DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Número 2

16 de Mayo de 2016 Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Ciruelos Rodríguez, Gonzalo	063/14	gonzalo.ciruelos@gmail.com
Costa, Manuel José Joaquín	035/14	manucos94@gmail.com
Gatti, Mathias Nicolás	477/14	mathigatti@gmail.com
Maddonni, Axel Ezequiel	200/14	axel.maddonni@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Una Nueva Esperanza **4**

1.1. Explicación formal del problema . . . 4

1.2. Explicación de la solución . . . 5

1.2.1. Construcción de G' . . . 5

1.2.2. Correctitud y optimalidad . . . 7

1.2.3. Explicación del código . . . 8

1.3. Complejidad del algoritmo . . . 11

1.4. Performance del algoritmo . . . 12

1.4.1. Método de experimentación . . . 16

2. El Imperio Contraataca **17**

2.1. Explicación formal del problema . . . 17

2.2. Explicación de la solución . . . 18

2.2.1. Explicación del código . . . 18

2.2.2. Pseudocódigo . . . 20

2.2.3. Correctitud y Optimalidad . . . 20

2.3. Complejidad del algoritmo . . . 21

2.3.1. Mejor Caso . . . 22

2.4. Performance del algoritmo . . . 23

2.4.1. Método de experimentación . . . 27

3. El Retorno del ~~que te~~ Jedi **28**

3.1. Explicación formal del problema . . . 28

3.1.1. Ejemplos . . . 28

3.2. Formulación Recursiva . . . 29

3.2.1. Demostración de Correctitud . . . 29

3.3. Pseudocódigo . . . 30

3.4. Complejidad del algoritmo . . . 32

3.5. Performance del algoritmo . . . 32

3.5.1. Método de experimentación . . . 33

4. Apéndice **34**

4.1. Generación de grafos conexos aleatorios . . . 34

4.2. Partes relevantes del código . . . 35

4.2.1. problema1.cpp 35
4.2.2. problema2.h 36
4.2.3. problema2.cpp 37
4.2.4. problema3.cpp 40

1. Una Nueva Esperanza

1.1. Explicación formal del problema

Sea $G = (V, E)$ un grafo simple conexo con $V = \{v_0, v_1, \ldots, v_{n-1}\}$, $n \geq 2$, y m aristas. Además sea $M \subseteq E$ tal que $M \neq \emptyset$. Se desea hallar un camino (no necesariamente simple) de v_0 a v_{n-1} que pase al menos dos veces por un eje de M (potencialmente el mismo) y que tenga longitud mínima.

En la figura (1) pueden verse tres ejemplos. Los tres son muy parecidos pero permiten ilustrar distintas situaciones. En el primero (de izquierda a derecha y de arriba a abajo) encontramos el camino simple $P = (v_0, v_1, v_3, v_4, v_5)$ de longitud 4 que cumple con lo pedido pues tiene dos aristas especiales y es de longitud mínima.

En el segundo se agregó una arista corriente entre los nodos v_0 y v_5. Puede notarse que en este caso el mejor camino es $P = (v_0, v_2, v_0, v_5)$ de longitud 3, el cual claramente no es simple pues pasa dos veces por el vértice v_0.

En el último caso, la arista recientemente agregada pasa a ser especial. Al hacer esto tenemos dos caminos de longitud mínima entre los que pasan por dos aristas especiales: el P que encontramos antes, y $P' = (v_0, v_5, v_0, v_5)$. Notar que P' no solo no es simple, sino que también usa a v_5 como nodo intermedio. Cualquiera de los dos es igualmente aceptable.

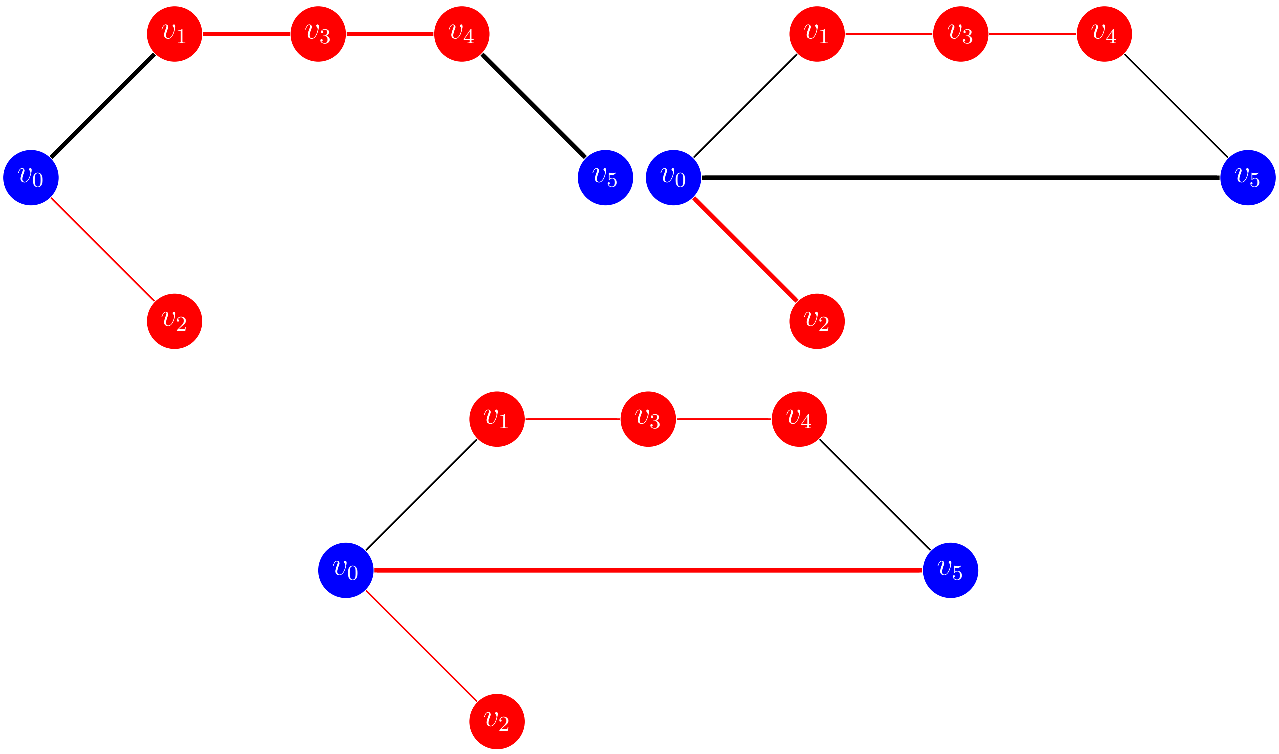


Figura 1: Ejemplos del problema. Las aristas especiales están pintadas de rojo y las comunes de negro. Las aristas que pertenecen a la solución están engrosadas. En azul distinguimos a los nodos inicial y final.

1.2. Explicación de la solución

Si bien lo que se pide es, en definitiva, encontrar un camino mínimo en G, la dificultad adicional que implica hacer que el camino contenga al menos dos aristas de M hace que no podamos aplicar de forma directa los algoritmos clásicos para este propósito. Para solventar esto consideraremos un grafo alternativo a G, G', con el cual resolver el problema planteado originalmente será equivalente a encontrar un camino mínimo en G' de forma tradicional (en este caso utilizando BFS).

1.2.1. Construcción de G'

Lo primero que haremos es armarnos un grafo nuevo G' a partir de G.

Consideramos tres grafos isomorfos a G: $G_0 = (V_0, E_0)$, $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$, tales que $f_k : V \to V_k / f_k(v_i) = v_{k \times n + i}$ para $k \in \{0, 1, 2\}$ son las biyecciones correspondientes. En particular, $G_1 = G$. La figura (2) ilustra la situación para un ejemplo puntual.

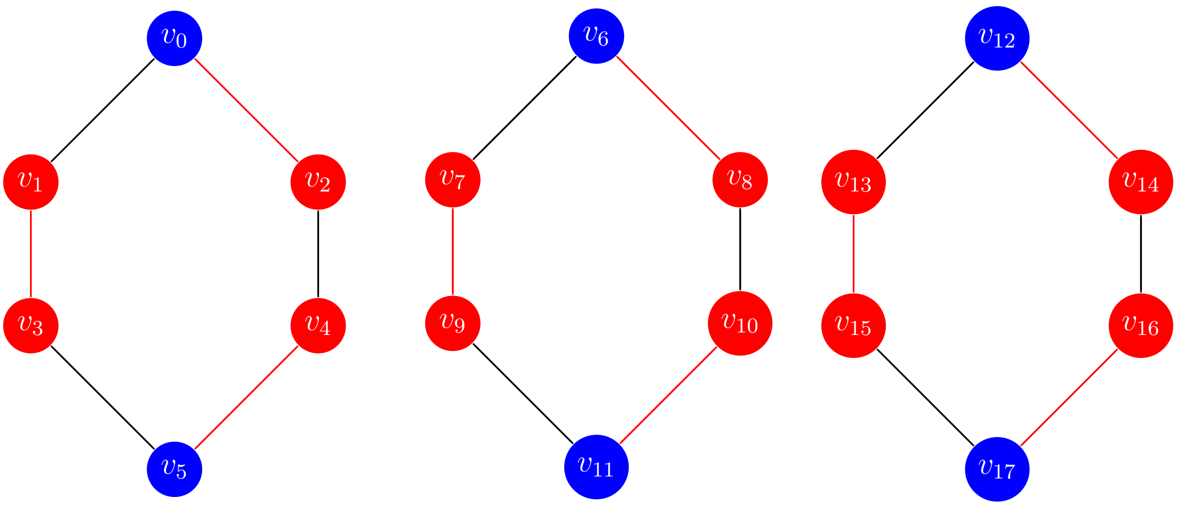


Figura 2: Tres isomorfismos del grafo original, construidos de la forma indicada.

Además, definimos M', un conjunto de arcos (aristas orientadas) de peso 1, como

$$M' = \{(f_0(v), f_1(w)) \text{ y } (f_0(w), f_1(v)) / (v, w) \in M\} \cup \{(f_1(v), f_2(w)) \text{ y } (f_1(w), f_2(v)) / (v, w) \in M\}$$

Por ejemplo, en la figura (2) la arista $(v_0, v_2) \in M$. Entonces queremos que M' tenga los arcos (v_0, v_8), (v_2, v_6), (v_6, v_{14}) y (v_8, v_{12}). Observar que en M' solo hay arcos que van de nodos de G_0 a nodos de G_1, y de G_1 a G_2. En ningún caso hay arcos de G_t a G_h, con $h < t$; ni arcos que vayan directamente de G_0 a G_2.

Entonces hasta acá tenemos tres grafos conexos isomorfos. Podemos pensarlos como las tres componentes conexas de un grafo con $3n$ vértices y $3m$ aristas. A continuación uniremos estas tres componentes mediante los arcos de M': sea $G^* = (V_1 \cup V_2 \cup V_3, E_1 \cup E_2 \cup E_3 \cup M')$.

Podemos pensar a G^* con la siguiente analogía: cada una de las tres componentes es un nivel, y los arcos "escaleras mecánicas" que permiten subir de un nivel al siguiente en forma unidireccional y que no se saltea niveles. En este contexto, G_k será el nivel k. Notar que entonces G^* no es fuertemente conexo pues desde un nodo del nivel 2 o 3 no puedo alcanzar a

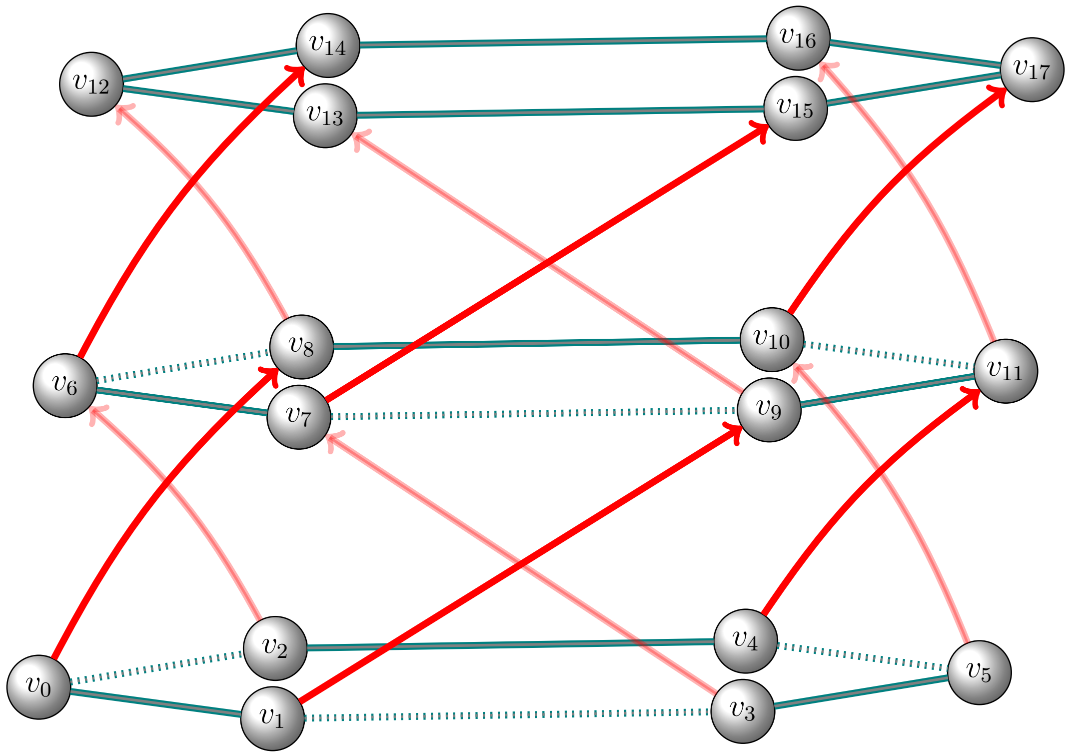


Figura 3: G' para el ejemplo dado en la figura (2). Las líneas punteadas señalan las aristas que estaban en G^* pero que quitamos.

un nodo en el nivel 1. Sin embargo, para cualquier vértice en el nivel 1 todos los vértices del grafo son alcanzables. En particular esto vale para v_0.

Finalmente, definimos a G' como el resultado de quitarle a los niveles 0 y 1 de G^* todas las "aristas isomorfas" a las aristas de M.[1]

En este caso ya no es cierto que cualquier vértice de G' sea alcanzable desde cualquier vértice del primer nivel: en efecto, si vemos la figura (3), el nodo v_2 no es alcanzable desde el v_0. No obstante, sigue valiendo el siguiente lema:

Lema 1.1: Todo vértice del tercer nivel (nivel 2) de G' es alcanzable desde todo vértice del primer nivel (nivel 0).

Demostración: Sean $u, w \in V_1$ (es decir, ambos del nivel 0). Supongamos que w no es alcanzable desde u en G'. Como G_1 inicialmente era conexo, esto significa que existía camino de u a w, Q, y seguro incluía alguna arista especial (sino seguiría existiendo en G' y w sería alcanzable desde u). Como todos los nodos que eran incidentes a una arista especial en G_1 son incidentes a un arco en G', seguro existe $z \in Q$, tal que z es incidente a un arco que permite pasar al segundo nivel. Luego, es posible llegar desde u a algún vértice del segundo nivel. Si no existe tal nodo w, entonces cualquier nodo del primer nivel es alcanzable desde u, y como M no era vacío, entonces seguro hay un camino desde u hasta el segundo nivel.

[1]En rigor, tanto G' como G^* sirven a nuestro propósito, pero G' tiene la ventaja de que permitirá reducir el uso de memoria (potencialmente mucho si hay muchas aristas especiales) y constantes en la complejidad de la implementación.

Es fácil ver que exactamente el mismo razonamiento se puede realizar para probar que es posible llegar desde cualquier nodo del segundo nivel a algún nodo del tercer nivel. Luego, por concatenación de ambas cosas, es posible llegar desde cualquier nodo del primer nivel a algún nodo del tercero. Pero como el tercer nivel sigue siendo conexo (pues nunca quitamos las aristas especiales) entonces es claro que esto es equivalente a poder llegar a cualquier nodo del tercer nivel. $\square$

Debido a este lema, y como todas las aristas de G' tienen peso 1, es posible aplicar el algoritmo BFS para hallar el camino mínimo entre un nodo del nivel 0 y otro del 2. Particularmente, entre los nodos 0 y $3n-1$.

En la sección siguiente probaremos que esto es equivalente a resolver el problema planteado originalmente.

1.2.2. Correctitud y optimalidad

La siguiente proposición garantiza la correctitud y optimalidad de nuestra solución.

Proposición 1.1: Sea $P' = (v_0, v_{i_1}, \ldots, v_{i_p}, v_{3n-1})$ un camino mínimo de v_0 a v_{3n-1} en G', entonces $P = P' \bmod n$ [2] es camino de v_0 a v_{n-1} en G, mínimo entre los que pasan por al menos dos aristas especiales.

Demostración: Hay que ver tres cosas respecto de P: que es camino de v_0 a v_{n-1}, que pasa por al menos dos aristas especiales, y que es mínimo respecto a los caminos que cumplen ambas cosas.

Va a ser útil recordar que $f_k : V \to V_k / f_k(v_i) = v_{k\times n+i}$ para $k \in \{0,1,2\}$ son las biyecciones de los isomorfismos planteados en la sección anterior. Además una pequeña observación que usaremos fuertemente:

Observación 1.1: $G = (V, E)$. $(u, w) \in E$ pero $(u, w) \notin M$ si y solo si $(f_0(u), f_0(w)), (f_1(u), f_1(w))$ y $(f_2(u), f_2(w))$ son aristas de G'.
En cambio, $(u, w) \in M$ si y solo si los arcos $(f_0(u), f_1(w))$, $(f_0(w), f_1(u)), (f_1(u), f_2(w))$ y $(f_1(w), f_2(u))$ están en G'.
Ambas cosas valen por como construimos G'.

- Es camino: Ante todo, por definición del operador módulo vale que

$$(\forall v_i \in P)\ 0 \leq i \bmod n \leq n-1$$

  Es decir que todos los vértices de P pertenecen a G, y además empieza en v_0 y termina en $v_{n-1} = v_{(3n-1) \bmod n} = v_{(2n+n-1) \bmod n}$. Queda ver que efectivamente nodos consecutivos en P son adyacentes en G.
  Si $v_i, v_j \in P$ son consecutivos entonces $v_{h+i} \in P'$ tiene que ser adyacente con $v_{h'+j} \in P'$ (pues son nodos consecutivos en un camino), donde h y h' son un par de constantes múltiplos de n. Por el contexto del problema h y h' solo pueden ser $0, n$ o $2n$. Luego, si

[2] $P = P' \bmod n \Leftrightarrow P_i = v_{j \bmod n}$ donde $P'_i = v_j$, $i = 0, \ldots, k-1$

$h = h' = k \times n$, por la observación 1.1, si v_{h+i} es adyacente a v_{h+j} en G_k (cosa que pasa, sino no podría pasar en G') entonces v_i es adyacente a v_j en G. Si $h \neq h'$, seguro que cada nodo es el extremo de un arco (pues están en diferentes niveles). Pero por construcción de G', solo puede haber un arco entre v_{h+i} y $v_{h'+j}$ si había una arista especial entre v_i y v_j en G, lo que significa que eran adyacentes. Luego, queda probado que P es un camino válido de v_0 a v_{n-1}.

- Pasa por al menos dos aristas especiales: el nodo v_{3n-1} pertenece al nivel 2 de G'. Esto significa que paso por dos arcos. Un arco entre $v_{k\times n+i}$ y $v_{(k+1)\times n+j}$ en G' solo existe si $(v_i, v_j) \in M$. Por definición de P, si tal arco pertenece a P' entonces tal arista especial pertenece a P. Luego, P tiene al menos dos aristas de M (podría tener más, pues en el tercer nivel las aristas especiales siguen existiendo y no son arcos).

- Es mínimo: Supongamos que P no es óptimo para el problema. Entonces existe Q tal que $|Q| < |P|$ y cumple con pasar por dos aristas de M. Construyamos Q', un camino de v_0 a v_{3n-1} en G', basado en Q.
  El método propuesto a continuación para armar Q' hace uso fuertemente de la observación 1.1, pues esta nos garantiza que las aristas y arcos mencionados efectivamente existen en G'.
  La idea es la siguiente: recorremos las aristas de Q en orden y las vamos poniendo en Q' hasta encontrar la primer arista especial, (v_i, v_j). En su lugar agregamos el arco $(v_i, v_{n+j}) = (f_0(v_i), f_1(v_j))$ a Q'. Seguimos completando Q' con aristas $(f_1(u), f_1(w))$ por cada arista común (u, w) que encontramos en Q. Eventualmente llegamos a una segunda arista especial (por hipótesis existe), (v_s, v_t), y en su lugar agregamos el arco $(v_{n+s}, v_{2n+t}) = (f_1(v_s), f_2(v_t))$ a Q'. Ahora bien, en este punto si Q es mínimo lo mejor que puede hacer es tomar el camino de distancia mínima desde v_t hasta v_{n-1}. Pero tal camino es isomorfo a un camino desde $v_{2n+t} = f_2(v_t)$ hasta $v_{3n-1} = f_2(v_{n-1})$, pues el tercer nivel es isomorfo a G. Por lo tanto agregando este camino a Q', llegamos a que Q' es un camino de v_0 a v_{3n-1} en G'.
  Pero como $|Q'| = |Q| < |P| = |P'|$, encontramos un camino más corto que P' en G' tal que conecta los mismos vértices. Esto es absurdo, pues por hipótesis P' era camino mínimo. Luego, el absurdo provino de suponer que P no era óptimo para el problema.

Finalmente, queda demostrada la proposición. □

De hecho, la recíproca de esta proposición también vale. No lo probamos sin embargo porque no es necesario para la correctitud de la solución. La forma de probarlo sería similar igualmente.

1.2.3. Explicación del código

Notar que para las dimensiones del grafo que toma BFS no usamos n y m sino n' y m', para no confundir con las dimensiones del problema original puesto que pueden ser diferentes, y de hecho por cómo lo vamos a usar, así va a ser.

Algorithm 1 Pseudocódigo del procedimiento BFS

```
1: procedure BFS(ListaAdyacencia vs, vertice root, vertice target, int n′)→ Vector<vertice>
2:     cola<vertice> c ← Vacia()                                  ▷ O(1)
3:     vector<int> distancia(n′, ∞)                               ▷ O(n′)
4:     vector<vertice> acm(n′, −1)                                ▷ O(n′)
5:     distancia[root] ← 0                                        ▷ O(1)
6:     acm[root] ← root                                           ▷ O(1)
7:     c.push(root)                                               ▷ O(1)
8:     while ¬c.vacia?() do                                       ▷ O(n) veces
9:         actual ← c.pop()                                       ▷ O(1)
10:        VerticesAdyacentes vecinos ← vs[actual]                ▷ O(1)
11:        for v ∈ vecinos do                                     ▷ O(d(v)) veces
12:            if distancia[v] = ∞ then                           ▷ O(1)
13:                distancia[v] ← distancia[actual] + 1           ▷ O(1)
14:                acm[v] = actual                                ▷ O(1)
15:                if v = target then                             ▷ O(1)
16:                    break                                      ▷ O(1)
17:                c.push(v)                                      ▷ O(1)
18:    int long_sol ← distancia[target] − 1                       ▷ O(1)
19:    vector<vertice> solucion(long_sol, 0)                      ▷ O(long_sol) ⊆ O(m′)
20:    vertice v ← acm[target]                                    ▷ O(1)
21:    for int i desde long_sol − 1 hasta 0 do                    ▷ O(m′) veces
22:        solucion[i] ← v                                        ▷ O(1)
23:        v ← acm[v]                                             ▷ O(1)
24:    return solucion
```

La implementación de BFS que realizamos está compuesta por dos partes: la primera hasta la línea 17 inclusive, es la implementación clásica del algoritmo de búsqueda en anchura para determinar caminos mínimos, en particular modificada un poco para que termine a penas compute un camino hasta el nodo objetivo pues es el único que nos importa realmente; la segunda consiste en reconstruir el camino mínimo entre *root* y *target* a partir del árbol de caminos mínimos. Notar que la función tiene como precondición que efectivamente exista algún camino desde *root* hasta *target*.

Para la primer parte tenemos esencialmente tres estructuras importantes:

- una cola FIFO de vértices donde iremos encolando los vecinos del nodo en el que estamos actualmente y que todavía no hayamos visitado; la misma está implementada sobre una lista doblemente enlazada, lo que permite que las operaciones de encolar, desencolar y ver el siguiente elemento sean todas $O(1)$.
- un vector de distancias tal que la posición i-ésima del mismo guarda la distancia desde el *root* hasta el nodo i (o bien ∞ si todavía no pasamos por i, o simplemente i no es alcanzable desde *root*).
- un vector de vértices que representará nuestro árbol de caminos mínimos desde el *root* hasta cualquier nodo, de forma que en la posición i-ésima del vector tendremos al padre

del nodo i en el árbol (o bien -1, si todavía no pasamos por i, o simplemente no es alcanzable desde $root$).

Que la búsqueda es correcta es resultado inmediato de que el algoritmo es el BFS tradicional cuya correctitud es conocida. Una discusión más detallada de este algoritmo junto con una demostración de su correctitud puede encontrarse en [Cor+09, Ch. 22.2].

Una vez hallado un camino mínimo hasta el nodo $target$, queremos ahora armar un vector de vértices que contenga a todos los vértices de dicho camino. Como la distancia es la cantidad de vértices en el camino menos uno, entonces el vector tendrá que tener tamaño igual a la distancia más uno. Pero como no nos interesa que el primer y último nodos estén en el vector entonces nos queda que el largo del mismo será $l = d(root, target) - 1$. Luego, es cuestión de llenar las posiciones de este vector de atrás para adelante, pues *a priori* para cada nodo solo sabemos cual es su padre en el árbol de caminos mínimos. La última posición tendrá al padre del nodo $target$, la anteúltima al padre del padre y así. Iterando l veces llegamos a que en la posición inicial del vector hay un nodo que es hijo de $root$ y ancestro de $target$.

Finalmente devolvemos este vector.

Algorithm 2 Pseudocódigo del main

procedure MAIN
    `int` n ▷ Cantidad de nodos
    `vector(vector(int))` $input$ ▷ $input[i]$ almacena los datos de la i-ésima arista pasada
    $inicializar(input, n)$ ▷ Leemos los datos pasados como parámetros e inicializamos
    `ListaAdyacencia` $adj\_list(3 * n, VerticesAdyacentes())$ ▷ $O(3 \times n) = O(n)$
    for $(v_1, v_2, e) \in input$ **do** ▷ m veces
        if $e = True$ **then** ▷ $O(1)$
            $adj\_list[v_1].push\_back(v_2 + n)$ ▷ $O(1)$
            $adj\_list[v_1 + n].push\_back(b + 2 \times n)$ ▷ $O(1)$
            $adj\_list[v_2].push\_back(v_1 + n)$ ▷ $O(1)$
            $adj\_list[v_2 + n].push\_back(v_1 + 2 \times n)$ ▷ $O(1)$
        else
            $adj\_list[v_1].push\_back(v_2)$ ▷ $O(1)$
            $adj\_list[v_2].push\_back(v_1)$ ▷ $O(1)$
            $adj\_list[v_1 + n].push\_back(v_2 + n)$ ▷ $O(1)$
            $adj\_list[v_2 + n].push\_back(v_1 + n)$ ▷ $O(1)$
        $adj\_list[v_1 + 2 \times n].push\_back(v_2 + 2 \times n)$ ▷ $O(1)$
        $adj\_list[v_2 + 2 \times n].push\_back(v_1 + 2 \times n)$ ▷ $O(1)$
    `vector<vertice>` $solucion \leftarrow bfs(adj\_list, 0, 3 \times n)$ ▷ $O(3n + 6m) = O(n + m)$
    $print(solucion.size() + 1)$
    for $v \in solucion$ **do**
        $print(v \% n)$

Nuestra función main tiene tres partes importantes:

- El armado de la lista de adyacencias de G'. Si la arista que estamos agregando es especial, agregamos los arcos correspondientes del nivel 0 al 1, y del 1 al 2. Si no lo es, entonces agregamos la arista tanto en el nivel 0 como en el 1. En ambos casos, agregamos la arista normalmente en el nivel 2.

- El llamado a BFS pasando como parámetros la lista de adyacencias anterior, tomando como *root* el nodo 0 y como target el $3n-1$. Dichos parámetros cumplen las precondiciones de BFS por el Lema 1.1 y por ser todas aristas de peso 1.
- La impresión del resultado. Acá es importantísimo notar que imprimimos los vértices módulo n pues lo que nos devuelve BFS son nodos del grafo G' y no de G. Por la Proposición 1.1 esto efectivamente constituye una solución al problema original.

1.3. Complejidad del algoritmo

La complejidad en peor caso de la solución es la complejidad de la función *main*. Omitiendo las partes de lectura y escritura de datos, tenemos que el costo de dicha función es el costo de armar el nuevo grafo más el costo de realizar BFS sobre él.

Viendo el algoritmo 2, el costo de armar el grafo es $O(n + 6m) = O(n + m)$. Vale destacar que esta complejidad es además claramente una cota inferior, pues el costo de armar el grafo nuevo depende únicamente de la cantidad de vértices y aristas, y no de características más particulares como la cantidad de aristas especiales, la longitud de los caminos simples, etc. Por lo tanto el algoritmo en general debe ser al menos $\Omega(n + m)$.

Por otra parte, observando el algoritmo 1, BFS tiene una complejidad en peor caso de

$$\begin{aligned} O(1 + 2n' + 3 + 2n' + (\sum_{i=0}^{n'-1} d(v_i)) \times 5 + m' + 1 + 2m') &= O(5 + 4n' + 2m' \times 5 + 3m') \\ &= O(4n' + 13m') \\ &= O(n' + m') \end{aligned} \tag{1}$$

Notar que en el primer término podemos escribir la sumatoria de los grados de todos los nodos debido a que en peor caso hará falta pasar por todos ellos, y por otra parte sabemos que pasamos por cada uno exactamente una vez. El segundo término resulta de agrupar y reemplazar la sumatoria por $2m'$ (cosa que podemos hacer pues es una identidad válida para todos los grafos).

En nuestro problema concreto $n' = 3 \times n$ y $m' = 2 \times 3 \times m = 6 \times m$ (por cada arista que sacamos estamos poniendo dos arcos).

Luego, la complejidad asintótica del algoritmo en peor caso es $O(n+m+3n+6m) = O(4n+7m) = O(n+m)$. Por otra parte como dijimos que también era $\Omega(n+m)$, tenemos que es $\Theta(n + m)$.

De hecho, asintóticamente también lo es en mejor caso (cuando existe un camino de longitud 3): si bien BFS puede ser $\Theta(1)$ debido a que nuestra implementación termina de buscar una vez que encuentra al nodo deseado, armar el grafo sigue siendo $\Theta(n+m)$ en cualquier caso. En definitiva no tiene sentido hablar de mejor o peor caso pues ambos son iguales en términos asintóticos.

1.4. Performance del algoritmo

Como dijimos anteriormente, el algoritmo tiene una complejidad de $\Theta(n+m)$. El algoritmo no tiene peor o mejor caso propiamente dichos (dado que es $\Theta(n+m)$ para todos los casos), pero veremos que, tomando n fijo y moviendo m, podemos hacer variar el tiempo que toma el algoritmo.

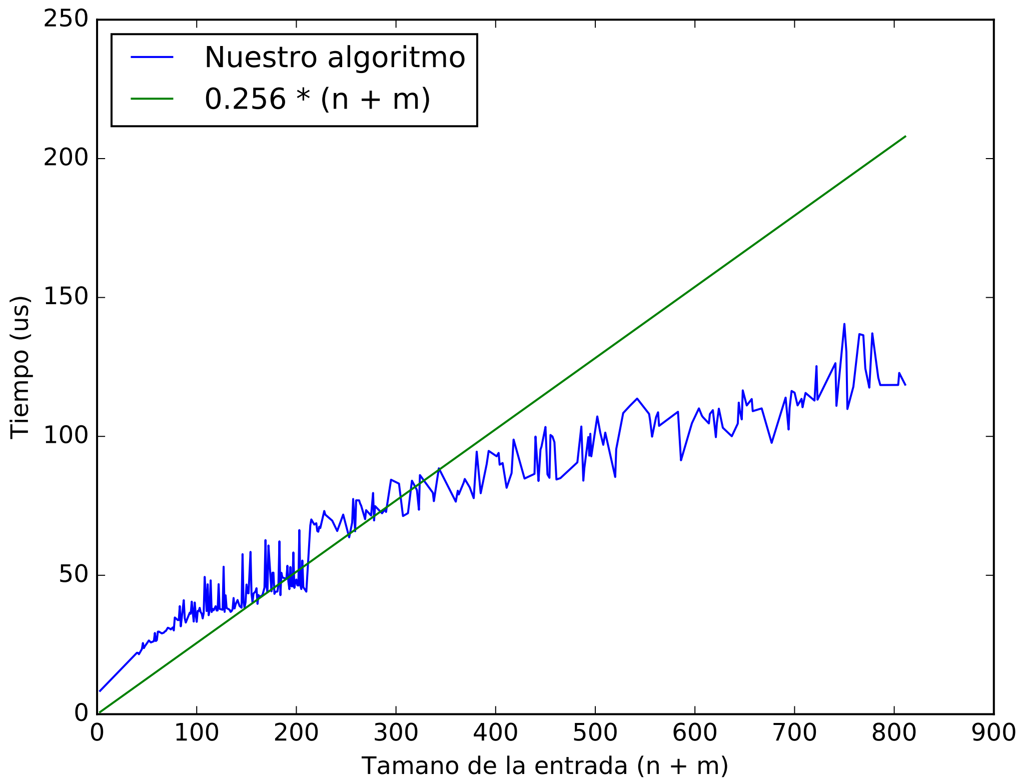


Figura 4: Tiempo que toma el algoritmo en μs para una entrada de tamaño $n+m$. m al azar entre $n-1$ y $\frac{n(n-1)}{2}$.

Como se observa, la implementación tiene complejidad lineal sobre $n+m$, como era esperado.

Para confirmarlo, usamos el gráfico de la funcion $\frac{T(n+m)}{n+m}$, donde T es el tiempo que tarda el algoritmo para la entrada de tamaño dado. Si vemos que converge a una constante, estaremos en el caso exacto de la definción de $\Theta(f(n))$.

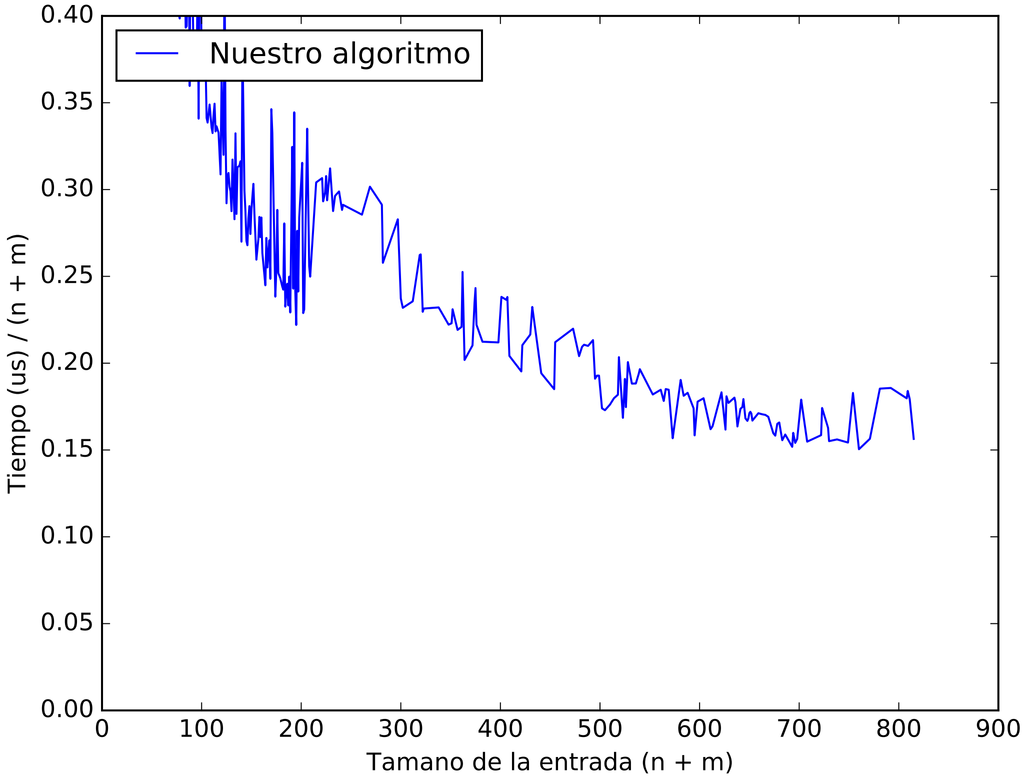


Figura 5: Tiempo que toma el algoritmo en μs dividido $n+m$ para una entrada de tamaño $n+m$. m al azar entre $n-1$ y $\frac{n(n-1)}{2}$

El ruido del gráfico se debe a que la escala es otra y distorciona las distancias entre los puntos. Sin embargo, se puede observar que converge a una constante, como era esperado.

Como habiamos dicho anteriormente, aunque el algoritmo es $\Theta(n+m)$, podemos ver casos particulares del algoritmo, en el que n está fijo y movemos m y ver como se comporta el algoritmo.

Primero veamos el caso en el que $m \in O(n)$. Esperaríamos que el algoritmo aquí tenga una complejidad de $O(n+m) = O(n+n) = O(n)$. Esto fue confirmado experimentalmente, como se muestra a continuación.

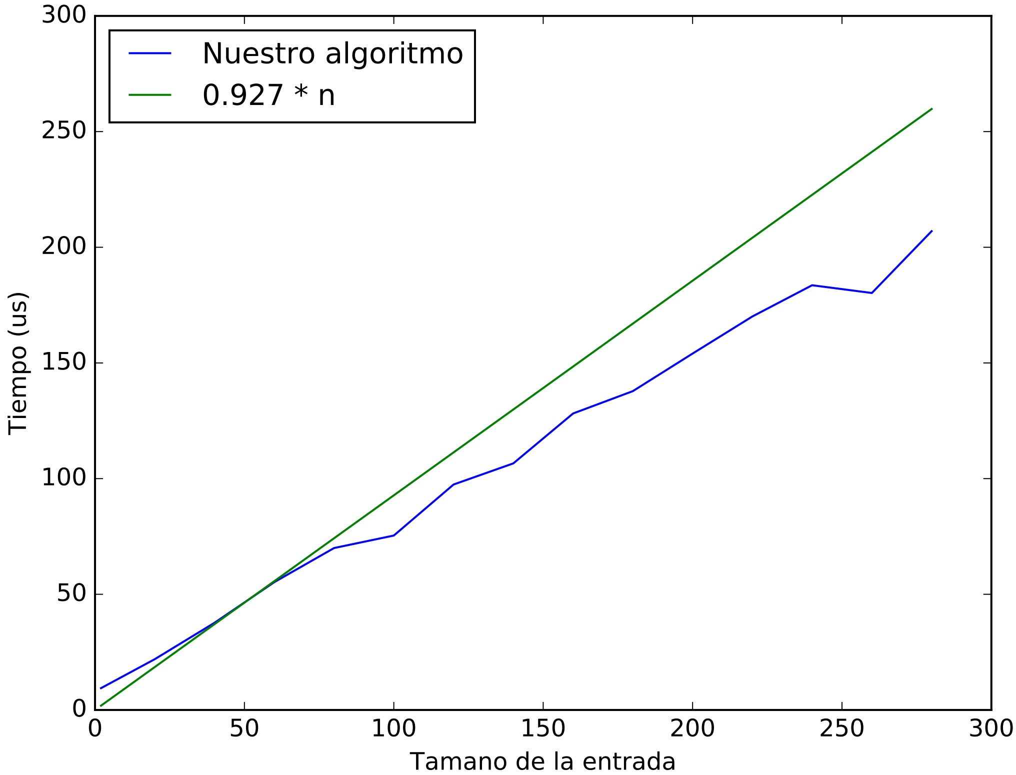


Figura 6: Tiempo que toma el algoritmo en μs para una entrada de tamaño n ($m \in O(n)$).

Ahora veamos el caso en el que $m \in O(n^2)$. Esperaríamos que el algoritmo aquí tenga una complejidad de $O(n+m) = O(n+n^2) = O(n^2)$. Esto fue, nuevamente, confirmado experimentalmente.

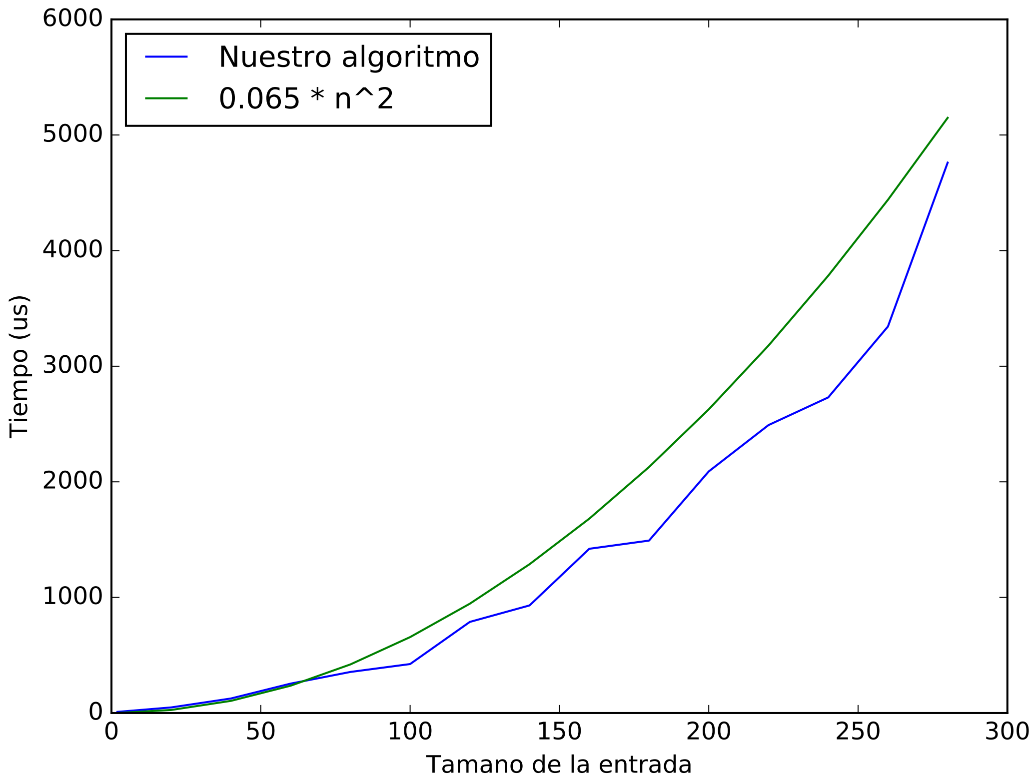


Figura 7: Tiempo que toma el algoritmo en μs para una entrada de tamaño n ($m \in O(n^2)$).

Por último, como diremos en detalle en la siguiente sección, en la que explicamos la métodología de experimentación, en todos los experimentos anteriores asumimos que no importan la cantidad de caminos especiales de un grafo dado.

Esto es bastante obvio desde el punto de vista del algoritmo (pues es una consecuencia inmediata de que la complejidad esté dominada por la construcción del grafo), pero nos parece algo muy interesante verificarlo experimentalmente, dado que en este hecho se basan todos los experimentos anteriores.

Vale notar que, no obstante, sí puede haber pequeñas diferencias como consecuencia de que BFS ciertamente puede variar su performance a partir de cómo se conectan los vértices y cuántas aristas especiales hay. Pero esto afecta solo constantes, que se traducen en poco más que ruido en el gráfico.

Como puede verse en la siguiente figura, si tomamos n y m fijos, la cantidad de caminos especiales del grafo no afectan significativamente la performance del algoritmo.

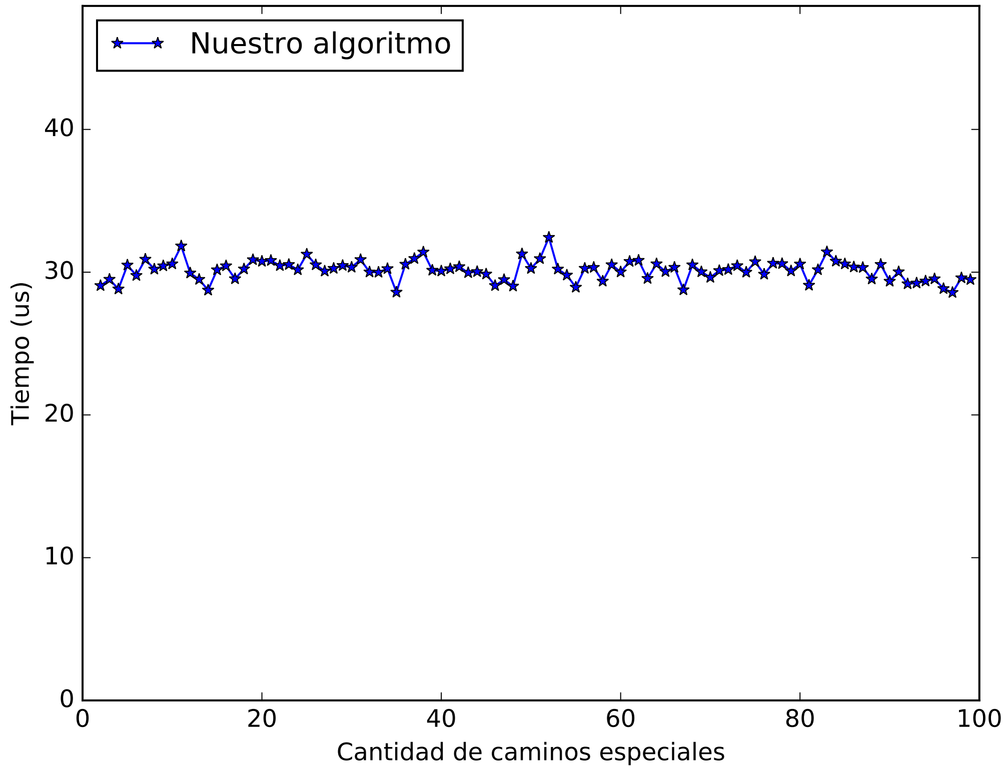


Figura 8: Tiempo que toma el algoritmo en μs para una entrada de tamaño $n = 15, m = 100$, variando la cantidad de caminos especiales.

1.4.1. Método de experimentación

Para la experimentación general del algoritmo, es decir, la verificación de que su complejidad era de $\Theta(n + m)$, generábamos distintos grafos al azar (n al azar y m elegido al azar tal que quede conexo).

En los casos particulares, dado n fijo, tomamos $m = n - 1$ para el primer experimento y $m = \frac{n(n-1)}{2}$ en el segundo.

Para la generación al azar de grafos utilizamos el algoritmo descripto en la sección 4.1 del apéndice. Vale la pena aclarar cómo hicimos para decidir cuántas aristas especiales tendría el grafo. Primero, lo que hicimos fue notar experimentalmente que, con n y m fijos, si tomábamos distinta cantidad de caminos especiales (de 2 a m), la varianza de las mediciones era muy baja, es decir, la cantidad de caminos especiales de un grafo no afecta a la performance.

Esto fue observado y comprobado experimentalmente. En consecuencia, para cada n y m fijo, tomamos grafos totalmente al azar, con cantidad de caminos especiales también al azar, y calculamos la mediana de todos los tiempos para determinar el tiempo total.

2. El Imperio Contraataca

2.1. Explicación formal del problema

El problema dado se puede modelar como un grafo (no dirigido) en el cual cada vértice es un planeta y las aristas representan las rutas entre los mismos, estas tienen pesos asignados los cuales representan la cantidad de litros que requieren para ser recorridas. Como se aclara que se puede llegar de un planeta a cualquier otro, podemos asumir que el grafo es conexo.

El objetivo es, a partir de un nodo 0, visitar todos los demás gastando la menor cantidad de litros posible. Esta idea de visitar todos los vértices del grafo y al mismo tiempo gastar la menor cantidad de litros se puede traducir a buscar un subgrafo de ciertas particularidades, entre otras cosas que comparta los mismos nodos que el grafo original pero sacando las aristas que no valgan la pena recorrer porque podemos atravesar otras menos costosas, pero de esto hablaremos en mas detalle en la siguiente parte del informe, por ahora veamos algunos ejemplos de soluciones validas e invalidas.

Dado el el grafo de la figura (9), el nodo 0 es el inicial y debemos visitar todos los demás.

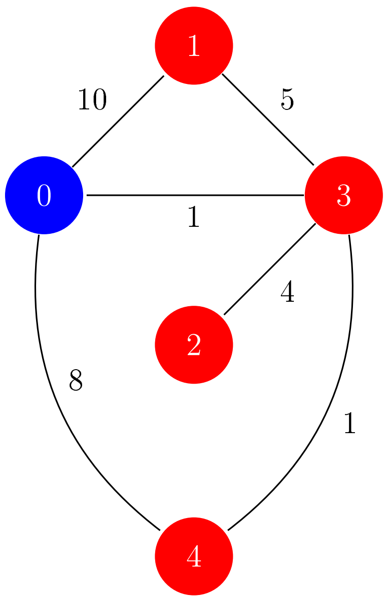


Figura 9: Ejemplo de una instancia posible del problema.

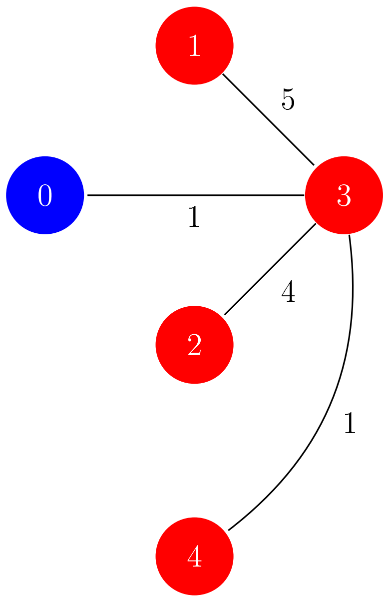


Figura 10: El recorrido representado es óptimo.

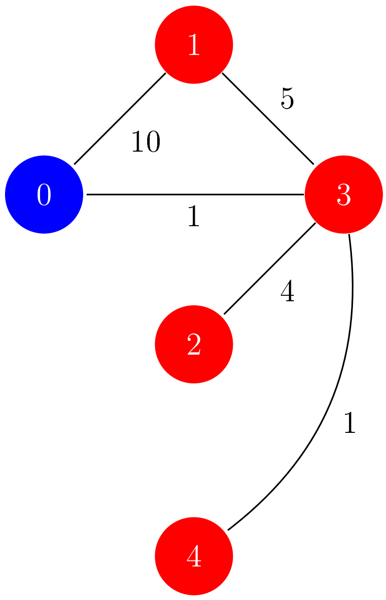


Figura 11: El recorrido representado no es óptimo. Si bien tiene todos los caminos mínimos también tiene uno de mas, el camino directo del vértice 0 al 1, este agrega un costo de 10 el cual es innecesario debida a que de todas maneras ya estábamos visitando con costo 6 al planeta 1.

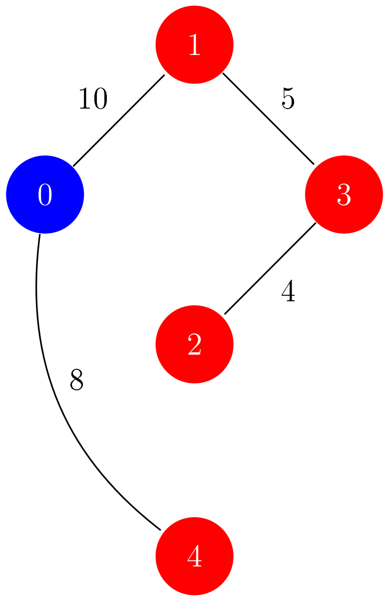


Figura 12: El recorrido representado por el grafo no es una solución ya que hubiera sido mas económico reemplazar el trayecto de 0 a 3 por un camino directo de costo 1, o reemplazar el camino de 0 a 4 por un camino indirecto de costo 2 en vez de el camino directo de costo 8.

Como aclaración importante notar que según el grafo la solución podría no ser única, pensar por ejemplo en un grafo completo (De $n > 2$ nodos) que tiene todas sus aristas de peso uno.

2.2. Explicación de la solución

Como ya dijimos el problema se puede entender como un grafo, en el cual debemos a partir del nodo 0 visitar todos los demás. Este recorrido solución no va a tener ciclos ya que eso implicaría que pasamos dos veces por un mismo planeta agregando un costo innecesario, por otro lado, este deberá ser conexo ya que sino tendremos planetas a los cuales no podremos visitar, sabiendo esto y que queremos que el recorrido sea de costo mínimo se deduce que lo que estamos buscando es equivalente al cálculo de un árbol generador mínimo. Para realizar esto decidimos utilizar uno de los algoritmos vistos en la teórica. En los siguientes párrafos hablaremos de como realizamos su implementación, porqué es correcto y cual es su complejidad.

2.2.1. Explicación del código

Para este ejercicio el código es un poco mas extenso que en los demás debido a que se tuvo que crear un heap que cumpla ciertas particularidades. Si bien en gran parte es un heap binario clásico se diferencia en que tiene un vector extra llamado *pos* que almacena la posición de cada nodo en el heap, esto es para asegurar un costo de $O(log(n))$ en ciertas operaciones. Debido a que el heap, al igual que los vectores, arrays y demás son estructuras típicas con las cuales asumimos todos los que lean este informe estarán familiarizados asumiremos cierto conocimiento previo y omitiremos una explicación profunda sobre el funcionamiento del código en esas partes para no extender de forma innecesaria el informe e ir directo a la explicación de los algoritmos vistos en la materia, de todas maneras mas adelante explicaremos con un detalle un poco mayor las complejidades de las funciones del heap, si alguna mención no queda del todo clara se puede recurrir al código del programa el cual esta en el apéndice del informe.

Los vértices del heap, llamados *vertex* están formados por dos *integers*, primero *key* el cual es el nombre que le asignamos al nodo y segundo *value* que representará la menor distancia encontrada hasta el momento desde su candidato a padre hasta dicho vértice. El heap ordena sus *vertex* de menos a mayor respecto de *value*.

VerticesAdyacentes es un vector con pares de enteros, el primero del par representa el numero del nodo adyacente y el segundo el peso del camino que lleva hacia el.

La función principal es la siguiente.

Algorithm 3 Pseudocódigo del main

```
procedure MAIN
    int n                                              ▷ Cantidad de nodos
    int m                                              ▷ Cantidad de aristas
    ListaAdyacencia adj_list(n, VerticesAdyacentes())  ▷ O(n)
    inicializar(adj_list)                              ▷ adj_list[i] = nodos adyacentes al nodo i, O(m)
    vector<Vertex> v_inicial                           ▷ O(1)
    v_inicial.push_back(nodo(0, 0))                    ▷ O(1)
    for inti = 1; i < n; i++ do                        ▷ n veces
        v_inicial.push_back(nodo(i, ∞))                ▷ O(1)
    MinHeap h(v_inicial)                               ▷ Creo a h, un heap a partir de v_inicial, O(n.log(n))
    vector<int,int> resultado ← Prim(h, adj_list)      ▷ O(m.log(n))
    int litros ← 0                                     ▷ O(1)
    for x ∈ resultado do
        litros ← litros + x.second
    imprimir_resultados(resultado, litros)             ▷ O(n)
```

Lo que nuestra función main hace se puede describir en los siguientes puntos.

- El armado de la lista de adyacencias de *adj_list*.
- El armado del heap *h*.
- El cálculo del AGM con Prim
- El cálculo del peso del árbol
- La impresión del resultado.

Conociendo las operaciones usuales de un heap y habiendo resaltado las características distintivas de nuestra implementación veamos ahora como realizamos la función *Prim*, esta toma el heap *h* y la lista de adyacencia *adj_list* como parámetros. Notar que los valores de los *vertex* del heap convierten al nodo 0 en la raíz.

El programa devuelve un vector *parent* de tamaño n el cual indica que en el AGM resultante el nodo i tiene como padre al nodo $parent[i].first$ y para llegar a el desde su padre se gastan $parent[i].second$ litros.

2.2.2. Pseudocódigo

Algorithm 4 Pseudocódigo de Prim

```
procedure PRIM(MinHeap heap, Vector<VerticesAdyacentes> adj_list) → Vector<int,int>
    Vector<int,int> parent(heap.Size(), < 0, 0 >)                    ▷ O(n)
    while heap.Size() > 0 do                 ▷ n veces, se hace un pop en cada iteración
        Vertex min_vertex ← heap.Pop()                                ▷ O(log(n))
        int u ← min_vertex.key                                        ▷ O(1)
        for vertex_and_weight ∈ adj_list[u] do                        ▷ d(u) veces
            int v ← vertex_and_weight.first                           ▷ O(1)
            int weight ← vertex_and_weight.second                     ▷ O(1)
            if heap.At(v) ∧ weight < heap.Value(v) then               ▷ O(1)
                parent[v].first ← u                                   ▷ O(1)
                parent[v].second ← weight                             ▷ O(1)
                heap.DecValue(v, weight)                              ▷ O(log(n))
    return parent
```

Intentemos entender que es lo que hace este algoritmo y luego veremos por qué es correcto y óptimo.

Primero se construye *parent* el vector de tuplas inicializado en pares de dos ceros.

Luego se comienza a iterar, en cada iteración se sacará el vértice de menor *value* del heap, este ya no será mas actualizado en el vector *parent* por lo que su posición en el árbol quedará fijada. Luego se ve por cada uno de sus vértices adyacentes si actualizar *parent* o no. Como se observa en la guarda del if esto sucederá si el vector adyacente esta aún en el heap (o sea si aun existe la chance de mejorar el camino) y si su distancia al último nodo que se extrajo de la cola es menor que el valor que tiene actualmente en el heap, o sea si efectivamente podemos mejorar el camino cambiando el padre. En ese caso se actualizará su valor en el heap y se guardara su nuevo padre y su nuevo peso en *parent*.

2.2.3. Correctitud y Optimalidad

Veamos por que el algoritmo de Prim es correcto, o sea, veamos por que dado un grafo G conexo y ponderado nos asegura conseguir un árbol generador mínimo.

Primero veamos que el resultado describe realmente un árbol generador. Como el grafo es conexo y en el heap todos los nodos (excepto el primero) arrancan con $value = \infty$ eventualmente todos los nodos (excepto el primero) caen al menos una vez dentro del if agregando así al vector de salida un padre y la distancia a la que están de el. El resultado final por lo tanto nos da un árbol debido a que cada nodo tiene un único padre y es generador de G por haber sido formado a partir de sus propios nodos (todos ellos) y aristas.

Veamos ahora porque este grafo, A, es mínimo. Como la cantidad de árboles generadores de un grafo es finita existe alguno que es el mínimo, llamémoslo A'. Si A es igual a A' entonces es un AGM. Si no lo es, sea e la primer arista agregada durante la construcción de A, que no está en A' y sea V el conjunto de nodos conectados por las aristas agregadas antes que e, o sea las que ya quedaron fijadas al sacar el nodo del heap. Entonces un extremo de e está en V y el otro no. Ya que A' es el árbol generador mínimo de G hay un camino en A' que une los dos extremos. Moviéndose por este camino, se debe encontrar eventualmente una arista f uniendo un nodo de V a uno que no está en V. En la iteración que e se agrega a A, f también se podría haber agregado y se hubiese agregado en vez de e si su peso fuera menor que el de e. Ya que f no se agregó se concluye que $P(f) \geq P(e)$.

Sea A'' el grafo obtenido al remover f y agregar e a A'. Es fácil mostrar que A'' es conexo: al quitar una arista de un árbol quedan determinadas exactamente dos componentes conexas, que luego conectamos con una arista que tiene un extremo en cada componente. Que tiene la misma cantidad de aristas que A' es trivial, y que el peso total de sus aristas no es mayor que el de A' se deduce de lo dicho en el párrafo anterior. Entonces también es un AGM de G y contiene a e y todas las aristas agregadas anteriormente durante la construcción de V.

Si se itera este procedimiento, eventualmente se obtendrá el árbol generador mínimo de G que es igual a A.

Esto demuestra que A es el AGM de G.

2.3. Complejidad del algoritmo

Dado un grafo G, conexo y ponderado, nuestro algoritmo tendrá una complejidad de $O(m \log(n))$ para el peor caso, la cual podrá ser mejorada para algunas instancias consiguiendo $O(m + n \log(n))$.

Analicemos de donde salen dichas afirmaciones. primero veremos el peor caso. La función main esta compuesta por la primer parte donde genera un vector de adyacencia, primero inicializado en valores nulos $O(n)$ y luego llenado con los datos de entrada $O(m)$.

Luego se crea un vector $v\_inicial$ el cual será el heap que recibirá de entrada la función $Prim$ este se llena con n nodos, el primero inicializado en 0 y los demás con $value$ infinito y un key distintivo entre 1 y $n-1$. Esto cuesta $\theta(n)$. Para convertirlo luego en un heap se utiliza $MinHeap$ que cuesta $O(n \log(n))$ por utilizar n veces a la función $MinHeapify$ que es $O(\log(n))$, ya que es una función recursiva que se ejecuta a si misma pero con una entrada de la mitad de tamaño, por lo que en el peor de los casos se llama $\log_2 n$ veces.

Una vez hecho esto se ejecuta Prim, viendo el pseudocódigo se puede ver que realizamos por cada elemento del heap un pop el cual cuesta $O(log(n))$ por utilizar $MinHeapify$ mas otras operaciones de costo constante.

Luego por cada vértice adyacente de cada nodo hacemos una operación que cuesta a lo sumo $O(\log(n))$ si caemos dentro del if, esto es por la función $DecValue$, la cual tarda a lo sumo la altura del árbol del heap en terminar, funciona intercambiando el nodo al cual le cambiaron el valor por su padre hasta que se convierta de nuevo en un heap válido. Esto fue observado y comprobado experimentalmente. En consecuencia, para cada n y m fijo, tomamos grafos totalmente al azar, con cantidad de caminos especiales también al azar, y calculamos la mediana de todos los tiempos para determinar el tiempo total.

La complejidad resultante de Prim si caemos dentro de todos los if (peor caso) será:

$$O(\sum_{i=0}^{n-1} d(v_i)\log(n) + n\log(n)) =$$

$$O((\sum_{i=0}^{n-1} d(v_i) + n)\log(n)) =$$

$$O((2m+n)\log(n)) = O(2m\log(n)) = O(m\log(n))$$

Notar que podemos prescindir de n en la suma $2m+n$ ya que el grafo es conexo y por lo tanto n será menor a $2m$.

Finalmente se calcula el peso del árbol con una complejidad temporal de $O(n)$ a partir del vector de salida de *Prim* y se imPrime a la salida.

Sumando todo lo dicho se ve que nada supera la complejidad de *Prim* por lo que el costo final queda limitado por este en $O(m\log(n))$

2.3.1. Mejor Caso

Dado un n y m fijo la mayor parte de las operaciones del programa van a realizarse sin distinción en la forma que tenga el grafo aunque en el caso del if que contiene el algoritmo de Prim dentro las características del grafo si pueden importar. Si solo cayéramos n veces en el if (mejor caso), entonces vamos a realizar una operación que cuesta $\log(n)$ solo n veces de todas las iteraciones que hicimos ($\sum_{i=0}^{n-1} d(v_i)$), sabiendo esto la complejidad de Prim será la siguiente.

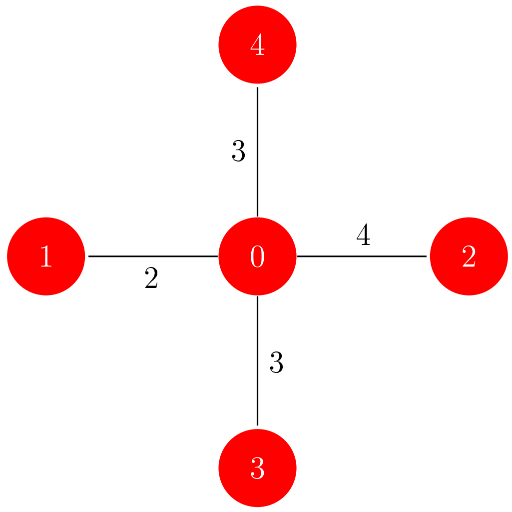
$$O(\sum_{i=0}^{n-1} d(v_i) - n + n\log(n) + n\log(n)) =$$

$$O(\sum_{i=0}^{n-1} d(v_i) - n + 2n\log(n)) =$$

$$O(2m - n + n\log(n)) = O(m + n(\log(n) - 1)) = O(m + n\log(n))$$

Se tendrá esta complejidad temporal por ejemplo cuando la guarda del if se cumpla para todos los adyacentes al primer nodo de heap y luego no se actualice mas el vector *parent*.

Esto puede suceder, por ejemplo en el siguiente grafo.



2.4. Performance del algoritmo

Este algoritmo tiene complejidad $O(m \log m)$. Sin embargo, como vimos anteriormente, esta no es una cota ajustada, si no que nuestro algoritmo puede acotarse más ajustadamente por $O(m \log n)$.

De todos modos, primero confirmemos que nuestro algoritmo tiene complejidad $O(m \log m)$.

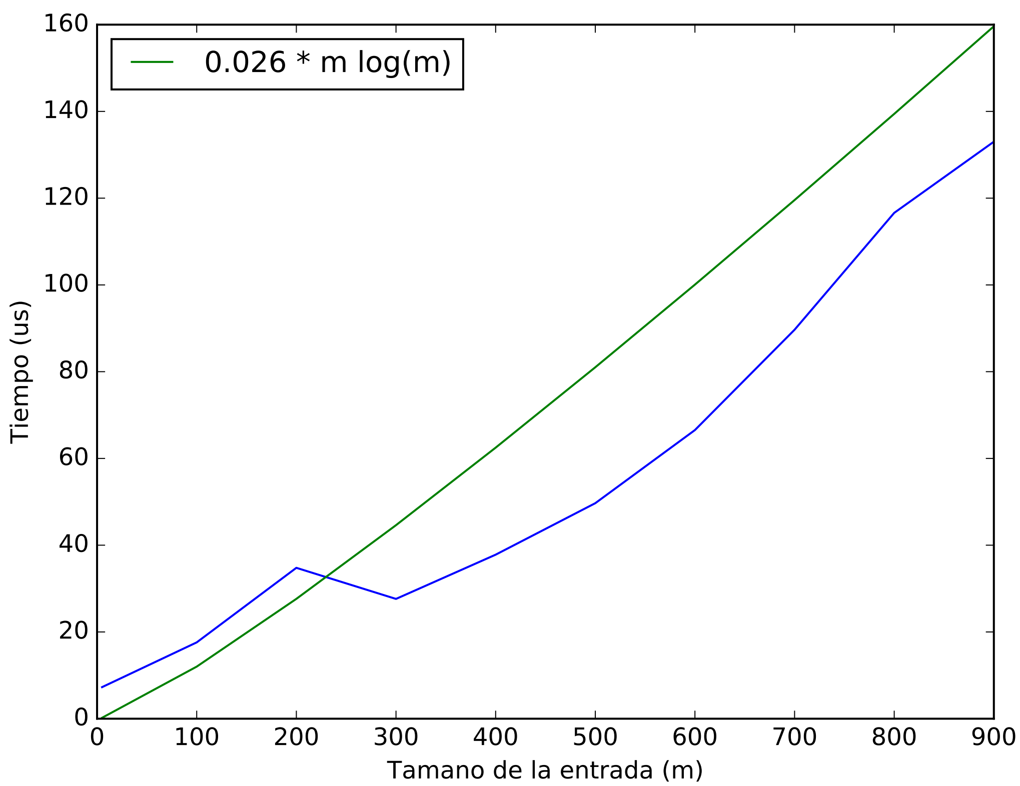


Figura 13: Tiempo que toma el algoritmo en μs para una entrada de tamaño m. n al azar.

Sin embargo, como dijimos antes, esta cota no es ajustada. Esto puede verse en el siguiente gráfico. Lo que hicimos fue tomar m fijo como antes, pero no mostrar las mediciones condensadas en un punto, si no que ahora mostramos todas las mediciones tomadas para un mismo m, variando n.

Como puede verse, la varianza es altísima.

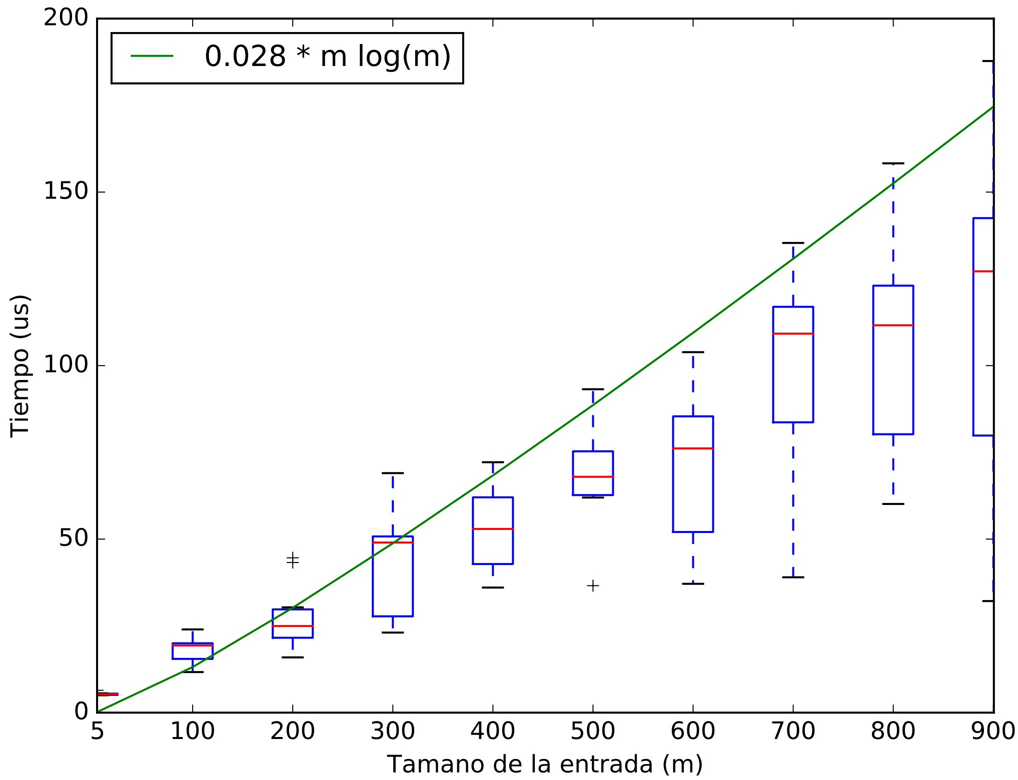


Figura 14: Tiempo que toma el algoritmo en μs para una entrada de tamaño m. n al azar. Se indican los valores del primer al tercer cuartil con un rectángulo azul y la mediana con una linea roja. El máximo y minimo se indican con lineas negras arriba y abajo del rectángulo.

Esto se debe a que en realidad, la complejidad no es independiente de n, si no que para m fijo, si se toma un n más pequeño, el tiempo que tarde el programa va a ser mucho menor que con un n un poco más grande.

Como dijimos antes, una cota más ajustada para nuestro algoritmo es la de $O(m \log n)$, así que pasemos a confirmar esto experimentalmente.

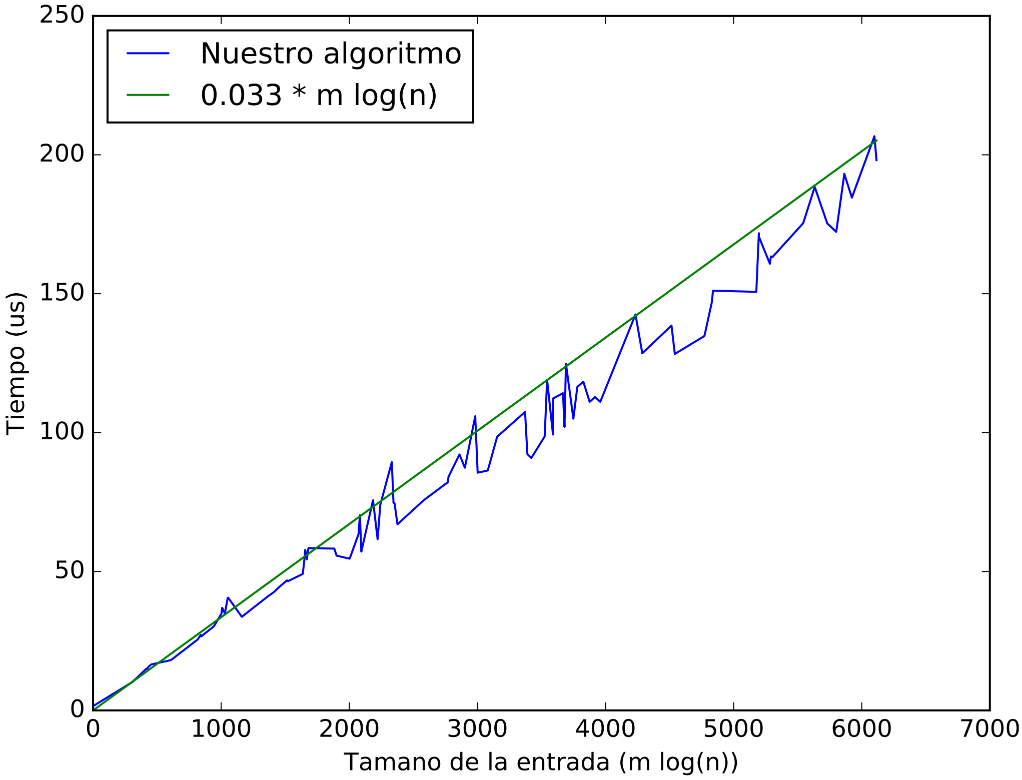


Figura 15: Tiempo que toma el algoritmo en μs para una entrada de tamaño $m \log n$.

Ahora pasemos a analizar los casos del algoritmo. Como habíamos dicho, el peor caso del algoritmo no se escapa de $O(m \log n)$. Veamoslo confirmado experimentalmente.

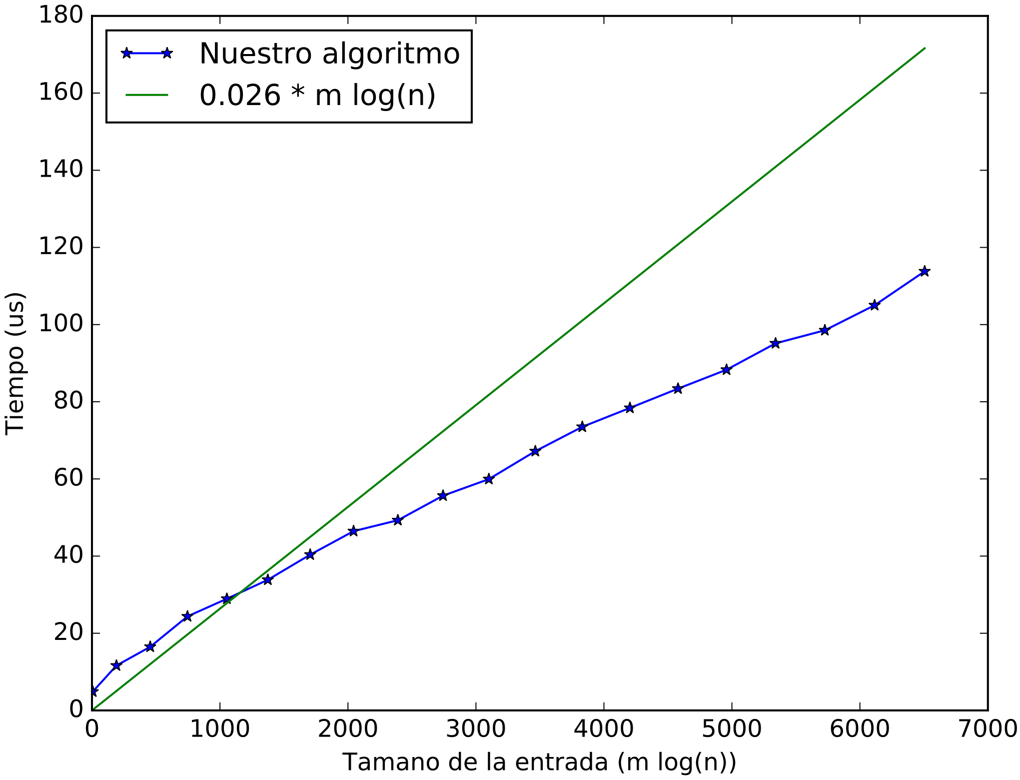


Figura 16: Tiempo que toma el algoritmo en μs para una entrada de tamaño $m \log n$.

En cuanto al mejor caso, como dijimos anteriormente, este posee una complejidad de $O(m + n \log n)$. De nuevo, lo confirmamos experimentalmente, con resultados positivos.

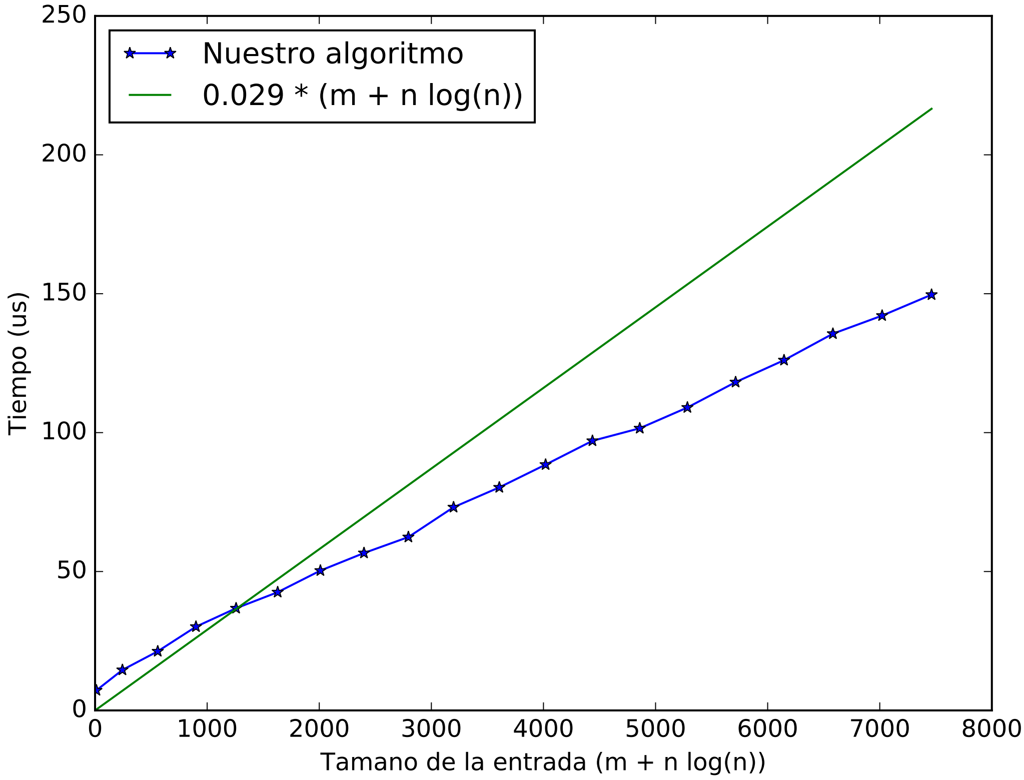


Figura 17: Tiempo que toma el algoritmo en μs para una entrada de tamaño $m + n \log n$.

2.4.1. Método de experimentación

Para generar los grafos al azar, al igual que en el problema anterior, usamos el algoritmo descripto en el apéndice.

Para las figuras 13 y 15 tomamos 100 mediciones (en el caso de 13 tomamos algunas más, porque para cada m tomábamos varios n distintos, al rededor de 10, o sea que las mediciones en este caso eran de 1000, 100 por cada n). De todas estas mediciones tomamos la mediana, que es lo que se representó en el gráfico.

Para el análisis de casos utilizamos grafos específicos, no generados al azar. Para el peor caso utilizamos el grafo camino, también conocido como P_n, que como explicamos anteriormente nos permite alcanzar el peor caso del algoritmo. Por otro lado, para el mejor caso utilizamos el grafo estrella, también conocido como S_n, que, como vimos antes, nos permite alcanzar el mejor caso del algoritmo.

En estos dos últimos experimentos nos pareció relevante marcar además los puntos donde tomamos mediciones, dado que el ruido es menor que el general, y sin las marcas no se entendería fácilmente donde fueron tomadas.

3. El Retorno del ~~que te~~ Jedi

3.1. Explicación formal del problema

Sea una matriz $M \in \mathbb{R}^{n \times m}$, donde cada posición de la matriz M_{ij} tiene un valor asociado h_{ij}. El problema consiste en calcular el camino mínimo de casilleros desde la posición (1,1) hasta la posición (N,M), donde los únicos dos movimientos posibles son:

Moverse hacia el casillero de abajo:

$$M_{i,j} \rightarrow M_{i+1,j}$$

ó *Moverse hacia el casillero de la derecha*:

$$M_{i,j} \rightarrow M_{i,j+1}$$

Realizar estos movimientos tiene un costo que depende de un parámetro de entrada H:

$$Costo(M_{i,j} \rightarrow M_{i+1,j}) = \begin{cases} 0 & \text{si } |h_{i,j} - h_{i+1,j}| \leq H \\ |h_{i,j} - h_{i+1,j}| - H & \text{caso contrario} \end{cases} \tag{2}$$

$$Costo(M_{i,j} \rightarrow M_{i,j+1}) = \begin{cases} 0 & \text{si } |h_{i,j} - h_{i,j+1}| \leq H \\ |h_{i,j} - h_{i,j+1}| - H & \text{caso contrario} \end{cases} \tag{3}$$

3.1.1. Ejemplos

	1	2	3	4
1	1	0	0	3
2	1	2	1	4
3	1	1	0	1
4	1	2	5	0

Ejemplo con H=0. El costo del camino mínimo resultante es 3.

	1	2	3	4
1	3	1	8	1
2	6	5	1	5
3	2	5	4	7
4	4	2	3	4

Ejemplo con H=2. El costo del camino mínimo resultante es 1.

3.2. Formulación Recursiva

Obtendremos la solución del problema usando un algoritmo de programación dinámica. Para eso, planteamos una formulación recursiva del problema. Sea la función f:

$f(i,j) =$ costo de un camino óptimo desde $M_{1,1}$ hasta Mi,j

Entonces, la solución del problema está dada por $f(n,m)$. Se propone la siguiente recursión para calcular f:

$$f(i,j) = min\begin{pmatrix} f(i-1,j) + Costo(M_{i-1,j} \to M_{i,j}), \\ f(i,j-1) + Costo(M_{i,j-1} \to M_{i,j}) \end{pmatrix} \tag{4}$$

con algunos casos particulares:

$f(1,1) = 0$

$f(1,j) = f(1,j-1) + Costo(M_{1,j-1} \to M_{1,j})$

$f(i,1) = f(i-1,1) + Costo(M_{i-1,1} \to M_{i,1})$

3.2.1. Demostración de Correctitud

El caso base $f(1,1)$ es trivial, ya que no realizo ningún movimiento.

Dada cualquier otra posición (i,j) de la matriz, el camino mínimo para llegar a ella tiene como última posición visitada o la casilla de su izquierda $(i,j-1)$ (si existe), o la casilla de arriba $(i-1,j)$ (si existe) por el enunciado del problema, ya que los únicos movimientos posibles son moverse hacia abajo o hacia la derecha.

Para demostrar que la función f propuesta calcula el camino mínimo hasta una casilla cualquiera $M_{i,j}$ debemos ver que se cumple el **Principio de Optimalidad.** Es decir, que dado un camino óptimo desde $M_{1,1}$ hasta Mi,j, $P_{i,j}$, entonces, el subcamino desde $M_{1,1}$ hasta el inmediato antecesor de $M_{i,j}$ ($P_{i-1,j}$ o $Pi,j-1$) debe ser óptimo:

Sea $P_{i,j}$ el camino óptimo desde $M_{1,1}$ $hasta Mi,j$. Sin pérdida de generalidad puedo suponer que el inmediato antecesor de $M_{i,j}$ en P es $M_{i,j-1}$. Supongamos que el subcamino hasta el antecesor de $M_{i,j}$ ($P_{i,j-1}$) no es óptimo. Entonces, $\exists$ $P\prime_{i,j-1}$ tal que $Costo(P\prime_{i,j-1}) < Costo(P_{i,j-1})$.

Pero entonces, puedo tomar el camino $P\prime_{i,j-1}$ y de ahí moverme a la posición $M_{i,j}$. $Costo(P\prime_{i,j-1}) + Costo(M_{i-1,j} \to M_{i,j}) < Costo(Pi,j-1) + Costo(M_{i-1,j} \to M_{i,j}) = Costo(P_{i,j})$ Pero esto es absurdo, ya que existiría un camino mejor que el óptimo hasta la posición (i,j).

Como aplica el principio de optimalidad y sólo hay dos posibles antecesores para una determinada casilla $M_{i,j}$, para calcular el camino mínimo hasta una posición (i,j) basta con tomar el mínimo entre las dos posibilidades, tomando el cuenta el costo de realizar el último movimiento. Es decir, el de menor costo sumando el costo de dar el último movimiento.

Para los casos borde de la matriz, en la primera columna y en la primera fila, donde sólo tengo un sólo posible antecesor (el inmediato de abajo y el inmediato de la izquierda respectivamente), el camino mínimo entonces es el camino mínimo hasta el único antecesor más el último movimiento. □

3.3. Pseudocódigo

Se presenta a continuación el pseudocódigo del algoritmo propuesto que resuelve el problema. El algoritmo calcula la función recursiva definida anteriormente para todas las posiciones posibles de la matriz, es decir, el costo mínimo para alcanzar cada una de esas posiciones, guardando junto con el resultado la lista de movimientos para llegar a cada una de ellas. Como se explica en la sección anterior, el costo para llegar a una posición determinada (i, j) depende solamente del costo para llegar a las posiciones $(i-1, j)$ y $(i, j-1)$. En los casos particulares de casilleros que se encuentran en la primera columna o en la primera fila de la matriz, los costos dependen solamente del costo para llegar al casillero de arriba o de la izquierda respectivamente. Esto permite implementar un algoritmo **bottom-up**, es decir, podemos definir un orden determinado para calcular los resultados parciales iterativamente, evitando el overhead que resultaría de implementar la función recursivamente (top-down). Entonces, resolvemos primero los subproblemas más chicos y luego los subproblemas más grandes, guardando los resultados en un diccionario. El orden propuesto es resolver por filas, empezando por la fila 1 y terminando por la fila N, de izquierda a derecha:

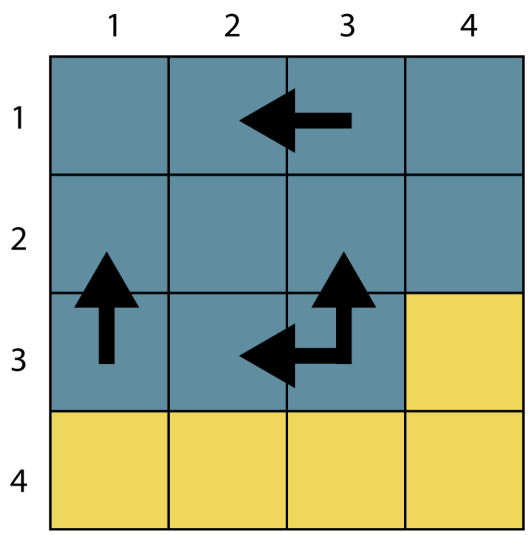


Figura 18: Dependencias entre subproblemas.

Como se ve en la figura 18, al resolver los subproblemas en este orden se respetan las dependencias entre los mismos, ya que al resolver uno ya se tienen calculados los resultados de los subproblemas correspondientes al casillero de la izquierda y al casillero por encima del mismo.

Para representar el diccionario donde se guardan los resultados de los subproblemas, utilizamos una matriz de triplas de tamaño $n \times m$, donde cada tripla es de tipo a : $<$`int`: *Altura*, `int`: *CostoMínimo*, `char`: *Movmimiento*$>$. Los campos corresponden a:

- la *Altura* de cada casillero, ingresada por parámetro
- el *CostoMínimo* para llegar a dicha casilla, que va calculando el algoritmo y guardando en la matriz
- el último *Movimiento* para llegar a dicha casilla por el camino de costo mínimo, donde X corresponde a un movimiento de tipo $M_{i,j} \to M_{i+1,j}$, y Y corresponde a un movimiento

de tipo $M_{i,j} \rightarrow M_{i,j+1}$.

Basta con guardar el último movimiento, ya que para rearmar la lista de movimientos entera, al final podemos recorrer el camino de costo mínimo de casilleros desde la posición (n, m) usando el último movimiento de cada uno de ellos.

Algorithm 5 Pseudocódigo del `main`

```
procedure MAIN
    n, m, h ← INPUT
    vector<vector<Casilla>> Tablero        ▷ creo matriz de n × m        ▷ Θ(n × m)
    InicializarAlturas(Tablero)       ▷ completo las casillas con sus respectivas alturas, Θ(n × m)
    int res ← solve(Tablero, n, m, h)                                   ▷ Θ(n × m)
    list<char> camino                                                   ▷ O(1)
    int i ← n − 1                                                       ▷ O(1)
    int j ← m − 1                                                       ▷ O(1)
    while !(i == 0 && j == 0) do                                        ▷ Θ(n + m)
        const char mov ← Tablero[i][j]{Movimiento}                      ▷ O(1)
        camino.pushfront(mov)                                           ▷ O(1)
        IF mov == Y THEN j − − ELSE i − − FI                            ▷ O(1)
    Imprimir (res, camino)                                              ▷ Θ(n + m)
```

Algorithm 6 Pseudocódigo de función `solve`

```
procedure SOLVE(vector<vector<Casilla>> &t, int n, m, h) → int CostoMínimo
    t[0][0]{CostoMínimo} ← 0                                            ▷ Θ(1)
    for int j desde 1 hasta m − 1 do                                    ▷ Θ(m)
        t[0][j]{CostoMínimo} ← costoColumna(t, h, 0, j)                 ▷ Θ(1)
        t[0][j]{Movimiento} ←'Y'                                        ▷ Θ(1)
    for  int i desde 1 hasta n − 1  do                                  ▷ Θ(n)
        t[i][0]{CostoMínimo} ← costoFila(t, h, i, 0)                    ▷ Θ(1)
        t[i][0]{Movimiento} ←'X'                                        ▷ Θ(1)
    for i desde 1 hasta n − 1 do                                        ▷ Θ(n × m)
        for j desde 1 hasta m − 1 do
            t[i][j]{costoMínimo} ← min{costoFila(t, h, i, j), costoColumna(t, h, i, j)}  Θ(1)
            if t[i][j]{CostoMínimo} == costoFila(t, h, i, j) then       ▷ Θ(1)
                t[i][j]{Movimiento} ←'X'                                ▷ Θ(1)
            else
                t[i][j]{Movimiento} ←'Y'                                ▷ Θ(1)
    return t[n − 1][m − 1]{CostoMínimo}                                 ▷ Θ(1)
```

donde:

- `costoFila`$(t, h, i, j) = t[i-1][j]\{CostoMínimo\} \; + \; Costo(M_{i,j} \rightarrow M_{i+1,j})$
- `costoColumna`$(t, h, i, j) = t[i][j-1]\{CostoMínimo\} \; + \; Costo(M_{i,j} \rightarrow M_{i,j+1})$

Ambas pueden ser calculadas en $O(1)$.

3.4. Complejidad del algoritmo

La complejidad de los algoritmos se puede deducir fácilmente de los pseudocódigos 5 y 6. La función `solve` que resuelve los subproblemas y los guarda en la matriz, es $\Theta(n \times m)$, debido a que siempre tendrá que resolver $n \times m$ subproblemas para obtener el resultado. Luego, recuperar e imprimir la lista de movimientos es $\Theta(n + m)$, ya que un camino desde la posición $(1, 1)$ hasta la posición (N, M) realizado con los movimientos descriptos en secciones anteriores siempre tiene un largo de $n + m$ casilleros. Como $\Theta(n + m) + \Theta(n \times m) + \Theta(1) = \Theta(n \times m)$, concluimos que la complejidad del algoritmo es $\Theta(n \times m)$.

3.5. Performance del algoritmo

Como dijimos antes, la complejidad del algoritmo es siempre $\Theta(nm)$, sin distinción entre casos, por lo que el análisis de performance es simple.

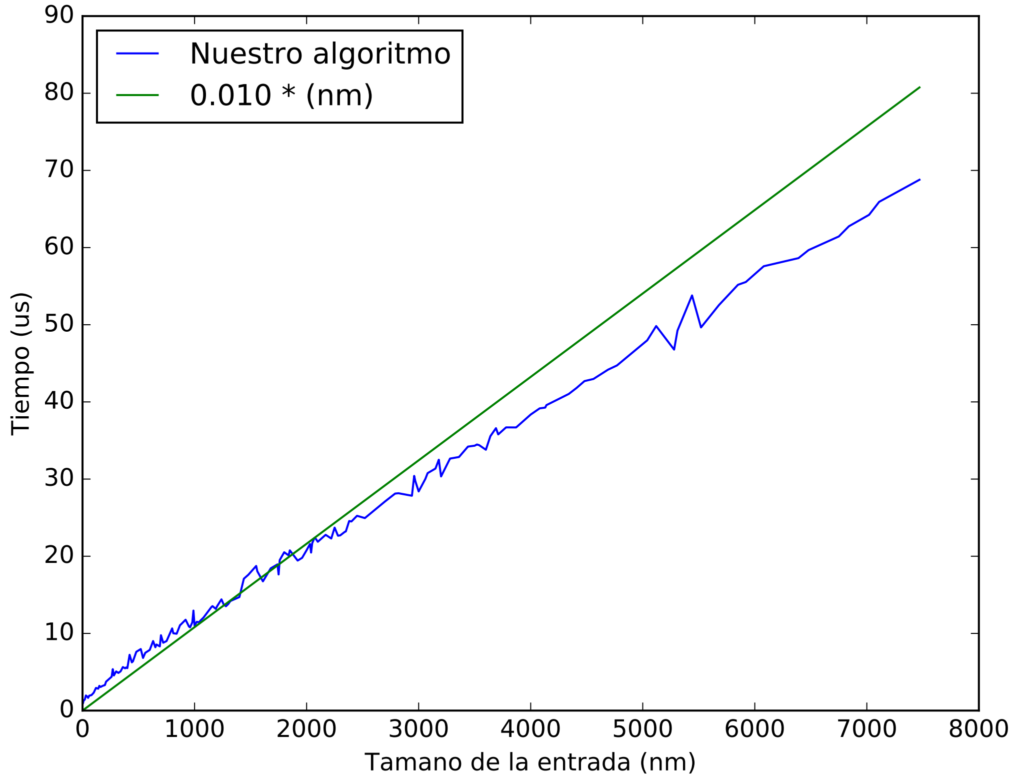


Figura 19: Tiempo que toma el algoritmo en μs para una entrada de tamaño mn.

En esta imagen se ve que se comporta como debe. Sin embargo, al igual que en los problemas anteriores, para confirmarlo totalmente, realizamos el gráfico de $\frac{T(nm)}{nm}$, dado que si esta función tiene a una constante cuando $nm \to \infty$, habremos confirmado experimentalmente que la complejidad del algoritmo es de $\Theta(nm)$.

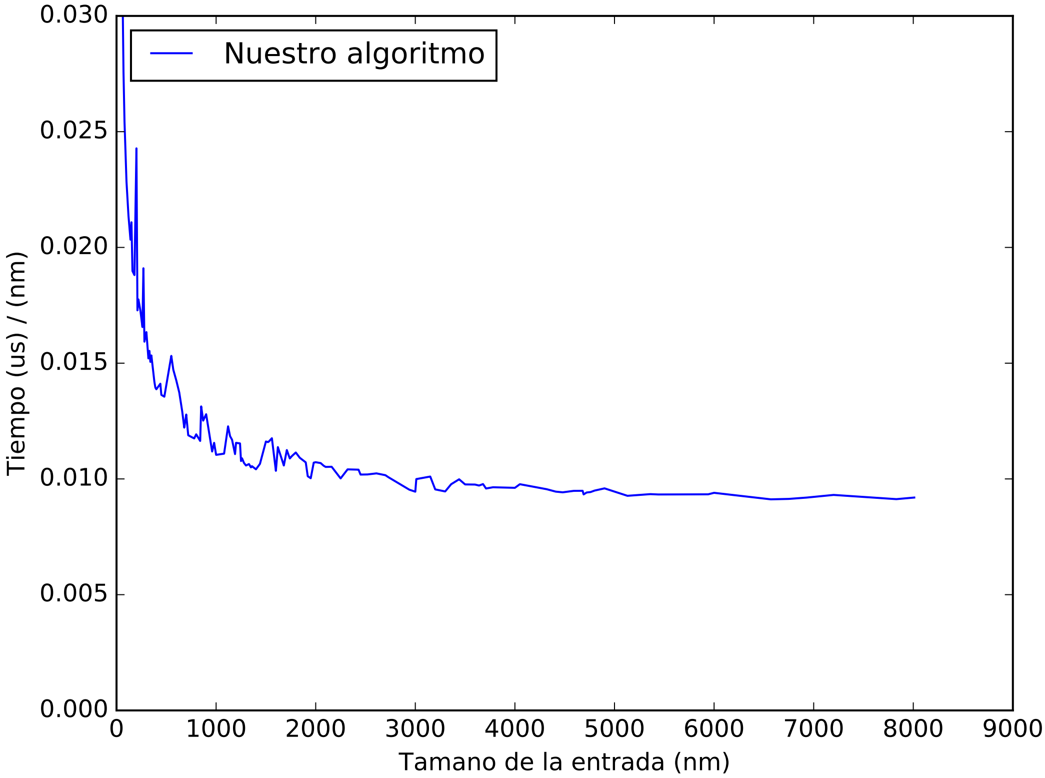


Figura 20: Tiempo que toma el algoritmo en μs dividido mn para una entrada de tamaño mn.

3.5.1. Método de experimentación

Dados n y m, generamos una matriz de $n \times m$, donde cada celda tiene un peso al azar. Para cada par n, m generamos varias matrices (cada una con pesos distintos en cada celda), y tomamos la mediana de esas mediciones.

De todas maneras, la varianza del tiempo para cada matriz de la misma dimensión era casi nula, dado que obviamente el valor de las celdas no afecta el tiempo. Sin embargo, nos parece importante aclarar que esto sucede, y que además fue verificado experimentalmente como dijimos.

4. Apéndice

4.1. Generación de grafos conexos aleatorios

Algorithm 7 Pseudocódigo del procedimiento para generar grafos conexos al azar

```
procedure GRAFO_RANDOM(int n, int m)→ Grafo
    kn ← {(0,1), (0,2), ..., (0,n), (1,2), (1,3), ..., (n−2, n−1)}
    vertices ← {random.range(0,n)}                         ▷ Empiezo con un vértice al azar
    agm ← {}
    while vertices.size() < n do
        aristas ← "aristas (u,v) de kn tal que u ∈ vertices y v ∉ vertices o viceversa"
        arista_nueva ← random.choice(aristas)
        agm.add(arista_nueva)
        kn.remove(arista_nueva)
        vertices.add("extremo de arista_nueva que no estaba en vertices")
                                  ▷ Cuando termina este ciclo tenemos un árbol de n aristas
    grafo ← agm
    while grafo.size() < m do
        arista ← random.choice(kn)
        grafo.add(arista)
        kn.remove(arista)
    for arista ∈ grafo do
        peso(arista) ← random.random()
    return grafo
```

El algoritmo, se basa en generar un grafo conexo minimal (es decir, un árbol) de n vértices. Para lograr esto, técnicamente lo que hacemos es empezar con K_n, es decir, el grafo completo de n vértices, con todos sus aristas de igual peso, y le encontramos un árbol generador mínimo utilizando Prim. Todo esto es obviamente trivial en este caso, dado que todas las aristas tienen igual peso, así que básicamente lo que hacemos es elegir una arista al azar en cada paso.

Luego, una vez que tenemos el árbol terminado, lo completamos con aristas al azar, hasta llegar al objetivo de m aristas.

Finalmente, se eligen pesos al azar para cada arista.

4.2. Partes relevantes del código

4.2.1. problema1.cpp

```

#include <iostream>
#include <queue>
#include <vector>

using namespace std;
typedef int vertice;
typedef vector<vertice> VerticesAdyacentes;
typedef vector<VerticesAdyacentes> ListaAdyacencia;

vector<vertice> bfs(ListaAdyacencia vs, vertice root, vertice target, int n) {
  queue<vertice> c;
  vector<int> distancia(n, -1);
  vector<int> acm(n, -1);
  vertice actual;
  distancia[root] = 0;
  acm[root] = root;
  c.push(root);
  while (!c.empty()) {
    actual = c.front();
    c.pop();
    VerticesAdyacentes vecinos = vs[actual];
    for (const vertice& v : vecinos) {
      if (distancia[v] == -1) {
        distancia[v] = distancia[actual]+1;
        acm[v] = actual;
        if(v == target) break;
        c.push(v);
      }
    }
  }
  //Tengo que reconstruir el camino de atras
  //para adelante
  int long_sol = distancia[target]-1;
  vector<vertice> solucion(long_sol, 0);
  //Arranco en el padre del nodo n-1
  vertice v = acm[target];

  for (int i = long_sol - 1; i >= 0; i--) {
    solucion[i] = v;
    v = acm[v];
  }

  return solucion;
}


int main() {
  int n, m;
  cin >> n >> m;
  vector<vector<int>> input;
  for (int i = 0; i < m; i++) {
    int ai, bi, ei;
```

```
    cin >> ai >> bi >> ei;
    input.push_back(vector<int>({ai, bi, ei}));
  }

  ListaAdyacencia adj_list(3*n, VerticesAdyacentes());
  for (const vector<int>& arista : input) {
    int ai = arista[0], bi = arista[1];
    bool ei = arista[2];
    if (ei) {
      adj_list[ai].push_back(bi + n);
      adj_list[ai + n].push_back(bi + 2*n);
      adj_list[bi].push_back(ai + n);
      adj_list[bi + n].push_back(ai + 2*n);
    } else {
      adj_list[ai].push_back(bi);
      adj_list[bi].push_back(ai);
      adj_list[ai + n].push_back(bi + n);
      adj_list[bi + n].push_back(ai + n);
    }

    adj_list[ai + 2*n].push_back(bi + 2*n);
    adj_list[bi + 2*n].push_back(ai + 2*n);
  }

  vector<vertice> solucion = bfs(adj_list, 0, 3*n-1, 3*n);

  cout << solucion.size() + 1 << endl;
  for (const int s : solucion) {
    cout << s % n << " ";
  }
  cout << endl;
}
```

4.2.2. problema2.h

```
#include <iostream>
#include <utility>
#include <vector>

struct Vertex {
  int key;
  int value;
};

// Para facilitar la lectura. El primer elemento es el otro vertice, y el
// segundo elemento es el costo de esa ruta.
typedef std::vector<std::pair<int, int>> VerticesAdyacentes;

// MinHeap de Vertices. Es la implementacion standard, solo que con algunos
// detalles cambiados. Hubo que agregar el arreglo "pos" que nos permite hacer
// DecValue de manera rapida (logn). No implementamos funciones de insercion
// porque no hacen falta.
class MinHeap {
 public:
  MinHeap(const std::vector<Vertex>& v);

```

```
  inline Vertex Top() const;

  Vertex Pop();

  void DecValue(int i, int new_value);

  inline size_t Size() const;

  inline int At(int i) const {
    return pos[i] >= 0;
  }

  inline int Value(int i) const {
    return a[pos[i]].value;
  }

 private:
  inline int parent(int i) { return (i - 1) / 2; }
  inline int left(int i) { return 2 * i + 1; }
  inline int right(int i) { return 2 * i + 2; }

  template <typename X>
  inline void swap(std::vector<X>& a, int i, int j) {
    X temp = a[i]; a[i] = a[j]; a[j] = temp;
  }

  template <typename X>
  inline bool inbound(std::vector<X>& a, int i) {
    return i < (int) a.size();
  }

  void MinHeapify(int i);

  // Array interno.
  std::vector<Vertex> a;
  // Posiciones de los elementos. Invariante: a[pos[i]].key = i.
  std::vector<int> pos;
};
```

4.2.3. problema2.cpp

```
#include "problema2.h"

#include <cstddef>
#include <limits>

// Funcion para debugear.
void printVertexVector(std::vector<Vertex>& vs) {
  std::cerr << "{";
  for(const Vertex& v : vs) {
    std::cerr << "(" << v.key<< ", " << v.value << "), ";
  }
  std::cerr << "}" << std::endl;
}

MinHeap::MinHeap(const std::vector<Vertex>& v)
```

```
  : a(v), pos(v.size(), 0) {
  for (int i = a.size() / 2; i >= 0; i--) {
    MinHeapify(i);
  }
  for (int i = 0; inbound(a, i); i++) {
    pos[a[i].key] = i;
  }
}

Vertex MinHeap::Top() const {
  return a[0];
}

Vertex MinHeap::Pop() {
  Vertex minimum = a[0];
  // El primero deja de existir.
  pos[a[0].key] = -1;
  // El ultimo pasa a estar arriba de todo.
  pos[a[a.size() - 1].key] = 0;

  // El ultimo pasa estar arriba de todo.
  a[0] = a[a.size() - 1];
  // Achico el vector.
  a.resize(a.size() - 1);

  // Bajo la raiz hasta dejarlo todo bien.
  MinHeapify(0);
  return minimum;
}

void MinHeap::DecValue(int v, int new_value) {
    // Consigo el indice de v.
    int i = pos[v];
    // Le pongo el nuevo valor.
    a[i].value = new_value;

    // Voy para arriba "heapificando" el array.
    while (i > 0 && a[i].value < a[parent(i)].value) {
        // Lo cambio con su padre.
        pos[a[i].key] = parent(i);
        pos[a[parent(i)].key] = i;
        swap(a, i, parent(i));
        i = parent(i);
    }
}

inline size_t MinHeap::Size() const {
  return a.size();
}

void MinHeap::MinHeapify(int i) {
    int smallest = i;
    int l = left(i);
    int r = right(i);

    if (inbound(a, l) && a[l].value < a[smallest].value)
      smallest = l;
    if (inbound(a, r) && a[r].value < a[smallest].value)
```

```
        smallest = r;

      if (smallest != i) {
          // Los vertices que tengo que cambiar.
          Vertex smallest_vertex = a[smallest];
          Vertex index_vertex = a[i];
          // Cambio las posiciones.
          pos[smallest_vertex.key] = i;
          pos[index_vertex.key] = smallest;
          // Cambio los vertices.
          swap(a, smallest, i);
          // Aplico MinHeapify para abajo.
          MinHeapify(smallest);
      }
}

std::vector<std::pair<int, int>> prim(
    MinHeap& heap, std::vector<VerticesAdyacentes> adj_list) {
  std::vector<std::pair<int, int>> parent(heap.Size(), std::make_pair(0, 0));

  while (heap.Size() > 0) {
    Vertex min_vertex = heap.Pop();
    int u = min_vertex.key;

    for (const auto& vertex_and_weight : adj_list[u]) {
      int v = vertex_and_weight.first;
      int weight = vertex_and_weight.second;
      if (heap.At(v) && weight < heap.Value(v)) {
        parent[v].first = u;
        parent[v].second = weight;
        heap.DecValue(v, weight);
      }
    }
  }
  return std::move(parent);
}


int main() {
  int n, m;
  std::cin >> n >> m;
  std::vector<VerticesAdyacentes> adj_list(n, VerticesAdyacentes());

  for (int i = 0; i < m; i++) {
    int ai, bi, li;
    std::cin >> ai >> bi >> li;
    adj_list[ai].push_back(std::make_pair(bi, li));
    adj_list[bi].push_back(std::make_pair(ai, li));
  }

  std::vector<Vertex> v_inicial;
  v_inicial.push_back({.key = 0, .value = 0});
  for (int i = 1; i < n; i++) {
    v_inicial.push_back({.key = i, .value = std::numeric_limits<int>::max()});
  }
  MinHeap h(v_inicial);
  std::vector<std::pair<int, int>> resultado = prim(h, adj_list);

```

```
  int litros = 0;
  for (const auto& x : resultado) {
    litros += x.second;
  }

  std::cout << litros << std::endl;
  for (size_t i = 1; i < resultado.size(); i++) {
    std::cout << resultado[i].first << std::endl;
  }
}
```

4.2.4. problema3.cpp

```
#include <algorithm> // min
#include <iostream>
#include <list>
#include <tuple>
#include <vector>

using namespace std;

// La tupla corresponde a: (altura, costoMinimo, Mov: X o Z)
typedef tuple<int, int, char> Casilla;

// Definiciones
int solve(vector<vector<Casilla>>& t, int n, int m, int h);
int costo (const vector<vector<Casilla>>& t, int h, int i, int j, int k, int l);

int costocolumna(const vector<vector<Casilla>>& t, int h, int i, int j);
int costofila(const vector<vector<Casilla>>& t, int h, int i, int j);

int main() {
  int n, m, h;
  cin >> n >> m >> h;

  vector<vector<Casilla>> Tablero(n, vector<Casilla>(m, Casilla()));

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      cin >> get<0>(Tablero[i][j]);
    }
  }

  int resultado = solve(Tablero, n, m, h);
  // Armo el camino minimo.
  list<char> camino;
  int i = n - 1;
  int j = m - 1;
  while (!(i == 0 && j == 0)) {
    const char mov = get<2>(Tablero[i][j]);
    camino.push_front(mov);
    mov == 'Y' ? j-- : i--;
  }

  cout << resultado << endl;
  for (const char mov : camino)
```

```
    cout << mov << endl;

  return 0;
}


int costo(const vector<vector<Casilla>>& t, int h, int i, int j, int k, int l){
  int delta = abs(get<0>(t[i][j]) - get<0>(t[k][l]));
  return delta <= h ? 0 : delta - h;
}

int costofila(const vector<vector<Casilla>>& t, int h, int i, int j) {
  return get<1>(t[i-1][j]) + costo(t, h, i-1, j, i, j);
}

int costocolumna(const vector<vector<Casilla>>& t, int h, int i, int j) {
  return get<1>(t[i][j-1]) + costo(t, h, i, j-1, i, j);
}


int solve(vector<vector<Casilla>>& t, int n, int m, int h) {
  get<1>(t[0][0]) = 0;

  for (int j=1; j < m; j++){
    get<1>(t[0][j]) = costocolumna(t, h, 0, j);
    get<2>(t[0][j]) = 'Y';
  }

  for (int i=1; i < n; i++){
    get<1>(t[i][0]) = costofila(t, h, i, 0);
    get<2>(t[i][0]) = 'X';
  }

  for (int i = 1 ; i < n; i++){
    for (int j = 1; j < m; j++){
      get<1>(t[i][j]) = min(costofila(t, h, i, j), costocolumna(t, h, i, j));
      if (get<1>(t[i][j]) == costofila(t, h, i, j))
        get<2>(t[i][j]) = 'X';
      else
        get<2>(t[i][j]) = 'Y';
    }
  }

  return get<1>(t[n-1][m-1]);
}
```

Referencias

[Cor+09] Thomas H. Cormen y col. *Introduction to Algorithms, 3rd Edition*. The MIT Press, 2009.