FACULTAD
DE INGENIERIA

Teoría de Algoritmos I - [75.29/95.06]

Trabajo Práctico Nº 2

Grupo MinMax

Alumno	Padrón	Mail
del Mazo, Federico	100029	delmazofederico@gmail.com
Kristal, Juan Ignacio	99779	kristaljuanignacio@gmail.com
Raveszani, Nicole	101031	nraveszani@gmail.com

Índice

1. Spy Vs Spy

Dado un mapa de una ciudad representado como un grafo donde cada arista es un camino entre dos puntos de este, y dada la posición de dos espías y de un aeropuerto, se quiere encontrar que espía llegará antes al aeropuerto. De llegar el espía negro antes que el espía blanco, lo intercepta a este último y será el ganador. De lo contrario, el ganador es el espía blanco, es decir, en el caso de empate también gana el espía blanco, con la idea de que el espía negro no llegó a emboscarlo cuando ambos tienen la misma distancia a recorrer.

1.1. Hallar el Ganador en un Grafo sin Costos

Si la distancia entre dos puntos cualesquiera es la cantidad de aristas a recorrer para llegar de uno a otro, el algoritmo utilizado para obtener la distancia entre los espías y el aeropuerto es el del recorrido Breadth-First sobre el grafo. Este recorrido tiene una complejidad de O(V + A), siendo V la cantidad de vértices y A la cantidad de aristas en el grafo.

Nuestra implementación de BFS utiliza una cola, donde el primer vértice que se almacena es la posición inicial de un espía. Mientras la cola no esté vacía, se retira un elemento en cada iteración y se analizan todos sus vértices adyacentes. Para cada adyacente, se comprueba si fue procesado anteriormente, lo cual significa que en alguna iteración anterior estuvo dentro de la cola y fue procesado. En caso de no haber sido procesado , se lo marca como visitado y se lo encola.

Todos los vértices que fueron visitados poseen sus datos almacenados en distintos diccionarios, cada uno con la función de guardar distintos tipos de información. Los diccionarios utilizados fueron visitados, orden y nivel. Visitados se utiliza para comprobar qué vértice fue analizado, para evitar que un vértice vuelva a ser encolado. Nivel almacena todos los vértices que se encuentran en un determinado nivel del grafo. Por ejemplo, en el nivel 0 se halla únicamente el vértice origen, que es de donde parte BFS a recorrer el grafo, en el nivel 1 están los adyacentes del origen, en el nivel 2 los adyacentes de los adyacentes al origen, y así. En el diccionario orden se almacena qué nivel le corresponde a cada vértice. Estos dos últimos diccionarios, son devueltos por BFS al finalizar.

Para calcular la distancia, se devuelve el orden del vértice donde se encuentra el aeropuerto, ya que el vértice origen es un espía.

1.2. Hallar el Ganador en un Grafo con Costos

Si la distancia entre dos puntos cualesquiera es la distancia euclídea entre ellos, el algoritmo utilizado para encontrar la mínima distancia entre ellos es el algoritmo de Dijkstra. Este algoritmo, con nuestra implementación que utiliza un heap, tiene una complejidad de O(A log(V)).

El heap encola una tupla que contiene la distancia al origen, que inicialmente es cero, y al origen (que se considera que no posee vértice padre). Se utilizan distintos diccionarios para almacenar la información de cada vértice: visitados, padre y distancia. En el diccionario de visitados, al igual que en BFS, sólo se almacenan aquellos vértices que fueron procesados. En el diccionario padre, se tiene al vértice adyacente por donde se halló un camino mínimo al vértice en cuestión. El diccionario de distancia almacena para cada vértice la mínima distancia desde sí

mismo hasta el origen, que es una suma de todas las distancias que se recorrieron para llegar a él desde el origen.

Mientras el heap no esté vacío, se desencola un elemento y se comprueba si fue visitado. En caso de ser la primera vez que se lo desencola, se lo marca como visitado y se analizan todos sus adyacentes.

Al momento de analizar cada adyacente se comprueban tres casos: si jamás se analizó la distancia del camino que se recorrió desde el origen hasta este vértice, si ya se guardó en otra iteración la distancia de un camino posible para llegar a este vértice pero ahora se halló un camino con menos costo, o bien, si el camino ahora hallado posee un costo mayor. En los dos primeros casos, se actualiza la distancia para este vértice (a través del correspondiente diccionario distancia), y se actualiza su padre. Finalmente, se devuelven los diccionarios padre y distancia.

Para calcular la distancia, se devuelve el valor del diccionario de distancia correspondiente al vértice final (aeropuerto), que representa la mínima distancia que se recorrió para llegar desde el origen hasta el final.

1.3. Complejidad Adicional por Devolver Los Caminos

Si ahora se desea obtener el camino entre dos vértices habrá que reconstruir el camino desde el final hasta el origen.

Para el caso donde no hay costos en el grafo, se debe considerar:

- agregar un condicional para verificar si la resolución del problema se pidió sin caminos: O(1);
- agregar un condicional adicional para verificar si el grafo es sin costo (O(1));
- llamar al algoritmo BFS (O(V + A)), una vez por cada espía (difiere en una constante con la complejidad anterior);
- crear una lista vacía para representar al camino recorrido desde el origen hasta el final (O(1));
- recorrer la distancia desde el final hasta el origen (O(distancia[final] * f(n))) con O(distancia[final]) del orden de O(V) (para el peor caso en donde el recorrido implica todos los vértices del grafo) con la complejidad de f(n) dada por:
    - agregar el vértice al final del camino (O(1));
    - Por cada uno de los adyacentes del vértice iterado, encontrar el adyacente que esté en un nivel más abajo del actual (O (adyacentes), que en el peor caso es O(V))
    - Asignar a ese vértice adyacente el nuevo vértice actual O(1)
- invertir el recorrido hallado para mostrarlo desde el principio (O(V))
- agregar dos condicionales para verificar si cada espía tiene un camino posible al aeropuerto (difiere en una constante con la complejidad O(1))
- imprimir el camino que recorrió cada espía. En el peor de los casos, cada uno tiene una longitud de V (todos los vértices), por lo que se tiene una complejidad de O(V).

Para este caso, se tiene que implementar la devolución de caminos no debería tener una complejidad asintóticamente mayor a : O(V + A + V*V)

Para el caso donde el grafo si tiene costos:

- agregar un condicional para verificar si la resolución del problema se pidió sin caminos: O(1);
- agregar un condicional adicional para verificar si el grafo es con costo (O(1));
- llamar al algoritmo de Dijkstra O(A log(V)) 1 vez por cada espía (difiere en una constante con la complejidad anterior);
- crear una lista vacía que representa al camino recorrido desde el origen hasta el final (O(1));
- recorrer el diccionario de padres e ir agregando al camino el padre de cada vértice iterado (O(V) para el peor caso del camino más largo posible);
- invertir el recorrido hallado para mostrarlo desde el principio (O(V));
- agregar dos condicional para verificar si cada espía tiene un camino posible al aeropuerto (difiere en una constante con la complejidad O(1));
- imprimir el camino que recorrió cada espía. En el peor de los casos, cada uno tiene una longitud de V (todos los vértices), por lo que se tiene una complejidad de O(V).

Para este caso, se tiene que implementar la devolución de caminos no debería tener una complejidad asintóticamente mayor a : O(A log(V) + V)

Por ende, la complejidad total no debería ser mayor a : O(A log(A) + V + A + V*V) (multiplicado por una constante arbitraria).

1.4. Devolver Los Caminos

En caso de que se quiera obtener el camino que hizo el espía para llegar al aeropuerto con un grafo no pesado, se reconstruye el camino en base a los diccionarios devueltos por el BFS. A partir del nivel donde se halla el aeropuerto, el cual sería el vértice actual, se lo agrega a la lista que contiene el camino y se busca cual de sus adyacentes se encuentra en un nivel menor. El adyacente que cumple con esta condición se convierte en el actual y se realiza el mismo proceso hasta llegar al nivel inicial. Al final, se inserta el origen al final de la lista y se devuelve la lista invertida.

Para el caso de un grafo pesado, también es necesario reconstruir el camino. Se almacena el vértice "final", que es el vértice donde se encuentra el aeropuerto, en una lista, y se comprueba quien es su padre. El mismo proceso se realiza con este nuevo vértice (el padre del anterior), hasta llegar al origen. Al final se devuelve la lista invertida.

2. Ejercicios de Reducciones

2.1. Ejercicio 1

2.1.1 Solución con Algoritmo de Fuerza Bruta

La versión por fuerza bruta del algoritmo realiza una iteración sobre una de las dos cadenas, rotando la misma un carácter a la vez, y en cada iteración compara la rotación realizada con la otra cadena. Según las complejidades de las sentencias consideradas en el anexo de la sección 3, el algoritmo diseñado es de orden cuadrático. Más precisamente, con n la cantidad de caracteres de las palabras, tiene un orden de complejidad temporal de:

$$O(n) = 3n^2 + 3n + 9$$

2.1.2. Solución con Algoritmo KMP

Optando por KMP, se utiliza dicho algoritmo para hallar, dadas dos cadenas A y B, hasta que i-ésimo carácter (de izquierda a derecha) A se encuentra contenido en B, ejecutando KMP sucesivamente. Podría darse el caso de que A y B coinciden en todos sus caracteres, en ese caso, se devuelve verdadero ya que ambas cadenas son iguales. En otro caso, existe un conjunto de caracteres de A que se encuentran contenidos en B, pero dicho conjunto no es igual a A. Entonces, se utiliza el algoritmo KMP por última vez para hallar si los caracteres restantes de A (a partir del i-ésimo donde no hubo match) están contenidos en B. Si se da el caso, se devuelve verdadero, ya que esto implica que B es una concatenación del conjunto de caracteres inicial y los caracteres restantes, pero en el orden inverso en que se encuentran en A.

Por lo anterior, se encontró un algoritmo Z que resuelve el problema de decidir si dos cadenas son rotación cíclica utilizando KMP. Verificamos que el problema que resuelve KMP sea una reducción del problema del enunciado. Sea X el problema de determinar si dos cadenas son rotación cíclica y sea Y el problema de hallar si una palabra (patrón) se encuentra contenida en otra (cadena), ambos problemas de decisión (basta devolver verdadero o falso para resolverlos). Sean A y B cadenas. Se quiere averiguar si A está contenida en B, es decir, resolver Y con A y B. Si A tiene más caracteres que B, luego A no está contenida en B (esto es una verificación polinómica), por lo que se devuelve falso. En otro caso, sea p la longitud de A, q la longitud de B y r = q - p. Se toman los primeros r caracteres de B, y se forma la cadena C. Luego se forma la cadena D como concatenación de las cadenas A y C (en ese orden), y se llama una instancia de Z con los parámetros B y D. Si Z devuelve verdadero, entonces A estaba contenida en B, ya el retorno de Z prueba que C estaba compuesto por los caracteres de B que no se encontraban en A (y estaban en el orden en que aparecen en la palabra B).

$A = a_1 \dots a_p$
$B = b_1 \dots b_p \dots b_q$
$C = b_1 \dots b_r$
$D = a_1 \dots a_p + b_1 \dots b_r$

$D = A + C$ (lease + como "concatenado con") y $B = C + b_{r+1} \dots b_q$
Si D y B son rotación cíclica, debe ser $A = b_{r+1} \dots b_q$, por lo que A está incluída en B.

Ahora, si Z devuelve falso en la anterior iteración, se toman los primeros r caracteres de B salteando el primero, y se vuelve a llamar a Z con B y la nueva D.

$A = a_1 \dots a_p$
$B = b_1\ b_2 \dots b_p \dots b_q$
$C = b_2 \dots b_{r+1}$
$D = a_1 \dots a_p + b_2 \dots b_{r+1}$

Si D y B son rotación cíclica, debe ser $A = b_{r+2} \dots b_q + b_1$

Y así sucesivamente mientras Z devuelve falso, se realiza una nueva iteración con los respectivos caracteres de r que correspondan a la misma (salteando desde el segundo caracter de B, salteando desde el tercer caracter de B, etc.) hasta que no se puedan saltear más caracteres de B (es decir, luego de la iteración donde $C = b_q + b_1 \dots b_{r-2}$), y si se llega a ese caso, entonces A no estaba contenida en B, por lo que se devuelve falso.

$A = a_1 \dots a_p$
$B = b_1 \dots b_i \dots b_p \dots b_q$
$C = b_i \dots b_{r+i-1}$ (si r + i - 1 > q , entonces $C = b_i \dots b_q + b_1 \dots b_{mod(r+i-1,q)}$)
$D = a_1 \dots a_p + b_i \dots b_{r+i-1}$ (o bien $D = a_1 \dots a_p + b_i \dots b_q + b_1 \dots b_{mod(r+i-1,q)}$)
SI D y B son rotación cíclica, debe ser $A = b_{r+i} \dots b_q + b_1 \dots b_{i-1}$ (o bien $A = b_{mod(r+i-1,q)+1} \dots b_{i-1}$)

(con i de 1 a q)

Luego, a partir de que se mostró como resolver una instancia arbitraria del problema Y (es decir, para A y B genéricos) a partir de una secuencia de llamadas del algoritmo que resuelve X, sumando a un conjunto de operaciones polinómicas (iteraciones para llamar a Z y comparaciones), se puede ver que la complejidad de Y está acotada por la complejidad de X a través de una transformación polinómica, por lo que Y es una reducción de X.

Considerando que se utiliza sucesivamente el algoritmo de matching KMP, y de la bibliografía se conoce que el orden de complejidad amortizado del mismo es lineal con respecto de la longitud de la cadena parametrizada (igual a la cantidad de letras de las palabras que se quiere verificar si son rotación cíclica), la complejidad resultante para esta variante es:

$$O(n) = n^2 + 4n + 9$$

2.1.3. Solución Mejorada

Una mejor forma para resolver el problema de hallar si dos cadenas A y B son rotación cíclica es considerar una cadena C que contenga todas las rotaciones posibles de B (o bien, de A) y verificar si A (análogamente, podría ser B) está contenida en C. Pero hallar C es simplemente concatenar B con sigo misma, ya que todas las llamadas rotaciones de B son en realidad concatenaciones de dos subcadenas contiguas de B, y unir B consigo misma contiene todas esas concatenaciones. Más formalmente sean X e Y dos subcadenas contiguas de B tales que la concatenación de X con Y forma B, lo cual se denota X + Y = B. Se dice entonces que A es una rotación cíclica de B si A = Y + X. Como, B + B = X + Y + X + Y, entonces B + B = X + A + Y, por lo que para cualesquiera X e Y, A es una rotación cíclica de B si A está contenida en B + B.

Esto permite resolver el problema del enunciado simplemente llamando una vez al algoritmo KMP con B + B como cadena y A como patrón, y en el caso de que el mismo devuelve verdadero, se tendrá que A y B son rotación cíclica. En este caso, como la llamada del algoritmo KMP es una sola, la complejidad del mismo es O(n + m), com m la longitud del patrón, y la complejidad total

está dada por:

$$O(n) = 5n + 3$$

2.2. Ejercicio 2

El problema del enunciado puede abstraerse en un grafo: los vértices son las ciudades a las que hay que viajar, y las aristas indican la posibilidad de viajar de una ciudad a otra (y entonces el grafo es conexo, porque se puede llegar a cualquier ciudad, no dirigido, porque nada impide que el viaje de una ciudad a otra no se hubiera podido hacer al revés, y pesado porque las distancias entre ciudades no son iguales). Como la cantidad de millas es igual a la distancia para cualquier par de ciudades, el problema del enunciado consiste en hallar el ciclo hamiltoniano entre los n vértices del grafo que maximiza la distancia a recorrer entre cada par de ciudades.

Se ve en principio que este problema no está en NP: aunque tuviéramos una solución posible (la sucesión de ciudades que forman el ciclo hamiltoniano) podríamos verificar que es un ciclo hamiltoniano polinomialmente (que se visitan todas las ciudades una vez y que se vuelve al inicio), pero no podríamos verificar que este ciclo es el que maximiza la distancia, ya que esto implicaría buscar todos los ciclos hamiltonianos posibles, comparar la distancia total recorrida y verificar que la solución dada tiene la máxima distancia posible. Pero buscar un ciclo hamiltoniano es un problema NPC, por lo que está verificación excede la complejidad polinomial.

Adicionalmente, se ve que este problema es similar al Problema del Viajante de Comercio (Traveling Salesman Problem, o TSP), solo que la optimización es para maximizar distancias y no para minimizarlas. Considerando el problema de decisión de TSP (en adelante TSPD), en el que se da un parámetro K y se pregunta si existe un ciclo hamiltoniano de longitud total menor o igual a K, se conoce que constituye un problema NPC. Entonces, como el problema del enunciado (en adelante X) no es NP, si se puede hallar una reducción de TSPD a X, entonces quedará en evidencia que X es un problema NP-Hard, ya que es al menos tan difícil como un problema NPC pero no es NP (esto último permite excluir la posibilidad de que sea NPC).

Se define en primer lugar el concepto de bifurcación. Se dirá que un grafo tiene una bifurcación cuando al llegar a un vértice, se puede salir del mismo por más de una arista que no se recorrió anteriormente:

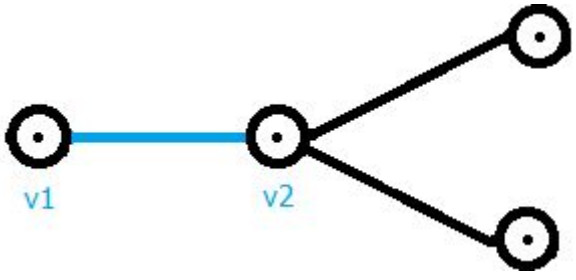

Fig. 1: ejemplo de bifurcación con v2,
siendo la arista en azul una ya recorrida.

Notar que la imagen representa sólo un caso posible: de acuerdo a la definición dada, podría haber "bifurcaciones" de más de dos aristas posibles para salir de un vértice.

Se demuestra que hallar cuáles vértices de un grafo G no dirigido, conexo y pesado tienen una bifurcación es un problema polinomial. Se recorre G con DFS (algoritmo eficiente para el recorrido de un grafo), guardando los vértices y las aristas visitadas. Si el vértice al que se llega no fue visitado y tiene más de una arista no visitada, se guarda el dato de que ese vértice tiene una bifurcación, así como también los pesos de las aristas que constituyen la bifurcación. Se toma uno de los caminos posibles, marcando el vértice abandonado y la arista por la cual se realizó el avance como como visitados (comportamiento de DFS) y se continúa recorriendo el grafo hasta que termine DFS. Este algoritmo es simplemente un recorrido DFS al que se le agrega la funcionalidad de guardar más información, sin modificar en ningún aspecto el recorrido. Guardar los pesos de las aristas de bifurcación y verificar si un vértice tiene una bifurcación son operaciones polinomiales, ya que su complejidad está dada por la cantidad de aristas del vértice analizado. En el peor caso, guardar la información de las aristas de bifurcación de un vértice es O(V), si es que ese vértice está conectado con todos los demás, por lo que la complejidad de este algoritmo es: O(V*V + A), siendo la complejidad de DFS O(V+A) y siendo la complejidad de guardar todas las aristas de bifurcación de todos los vértices O(V*V). Se llamará **Z_bifurcaciones** al algoritmo que permite resolver el problema indicado.

Ahora, supongamos que se tiene un algoritmo **Z_enunciado** para resolver X (devolver el ciclo hamiltoniano que maximiza las distancias entre vértices) y un grafo genérico G pesado, no dirigido y conexo. Primero se ejecuta Z_bifurcaciones en G. Si se recorre el resultado del mismo (operación polinomial, ya que no es asintóticamente más complejo que obtener ese resultado) y se ve que ningún vértice tiene bifurcaciones, entonces se devuelve el resultado de comparar la distancia total del lo que resulta de aplicar Z_enunciado sobre G, con el K de TSPD (la comparación entre esos dos valores es una operación polinomial), devolviendo verdadero en caso de que esa distancia sea menor o igual a K , o falso en caso contrario (o bien si Z_enunciado muestra que no había un ciclo hamiltoniano posible) ya que si hay un ciclo hamiltoniano, es único (no hay bifurcaciones, y el grafo es conexo).

Ahora, en caso contrario, dado un vértice con bifurcaciones, se crea un grafo auxiliar G2, que es igual a G con la diferencia de que se le saca una de las aristas de la bifurcación (operación polinomial: crear otro grafo con la misma cantidad de vértices y un arista menos). Luego se recorre G2 con Z_enunciado y se guarda la distancia total obtenida, si es que fue posible formar un ciclo hamiltoniano. Luego se crea un grafo G3 igual a G pero se quita otra de las aristas de la bifurcación encontrada en el vértice, y se vuelve a guardar el resultado de la distancia total de recorrer G3 con Z_enunciado, si es que fue posible lograr un ciclo hamiltoniano. Así sucesivamente, por cada bifurcación del vértice, se crea un grafo quitando alguna de las aristas de la bifurcación, y se guarda la distancia total resultante de aplicar Z_enunciado sobre ese grafo. Cuando se han recorrido todos los grafos auxiliares para cada arista de bifurcación del vértice en cuestión, se toma la arista con la cual la distancia recorrida fue menor (habiendo ciclo hamiltoniano para esa arista), y se remueven el resto de G (operación polinomial, ya que es quitar una cantidad de aristas del grafo, en el peor caso tiene una complejidad de O(A)).

Eso se repite por cada vértice que tenga bifurcaciones (en el peor caso, todo el resto, pero esa es una serie de iteraciones en las que se llama Z_enunciado acotada por la cantidad de aristas, por

lo que es una operación polinómica la de la iteración). En algún punto, se habrán quitado todas las bifurcaciones de G, y el resultado será un grafo G con un recorrido que minimiza la distancia total recorrida entre todas las ciudades. De aquí, basta devolver verdadero si para la distancia total final del resultado de aplicar Z_enunciado en la última forma de G, dicha distancia es menor o igual al K de TSPD, ya que ese grafo tiene un único ciclo hamiltoniano que minimiza la distancia total.

Como se fue aclarando, todas las operaciones que integran la transformación son polinomiales, excepto por la propia llamada a Z_enunciado, que resuelve un problema que no es NP. Como TSPD se puede resolver con una serie de operaciones polinomiales (incluyendo Z_ Bifurcaciones) y una serie de llamadas al algoritmo que resuelve X (en el peor de los casos, se lo llama A veces), entonces la complejidad de TSPD está acotada por la complejidad de X, y X es un problema NP-Hard, por ser al menos tan difícil como un NPC, y no ser NP.

2.3. Ejercicio 3

2.3.1. Algoritmo Polinomial

Dada una lista de cursos:

```
ordenar_cursos_por_horario_fin(cursos) # Devuelve una lista enlazada para
mejorar eficiencia al remover.
aulas = 0
while cursos:
      anterior = curso_vacio()
      for curso in cursos:
            if not anterior.overlap(curso):
                  cursos.remover(curso)
                  anterior = curso
      aulas+=1
return aulas
```

Para mostrar que es eficiente, se considera:

- Ordenar los cursos por su horario de finalización es de orden O(nlog(n)) con n la cantidad de cursos, ya que es un ordenamiento en base a un solo dato (la fecha de finalización) y se asume la menor complejidad para ordenar.

- Verificar si hay solapamiento u overlap entre cursos es una operación de tiempo constante ya que solo basta comparar los momentos de inicialización y finalización.

- En el peor de los casos, todos los cursos se superponen entre sí (todos tienen el mismo horario de inicio y fin), y en ese caso el while cursos se ejecuta n veces (creándose n aulas), por lo que el algoritmo tiene orden cuadrático, y como es polinómico, se considera eficiente.

Ahora, basta verificar que la cantidad de aulas devueltas es mínima. Supongamos que no lo fuera, y entonces se puede tener un aula menos y lograr que no haya horarios solapados para

ningún par de cursos. Pero esto es decir que, en el marco de la solución dada, se "pidió" una aula adicional para separar dos cursos C1 y C2 cuyos horarios no se superponen, y esto es absurdo porque el criterio para agregar un aula se da por el solapamiento de horarios. Como el absurdo proviene de suponer que la cantidad de aulas no es mínima, entonces la cantidad de aulas debe ser mínima.

2.3.2. Reducción a Coloreo de Grafos

Se conoce que el problema del Coloreo de Grafos en su versión de optimización (en adelante CDG) es un problema NPH. Ahora sea el problema del enunciado (en adelante Y), el cual se sabe que es un problema en P, ya que se encontró una solución eficiente al mismo. Se pide reducir Y a CDG.

Para esto, considérese el algoritmo Z que resuelve CDG, es decir, que devuelve para un mapa dado un coloreo que minimiza la cantidad de colores utilizados. Entonces, se puede transforma la lista de cursos en un mapa (grafo), con una operación polinómica que por cada curso crea una región en el mapa (un vértice en el grafo), y se mantiene en algún tipo de estructura auxiliar la asociación entre el curso y su región correspondiente (lo cual también es una operación polinómica, ya que solo hay que guardar 2n datos, donde n es la cantidad de cursos). A continuación, se crea una arista entre las regiones que representan cursos que se superponen (lo cual se puede hacer verificando en la estructura auxiliar los cursos que se superponen, en tiempo cuadrático, y por cada caso de superposición, agregar una arista entre las regiones correspondientes a los cursos). Ahora, sea G un grafo no dirigido y no pesado que representa el mapa creado. Las regiones vecinas, es decir, aquellas unidas por una arista, no se pintarán del mismo color cuando se llame a Z con G. En particular, aquellos cursos que se superponen se pintarán con colores distintos, pero Z permite hacer esto con la mínima cantidad de colores posibles, por lo que la cantidad de colores utilizados en el mapa coloreado que devuelve Z es también la cantidad de agrupaciones de cursos con horarios superpuestos, que coincide con la cantidad de aulas, por lo que basta contar esa cantidad de colores utilizados en el mapa (operación polinomial, ya que implica recorrer cada vértice del grafo mapa) para resolver el problema Y. Como esto se pudo hacer a partir de una serie de operaciones polinómicas y una llamada a un algoritmo que resuelve CDG, se puede afirmar que Y es una reducción de CDG.

2.3.3. ¿P = NP?

En base a los puntos anteriores, lo que se logró demostrar es que la complejidad de CDG está acotada inferiormente por la complejidad de Y: dicho de otra manera, que CDG es al menos tan difícil como Y. Esto es coherente con la información con la que se contaba al principio: Y pertenece a P y CDG pertenece a NPH, por lo que la demostración dada muestra un caso particular de que un problema NPH es al menos tan difícil como uno P (siendo que esto se cumple en general por definición de NPH). Esta reducción no nos brinda información como para garantizar lo que se pide en la consiga. Si, por el contrario, la demostración hubiera sido de que la complejidad de un problema P está acotada inferiormente por la de un problema NPH, entonces se habría demostrado que existe una reducción para dicho problema "dificil" que permite resolverlo de forma eficiente, a partir de lo cual se podría generalizar que cualquier problema NPC tiene un algoritmo que lo soluciona eficientemente, y entonces, cualquier problema NP tendría una solución eficiente, por lo que P sería igual a NP.

3. Anexo: Órdenes de Complejidad Temporal de Sentencias

- Asignación de una variables: O(1)
- Uso de operador lógico u aritmético: O(1)
- Obtener la longitud de un arreglo: O(1)
- Trabajar con un ciclo: O(n . f(n) + g(n)), donde f(n) es una función para la complejidad temporal de lo que implica la ejecución del ciclo, n representa la cantidad de iteraciones, y g(n) es una función de complejidad temporal para la sentencia inicial del ciclo. Esta última suele ser 1, aunque puede variar dependiendo de si se usan varios operadores lógicos o asignaciones para establecer el ciclo.
- Hallar el índice de un elemento en un arreglo: O(n)
- Borrar un elemento de un arreglo: O(n)
- Comparar dos cadenas: O(2n)

Ante un condicional, siempre se considera el peor caso para el cálculo.
Con las cadenas, los operadores lógicos o aritméticos tienen complejidades temporales proporcionales a las longitudes de las cadenas en los operandos, dependiendo el caso.

Para conocer la complejidad temporal de las sentencias utilizadas en python se consultó: https://wiki.python.org/moin/TimeComplexity

4. Anexo: Procedimientos para la Ejecución y Compilación

3.1. Compilación

Como el trabajo fue realizado en python, no debe ser compilado. La versión de Python usada fue 3.6.1, para la arquitectura 64 bits (el código podría llegar a ser compatible con versiones previas de python 3, pero se recomienda utilizar la indicada).

3.2. Ejecución Spy Vs Spy

Para ejecutar el programa hay que tener generado un mapa, que se puede crear especificando su largo su ancho y el porcentaje de carga. Por ejemplo, para un mapa de 10x10, un 100% cargado (es decir, grafo conexo):

$ python ManejoDeArchivos.py 10 10 100

Con el mapa ya generado, el programa se ejecuta especificando los números de línea de los cuales se obtienen las posiciones los espías (comenzando desde 0). Por ejemplo, en un mapa con al menos 10 aristas:

$ python main.py 5 7 9 → Se asignan al primer vértice en las líneas 5 7 y 9 las posiciones del espía blanco, el espía negro y el aeropuerto, respectivamente.
Notar que las líneas deben ser válidas con respecto al mapa creado.

Las diferentes variantes del programa son ejecutadas con los distintos argumentos por línea de comando:

- $ python main.py 5 7 9 --pesado → Se corre el programa pero asignándoles pesos a las aristas (el peso es la distancia euclídea entre los dos puntos)
- $ python main.py 5 7 9 --sin-camino → Se corre el programa pero no se calcula el camino, solamente se anuncia al ganador de los espías.
- $ python main.py 5 7 9 --pesado --sin-camino → Se pueden combinar ambos parámetros.