Proyecto 3 Algoritmos y complejidad

Diego Iturra, Nicolás Herrera, Aníbal Polanco

July 2022

1 Parte 1: Alineación de secuencias global y algoritmo de Needleman-Wunsch.

(a) La heurística de la recurrencia es la siguiente, donde se toma el paso anterior y se maximisa el valor de T(n) en base al camino a tomar.
$T(n) = T(n-1) + máx(path)$

(b) La solución top-down con memorizacion la realizamos en la primera parte del problema.
Ocupamos memorizacion para hacer la matriz de puntaje casilla a casilla y guardamos en esta misma el valor obtenido en cada celda, para hacer la celda $(i+1, j+1) \vee (i, j+1) \vee (i+1, j)$, se tienen en cuenta las posiciones adjacentes ya puestas en la matriz (diagonal, superior o lateral) para elegir de donde viene y asignar el puntaje correcto.

(c) Una vez obtenida la tabla de puntajes, todavía tenemos el problema de obtener el mejor camino para asi obtener el mejor match entre secuencias.

Adicionalmente creamos una matriz del largo de ambas secuencias, y asignamos el puntaje para la celda[i][j] de 1 o -1, acorde a si $seq1[i] = seq2[j] \vee seq1[i] \neq seq2[j]$

Haciendo uso de la tabla de puntajes, y de la matriz de matches entre las secuencias, podemos ocupar el enfoque bottom up para finalmente retornar las secuencias de mayor puntaje: Partimos desde la última celda de la matriz de puntajes y desde esta vamos "escalando" en la matriz hacia la celda anterior con mayor puntaje y sumamos a ambas secuencias de salida el caracter correspondiente, ya sea la letra de la secuencia o el gap, dependiendo del camino tomado.

(d) Para realizar el problema, se implementearon 3 funciones principales:
Sean $n = len(seq1)$ & $m = len(seq2)$

1) needlemanWunsch(scoreMatrix, seq1, seq2)
Este algoritmo consiste de un for(0..n), un for(0..m) y un doble for (0..n)(for(0..m)) usados para rellenar la matriz de puntajes de dimenciones $(n + 1) * (m + 1)$. Vale la pena mencionar que este algoritmo fue programado dinamicamente usando el enfoque Top-Down.
Espacialmente la complejidad es $(n + 1)(m + 1) \Leftrightarrow O(n * m)$.
Temporalmente la complejidad es la misma, ya que se realiza un número constante (4) de operaciones por cada casilla visitada, por lo tanto la complejidad temporal también es $O(n * m)$.

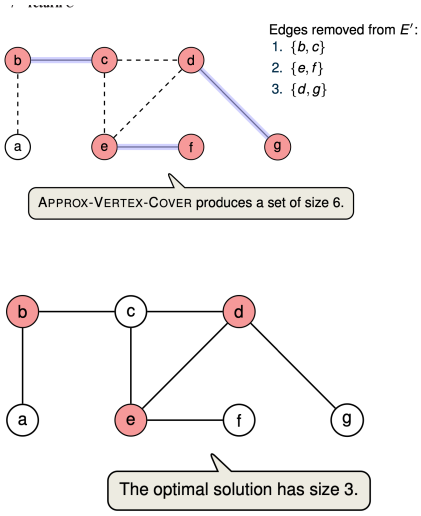
2) fillMatchMatrix(matchMatrix, seq1, seq2)
Esta función rellena la matriz matchMatrix de $n*m$, con 1 o -1 acorde a la igualdad de los caracteres de seq1[i] y seq2[j].
Espacialmente la complejidad es $O(n * m)$.
Temporalmente la complejidad también es $O(n * m)$.

3) getAlignmentSolution(scoreMatrix, matchMatrix, seq1, seq2)
Este algoritmo hace uso de la scoreMatrix creada con needlemanWunsch(), de la matchMatrix creada con fillMatchMatrix() y del enfoque de programación dinamica bottom up haciendo uso de la scoreMatrix.
El algoritmo consiste en un $while(i > 0 \& j > 0)$ en el que se recorre el camino con más puntaje desde el final de la matriz de puntajes hasta que se llega a uno de los bordes (i o j igual a 0).
En el peor de los casos la complejidad de este algoritmo es de $O(n + m - 1)$, y en el mejor es $O(n)$. Por lo tanto la complejidad temporal es $O(n + m)$.
La complejidad espacial, asumiendo $m \leq n$, es $O(2n)$ dado que se rellenan dos secuencias de largo máximo n.

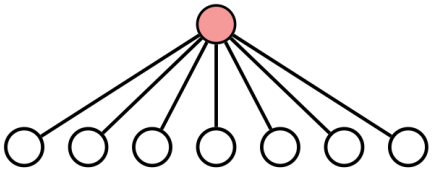
Finalmente, teniendo en cuenta lo descrito en a, b y c, podemos decir que la complejidad temporal total de nuestra implementacion es:

$$\Theta(2 * m * n + (m + n))$$

2 Parte 2: Minimun vertex cover.



(a) El grafo mostrado más arriba corresponde al grafo usado de ejemplo en la implementacion correspondiente a esta parte del proyecto. La solucion obtenida por nuestra implementación del algoritmo corresponde a la primera imagen y es 2-aproximado.
El grafo anterior fue obtenido de `https://www.cl.cam.ac.uk/teaching/1920/AdvAlgo/vertexcover_handout.pdf`



Este último grafo corresponde a uno que nos entrega un aproximado igual al óptimo.

(b) Sea maximaCoincidencia(clique,VerticesDelGrafo) una función que retoruna un int correspondiente a la cantidad de coincidencias entre un vertice en clique y en VerticesDelGrafo.

```
V ←Set de Vertices del grafo
Cliques ←Lista de cliques maximales del grafo
SC ←Lista vacía
while V.notEmpty() do
    maxMatch ← 0
    cliqueToAdd ← Vacío
    for c in Cliques do
        if (maximaCoincidencia(c,V) <maxC) then
            maxC ← maximaCoincidencia(c,V)
            cliqueToAdd ← c
        end if
    end for
    SC.add(cliqueToAdd)
    S.remove(cliqueToAdd)
    V.remove(allVertexIn cliqueToAdd)
end while
```

El algoritmo descrito tiene como objetivo entregar una lista mínima de los cliques que juntos equivalen a todos los vértices del grafo dado, o sea busca entregar el setcover del grafo dado una lista de cliques maximales. Corresponde a una implementación en pseudocódigo al algoritmo greedy visto en clases.

2(b) continúa en la siguiente página.

Teniendo la lista de mínimos cliques maximales que describen todos los vértices del grafo (SC), planteamos el siguiente algoritmo.
La heurística descrita en el siguiente pseudocódigo busca cliques en la lista SC de tal manera que para dos cliques dentro de SC distintos se encuentre una arista entre un nodo perteneciente a Clique1 y un nodo perteneciente a Clique2 que forme parte grafo original.
La arista encontrada se agrega al vertex cover de salida y se eliminan los cliques de una copia de SC usada para darle término al algoritmo.
Una vez la copia de SC esté vacía, se tiene que todos los cliques maximales estan representados en la lista de vertex cover de salida y al mismo tiempo, existen aristas que conectan estos cliques que involucran a los vertices entregados en la lista de vertex cover.

```
finalVertexCover ← empty list
SCCopy ← SC
G ← GrafoOriginal
for Clique1 in SC do
    for Clique2 in SC do
        if Clique ≠ Clique2 then
            for a in Clique1 do
                for b in Clique2 do
                    if (a,b) in G or (b,a) in G then
                        finalVertexCover.add(a).add(b)
                        break
                    end if
                end for
            end for
            if Clique1 in SCCopy then
                SCCopy.remove(Clique1)
            end if
            if Clique2 in SCCopy then
                SCCopy.remove(Clique2)
            end if
            if SCCopy.empty then
                return finalVertexCover.set()
            end if
        end if
    end for
end for
```

La relación entre este algoritmo y el greedy implementado en 2(a) es que este último toma aristas del grafo original, y el descrito en este pseudocódigo toma aristas en el grafo original en las que sus nodos forman parte de cliques maximales distintos.