):
                dist = self.posbase - aux
            self.posbase +=1
        self.posbase = aux
        return dist

    """
    Mide el minimo en un conjunto de operaciones
    O(#l)
    En esta implementacion podemos decir que es:
    O(2)=O(1)
    """
    def min(self,l):
        minimo = None
        for i in l:
            if (i[0]==0):
                continue
            if (minimo is None) or (i[0]*Cost().costo(i[1]) < Cost().costo(minimo)):
                minimo = i[1]
        return minimo

    """
    Idem anterior
    O(#l)
    A efectos de esta implementacion:
    O(3)=O(1)
    """
    def min2(self,l):
        minimo = None
        costo = None
```

```
        for i in l:
            if i[0] is None:
                continue
            if (minimo is None) or (i[0] < costo):
                minimo = i[1]
                costo = i[0]
        return minimo

    """
    Devuelve el costo total de una lista de operaciones
    O(#l)
    A efectos de esta implementacion
    O(1)
    """
    def opcost(self,l):
        aux = 0
        for i in l:
            cost = (i[0]*Cost().costo(i[1]))
            aux += cost
        return aux

    """
    Determina cual es la mejor manera de obtener el caracter actual
    con los elementos disponibles en la palabra base
    O(#Base)
    """
    def solve(self,pos,aux=None):
        r = []
        if self.eob():
            """No hay otra opcion más que insertar, ya que se agotaron
            los caracteres disponibles en la base"""
            return self.insertar(pos)   #O(1)

        #O(1)
        if self.verBase() == self.objective[pos]:
            """Es el caso de copiar"""
            aux = self.checkintercambio(pos) #O(1)
            """Evaluo si en vez de copiar se puede intercambiar y si es mas optimo"""
            #O(1)
            if not aux or (aux and self.min([(1,'copiar'),(1,'intercambiar')])=='copiar'):
                """Evaluo si en vez de copiar se puede reemplazar"""
                #O(1)
                if (self.min([(1,'copiar'),(1,'reemplazar')])=='copiar'):
                    return self.copiar() #O(1)
                else:
                    """No tiene sentido que reemplazar sea mas eficiente, igual se implementa"""
                    return self.reemplazar(pos) #O(1)
```

```
#O(1)
if pos < len(self.objective):
    """Evaluo caso de intercambio"""
    if self.checkintercambio(pos):
        r = self.intercambiar()  #O(1)
        """
        Se guarda en dos posiciones de memoria dado que al intercambiar
        estamos solucionando dos posiciones del objetivo
        """
        try:
            self.mem[pos+1]=self.mem[pos-1]+r   #O(1)
        except KeyError:
            if (aux is not None):
                self.mem[pos+1]=aux+r     #O(1)
            else:
                pass
        return r

"""En este punto tengo que evaluar borrar, insertar o reemplazar"""

d = None
c = None
d1 = self.distinter(pos)
if d1>0:
    """Distancia cero implica que no es posible la operacion"""
    c1 = self.opcost([(d1,'borrar'),(1,'intercambiar')])  #O(2)=O(1)
    d = d1
    c = c1

d2 = self.distcopy(pos)
if d2>0:
    c2 = self.opcost([(d2,'borrar'),(1,'copiar')])   #O(2)=O(1)
    if (c2 < c) or (c is None):
        c = c2
        d = d2

c3 = self.opcost([(1,'insertar')]) #O(1)
c4 = self.opcost([(1,'reemplazar')]) #O(1)

op = self.min2([(c,'borrar'),(c3,'insertar'),(c4,'reemplazar')]) #O(3)=O(1)

"""
El peor caso seria que estemos en el principio y que debamos copiar un elemento
que se encuentre al final de la palabra base
O(#base)
"""
if op == 'borrar':
    for i in range(0,d):       #O(d)
```

```
                r += self.borrar()        #O(1)
            r += self.solve(pos,r)        #O(1)
            """
            A pesar de ser recursivo, siendo que ya analizamos que vamos a copiar/intercambiar
            por lo que en la llamada recursiva es seguro que no llegue a este punto y
            devuelva una solucion lineal
            """
            return r
        elif op == 'insertar':
            return self.insertar(pos) #O(1)
        else:
            return self.reemplazar(pos) #O(1)


    """
    Se emplea programacion dinamica usando memorización.
    Se hacen tantas llamadas a la funcion como caracteres a obtener de allí
    Siendo que el peor de los casos de solve es O(#base):
    O(#base * #Objetivo)
    """
    def solution(self,pos):
        """Como premisa supongo que ya poseo como conseguir una solucion anterior
        """
        if pos == 0:
            return self.solve(pos)

        if not self.mem.has_key(pos-1):
            self.mem[pos-1] = self.solution(pos-1)
        if not self.mem.has_key(pos):
            self.mem[pos] = self.mem[pos-1] + self.solve(pos)
        if pos == len(self.objective)-1 and not self.eob():
            """Ya obtuvimos la solucion y aun hay elementos en la base que deben ser
            descartados"""
            self.mem[pos] += self.terminar()
        return self.mem[pos]

def checkArguments():
    return True




def main():
    print "Teoria y Algoritmos 1 - [75.29]"
    print "TP2 - Distancia de Edicion"
    print "Autores: Alejandro Pernin (92216)\n"
    s1 = Cost(file_source=sys.argv[3])
```

```
        pal = sys.argv[1]
        pal2 = sys.argv[2]
        p = Problem(pal,pal2)
        s = p.solution(len(pal2)-1)
        for i in s:
                print i
        print "El costo es: %s" % str(p.cost)
if __name__ == '__main__':
        main()
```