

75.29 - Teoría de Algoritmos I: Trabajo Práctico n. 2

Equipo Q:
Lavandeira, Lucas (#98042)

Rozanec, Matias (#97404)
rozanecm@gmail.com

Sbruzzi, José (#97452)

14.mayo.2018



Facultad de Ingeniería, Universidad de Buenos Aires

Índice

I	Resolución	3
1.	Parte 1: Spy vs Spy	3
1.1.	Caminos sin costos	3
1.1.1.	Complejidad del algoritmo para caminos sin costos	3
1.2.	Caminos con costos	3
1.2.1.	Complejidad del algoritmo de caminos con costos	3
1.3.	Instrucciones para acceder al código y ejecutarlo	4
1.3.1.	Preparación	4
1.3.2.	Ejecución	4
2.	Parte 2	5
2.1.	Ejercicio 1	5
2.1.1.	Conclusiones	6
2.2.	Ejercicio 2	7
2.3.	Ejercicio 3	8
2.3.1.	Pseudocódigo	9
2.3.2.	Función variante	11
2.3.3.	Reducción a problema de coloreo de grafos	11
2.3.4.	¿P=NP?	11

Parte I

Resolución

1. Parte 1: Spy vs Spy

1.1. Caminos sin costos

Para resolver el problema partimos de la idea de que el espía que está mas cerca del aeropuerto se queda con los documentos. Así, para resolver el problema con costos unitarios, se decidió hacer una versión modificada de BFS, partiendo desde el aeropuerto. Así, el algoritmo a medida que se ejecuta descubre capas.^{en} el grafo, donde cada capa está a una distancia mayor del aeropuerto que la anterior. El algoritmo se detiene cuando descubre una capa que contiene al menos un espía. De esta forma determina el más cercano, o indica que ambos están a la misma distancia.

1.1.1. Complejidad del algoritmo para caminos sin costos

En el archivo `problema.clj` se abstraigo lo que tienen en común las dos situaciones a resolver: que la solución se realiza con la estrategia greedy. Así, `solucion-greedy` se llamará una vez para cada una de las alternativas que surjan. Debido a la naturaleza de Breadth-First-Search (`bfs.clj`), podemos determinar que esta función se ejecutará, como mucho, una vez por cada vértice.

Los pasos `terminado?`, `conclusion` son $O(1)$ si se considera que el tamaño de la lista `hasta` es constante, lo cual es verdadero para el problema en cuestión. Se utilizan los conjuntos y diccionarios del lenguaje para brindar acceso en tiempo constante. Sin embargo, la función `alternativa` tiene una duración que crece linealmente con la cantidad de `vecinos` que tenga el vértice `actual`. Si `alternativa` se ejecuta para todos los vértices del grafo, la cantidad acumulada de vecinos será el doble de la cantidad de aristas del grafo. Así, el algoritmo implementado tiene una complejidad temporal igual a la de BFS: $O(-V-+-E-)$.

Agregando la etapa en que se retorna el camino recorrido por cada espía, la complejidad sólo puede llegar a $O(-V-+-E-)$, ya que durante el algoritmo se calculan todas las distancias necesarias para alcanzar los espías desde el aeropuerto: El algoritmo que forma este camino sólo tiene que recorrer el grafo desde los puntos de partida de cada espía eligiendo como siguiente nodo el vecino con menor distancia al aeropuerto. La cantidad máxima de nodos a verificar es $-V-$ y la cantidad de vecinos revisados acumulados es $-E-$.

Por lo tanto, incorporando esta etapa, la complejidad temporal sigue siendo $O(-V-+-E-)$.

1.2. Caminos con costos

El algoritmo aprovecha la abstracción `Problema` discutida anteriormente, para describir un algoritmo similar a Dijkstra. El algoritmo consiste en realizar Dijkstra teniendo como punto de inicio el aeropuerto y terminando la ejecución cuando se fijan los valores de las distancias entre cada uno de los espías y el aeropuerto.

1.2.1. Complejidad del algoritmo de caminos con costos

En el archivo `dijkstra.clj` puede observarse la descripción del algoritmo. También se aprovecha la abstracción `Problema` descripta anteriormente. Nuevamente, al igual que para `bfs`, se tiene un conjunto de vértices visitados, de manera que un vértice que pertenece a tal conjunto no será analizado nuevamente, su distancia al aeropuerto queda fija. Así, al igual que con BFS, `alternativa` se ejecuta como mucho una vez por cada vértice.

La determinación de las variables `vecinos-no-visitados` y `distancias-actual-vecino` toma un tiempo que depende linealmente de la cantidad de vecinos que tenga el vértice en cuestión, con lo cual, si se llama `alternativa` para cada vértice, estas toman un tiempo total máximo $O(-E-)$. La complejidad temporal de determinar `actual` es $O(\log -V-)$ porque `visitable` del record `estado-dijkstra` (en el archivo `estado-dijkstra.clj`) es un `sorted-map`, que es representado internamente con una estructura de heap. Lo mismo sucede para el llamado de

e/sin-siguiente-a-visitar: se llama a **e/siguiente-a-visitar** y luego se crea una versión modificada del **sorted-map**, lo cual también es $O(\log V)$.

Sin embargo, el llamado a **e/distancias-a-traves-de** tiene una complejidad mayor que los anteriores. Esta función llama una vez por cada vecino del vértice **actual** a **distancias-a-traves-de-uno**. Esta función llama, a su vez, a las funciones **sin-vertice** y **con-vertice**, ambas crean versiones modificadas del **sorted-map**, lo cual es $O(\log V)$. Como **distancias-a-traves-de-uno** se puede ejecutar como mucho $2E$ veces (una vez por cada vecino de **actual**), tenemos que ejecutar el algoritmo propuesto toma un tiempo $O(V + E \log V)$.

Añadir el cálculo de los caminos de cada espía funciona de manera idéntica que con el algoritmo para el grafo con pesos unitarios: La etapa tendrá complejidad $O(V + E)$, con lo cual el tiempo de ejecución total del algoritmo no cambia.

1.3. Instrucciones para acceder al código y ejecutarlo

1.3.1. Preparación

Primero es necesario instalar leiningen para poder acceder al comando **lein**. Se provee el código fuente en un .zip por mail, sin embargo, también se encuentra disponible en github.com/jisbrpzzi/spies-algo. Se recomienda esta última opción.

1.3.2. Ejecución

Los comandos para ejecutar el proyecto son:

- **lein bfs** ejecutar de forma interactiva el algoritmo para grafos sin pesos
- **lein dijkstra** ejecutar de forma interactiva el programa para grafos con pesos
- **lein test** ejecutar los test

2. Parte 2

2.1. Ejercicio 1

El problema a resolver es uno de detección de rotaciones cíclicas en dos strings (cadenas de caracteres). Un string es una rotación cíclica de otro cuando, al “conectar” el principio y el fin de cada una, se tenga como resultado ciclos indistinguibles uno del otro. Por ejemplo, el ABCD y CDAB son rotaciones cíclicas uno del otro, porque los ciclos ABCDABCDABCD... y CDABCDABCDAB... son indistinguibles si no sabemos su punto de inicio. El problema puede ser caracterizado como uno de decisión: simplemente queremos saber si se cumple la condición pedida o no, es decir, los algoritmos a implementar simplemente devuelven un valor booleano, verdadero o falso. Se pide resolverlo usando distintos algoritmos y comparar sus órdenes de tiempo de ejecución.

La primera solución a implementar es por fuerza bruta. Si asumimos que la comparación de strings es de orden lineal, ir haciendo la rotación de uno de los dos strings y comparando hasta llegar a una igualdad da como resultado un algoritmo de orden cuadrático en el peor caso, cuando se prueban todas las rotaciones y la última (o ninguna) comparación da verdadera.

La segunda solución es implementando el algoritmo KMP de *string matching*. El algoritmo se puede resumir en una forma optimizada de decidir si una palabra *word* está contenida dentro de un texto fuente *text* en orden lineal, realizando un precómputo de una tabla de valores que asisten al algoritmo durante la ejecución. El caso de uso común de KMP es para ubicar una única palabra en un texto grande, mientras que nuestro problema es diferente: queremos decir si un string es una rotación cíclica de otro. Lo que hacemos es ejecutar el algoritmo bajo un ciclo *for*, rotando uno de los strings entre cada iteración. El algoritmo KMP es de orden lineal de la longitud del texto, que para nuestro caso es el mismo que el de la palabra.

Esta aplicación se puede ver como una *reducción* de un problema a otro. Las reducciones son una herramienta que nos permite obtener información de la complejidad de un algoritmo desconocido, basándonos en uno conocido. Llamemos al algoritmo que no conocemos como X, y al que sí, como Y. Si podemos resolver X utilizando a un algoritmo que resuelve Y como una caja negra, utilizándola una cantidad polinomial de veces, se pueden establecer relaciones entre las complejidades de ambos problemas.

En particular, la reducción que nos interesa es X siendo el problema de rotaciones cíclicas, Y el de string matching, y nuestra caja negra es el algoritmo KMP. Al ejecutar el algoritmo unas N veces en el peor caso (siendo N la longitud del string), se puede decir que el problema de rotaciones cíclicas es reducible de manera polinomial al problema de string matching que resuelve KMP. Acabamos de diseñar un algoritmo de resolución al primer problema como una cantidad de llamadas polinomial a una caja negra. La principal conclusión que podemos sacar de esto es muy intuitiva: la complejidad de string matching es, a lo sumo, N veces la complejidad de la caja negra, porque encontramos un algoritmo que lo resuelve con exactamente ese orden temporal. Siendo el algoritmo KMP de complejidad lineal, esta solución tiene una complejidad cuadrática.

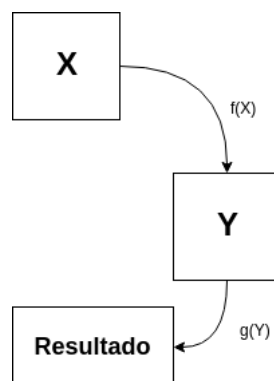


Figura 1: Reducciones

Este uso de KMP es lejos de ser óptimo, no obtuvimos una complejidad mejor a la fuerza bruta, y en ejecuciones reales los cálculos adicionales de la tabla asistente que hace KMP para lograr un

orden lineal ralentizan la ejecución suficientemente para que sea un orden entero más lento que una implementación por fuerza bruta. Una mejora sustancial al algoritmo es ejecutar KMP tomando como palabra a ubicar el texto rotado, y como fuente **una copia duplicada** del texto original, es decir si el texto es ABC, tomaremos como fuente ABCABC. Es fácil de ver que cualquier rotación posible del texto original está completamente contenida dentro del texto duplicado, tendrá una parte dentro de la primera mitad, y el resto contiguo en la segunda. Con esta estrategia, una única ejecución de KMP es suficiente para poder decidir si un string es rotación de la otra.

Esta estrategia también es una reducción del problema a uno de string matching resoluble por KMP. Aquí, se debe aplicar una transformación de los datos de entrada para que “encaje” con el uso que queremos darle a KMP, a diferencia del caso anterior que se pasaban el string original y la potencial rotación como palabra y texto de KMP. El orden mejora sustancialmente al solo ser una única ejecución de KMP, es lineal: la transformación definida también tiene orden lineal, y se hace una única ejecución de la caja negra, a diferencia de la primera versión que hacía N ejecuciones en el peor caso.

Cabe aclarar que las cotas de complejidad temporal que encontramos no son conclusivas, la reducción solo nos puede garantizar que la complejidad del problema que está siendo reducido es, *a lo sumo*, del mismo orden que el algoritmo al que se reduce, asumiendo que las funciones de transformación de datos de entrada y salida son de orden menor o igual a los problemas en sí, que es nuestro caso. El problema de rotaciones cíclicas es entonces, como máximo, de la misma complejidad que el string matching de KMP, pero nada nos dice que no exista una solución que nos de un orden menor.

2.1.1. Conclusiones

El algoritmo por fuerza bruta inicial es un orden cuadrático en la longitud del string. Las reducciones a KMP nos llevan a decir que el problema de rotaciones cíclicas es, **a lo sumo**, de complejidad cuadrática en la primera versión, y lineal en la segunda. No se demuestra que esta última cota es la complejidad real del problema porque la reducción como herramienta no nos permite poder afirmarlo.

A continuación se muestra los tiempos de ejecución de los tres algoritmos para distintos tamaños de cadenas de caracteres. En todos los casos se busca detectar si una cadena de tipo AAA...B es rotación cíclica de B...AAA. De la manera que fueron implementadas las soluciones coincide con el peor caso posible de tiempo de ejecución, en donde se rota N veces la segunda cadena para llegar a la primera. Se puede apreciar los órdenes cuadráticos de las implementaciones por fuerza bruta y KMP no optimizado (ésta mucho más pronunciada que la primera), y la sustancial mejora que se logra cuando se optimiza el código para usar una sola ejecución de KMP.

longitud (N)	kmp	bruteforce	kmp-optimizado
1000	0.24	0.06	0.03
2000	0.82	0.15	0.03
3000	1.79	0.29	0.03
4000	3.17	0.48	0.03
5000	4.94	0.75	0.03
6000	7.25	1.05	0.03
7000	10.16	1.46	0.03
8000	12.96	1.85	0.03
9000	16.5	2.39	0.04
10000	22.61	2.95	0.01
100000			0.06
1000000			0.37
5000000			1.74

Cuadro 1: Tiempos de ejecución en segundos para los distintos algoritmos

2.2. Ejercicio 2

El problema descrito es, en efecto, una variante del conocido problema del viajante (*TSP* según sus siglas en inglés), en donde dado un conjunto de n ciudades, y una ciudad de inicio, se desea encontrar un camino tal que:

- Comience y termine en la ciudad de inicio
- Pase por todas las ciudades exactamente una vez
- Minimice la distancia recorrida (u otra métrica)

Nuestra variante busca simplemente maximizar la distancia recorrida total en vez de minimizarla. Cabe mencionar que las reglas preestablecidas contribuyen a poder modelar la situación como un problema del viajante.

El problema del viajante puede ser definido sobre un grafo: Dado un grafo completo (todos los nodos tienen aristas con todos) con pesos, y un nodo inicial en el grafo, encontrar un ciclo (también conocido como tour) en el grafo tal que:

- Comience y termine en el nodo inicial
- Pase por todos los demás nodos exactamente una vez
- Minimice el peso de las aristas del ciclo

Las primeras dos condiciones se pueden resumir a pedir que el ciclo sea Hamiltoniano.

Se demostrará a continuación que el problema es *difícil*, es decir, que es un problema caracterizado como NP-Hard, no resoluble en tiempo polinomial, e igual o más difícil que cualquier problema NP. Para ello, nos basamos en la herramienta anteriormente vista, las reducciones polinomiales, para demostrar que un problema NP-Hard que tomamos como conocido, es reducible a nuestra variante del *TSP*. Nuestro problema conocido será el de búsqueda de ciclos Hamiltonianos dentro de un grafo.

Se mostrará que $\text{Ciclo Hamiltoniano} \leq_p \text{TSP}$.

Dado un grafo dirigido $G = (V, E)$, se define la siguiente instancia del TSP. Se tiene una ciudad v'_i para cada nodo v_i del grafo G . Se define $d(v'_i, v'_j)$ igual a 2 de haber una arista (v_i, v_j) en G , y se define igual a 1 en caso contrario. Esto arma el grafo completo del TSP.

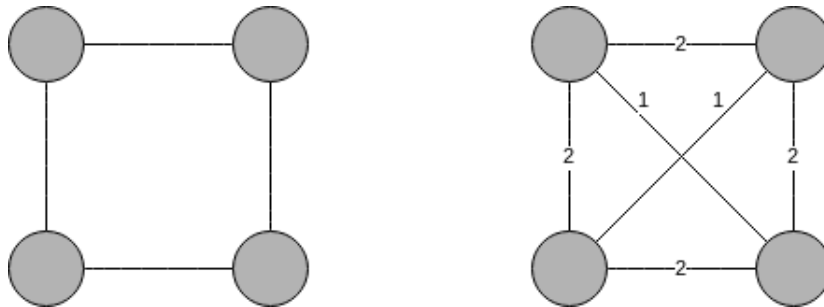


Figura 2: Grafo original G y transformación usada en TSP

G tiene un ciclo Hamiltoniano si y sólo si hay un ciclo de longitud como mínimo $2n$ en el TSP estudiado. Si G tiene un ciclo Hamiltoniano, entonces este ordenamiento de las correspondientes ciudades define un tour de longitud $2n$. ¿Por qué? Si existe un ciclo Hamiltoniano sobre G , necesariamente ese camino será la solución a nuestro TSP que maximiza distancias, porque por definición, dijimos que las aristas que estaban originalmente en G tendrán peso 2, y las que agregamos para hacer el grafo completo, tendrán peso 1. Estas n aristas de peso 2 terminan sumando $2n$.

A la inversa, suponga que hay un tour de como mínimo $2n$. La expresión para la longitud de este tour es la suma de n términos y como, por definición, hay sólo aristas de peso 1 y 2, entonces debe ser el caso que todos los términos son iguales a 2. Por lo tanto, todas estas aristas forman el

ciclo Hamiltoniano en el TSP, y también en el grafo original G , porque las aristas de peso 2 en el TSP son las que pertenecían a G .

Esta equivalencia demostrada nos lleva a concluir que el problema de encontrar ciclos Hamiltonianos en grafos puede ser resuelto de manera general a través de un caso particular de nuestro TSP que busca distancias máximas. Sabiendo que está demostrado que el problema de ciclos es NP-Hard, necesariamente nuestro TSP también lo es, ya que encontramos una manera de resolver ciclos Hamiltonianos utilizando TSP como caja negra interna, entonces este último es necesariamente igual o más difícil que el primero.

2.3. Ejercicio 3

Como primer paso, demostraremos que, dada una cantidad n de cursos, si llamamos m a la cantidad máxima de cursos superpuestos en cualquier instante de tiempo, la cantidad de aulas necesarias para realizar la asignación es m .

Demostración

Para demostrar esto, se plantearán las cotas superior e inferior, y se demostrará que ambas coinciden, siendo el valor de ambas: m .

Cota inferior

De haber m cursos que se dictan al mismo tiempo, como no se pueden dictar cursos distintos en una misma aula en simultáneo, queda probado que como mínimo debe haber tantas aulas como cursos se dictan en simultáneo. Por lo tanto, es válido afirmar que

$$Aulas\ necesarias \geq m$$

Cota superior

Para todo instante t de tiempo habrá, según lo demostrado anteriormente, como mínimo m aulas disponibles. Se puede comprobar rápidamente que para ningún t se necesitarán más que m aulas, dado que si dos cursos no se superponen temporalmente, no hay razón por la que no puedan compartir una misma aula. Además se está tratando el caso en que todas las aulas son iguales, lo que evita complicaciones más allá del análisis presentado.

Llegamos entonces a que

$$Aulas\ necesarias \leq m$$

Queda así demostrado que $Aulas\ necesarias = m$

2.3.1. Pseudocódigo

Se muestra el algoritmo para encontrar el valor de m .

Algorithm 1: Pseudo código que resuelve el problema.

Data: Horarios de inicio y finalización de cada uno de los n cursos: $T_{inicio,j}$ y $T_{fin,j}$ denotan los tiempos de inicio y finalización del curso j -ésimo. Conjunto de todos los tiempos: $T_i, i \in (1, 2n)$

Result: Menor cantidad de aulas necesarias para acomodar todos los cursos suponiendo que todas las aulas son iguales.

```

min time slice  $\leftarrow \infty$ ;
min time  $\leftarrow \infty$ ;
max time  $\leftarrow 0$ ;
foreach  $T_i$  do
    if ( $current\ min\ time\ slice = |T_i - T_j| < min\_time\_slice, \forall j/i \neq j$ ) then
        min time slice := current min time slice;
    if  $T_i < min\ time$  then
        min time :=  $T_i$ ;
    if  $T_i > max\ time$  then
        max time :=  $T_i$ ;

current time := min time;
max superpositions := 0;
while  $current\ time < max\ time$  do
    current num of superpositions := 0;
    foreach  $T_{inicio,j}, T_{fin,j}$  do
        if  $T_{inicio,j} \leq current\ time < T_{fin,j}$  then
            ++current num of superpositions;
    if  $current\ num\ of\ superpositions > max\ superpositions$  then
        max superpositions := current num of superpositions;
    current time += min time slice;

```

Mín time slice

El mínimo time slice es lo que define el paso de tiempo que habrá que tomar para poder analizar superposiciones en la menor cantidad de *timestamps* sin perder de vista ninguna superposición. El paso a tomar debe ser entonces equivalente al mínimo intervalo de tiempo entre dos T_i s, siendo T_i cada uno de los tiempos de inicio y finalización de los cursos. Se muestran unos gráficos con el fin de simplificar la comprensión de lo explicado.

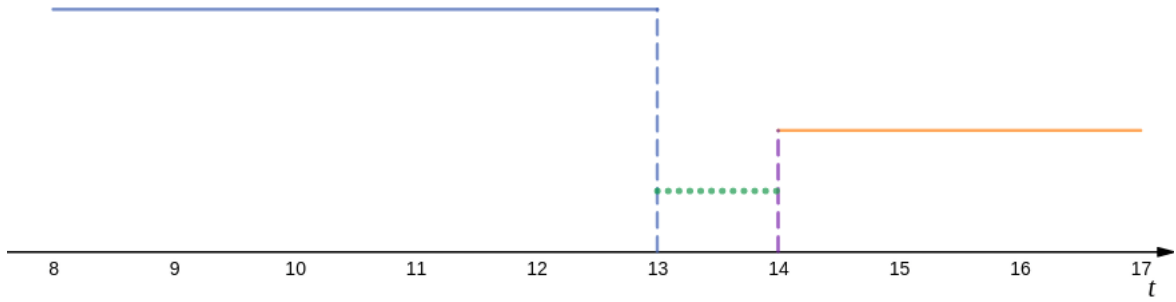


Figura 3: En esta figura se puede ver el caso de dos cursos separados, sin superposición. El time slice mínimo será de una hora: analizando los tiempos cada una hora partiendo del primer inicio, se contemplarán todas las posibles superposiciones. Min time slice: 1 hora

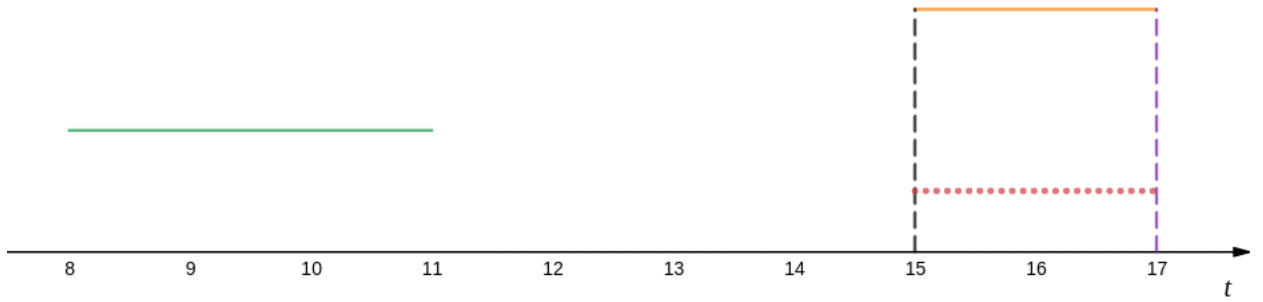


Figura 4: En esta figura se puede observar el caso de dos cursos separados. Esta vez el time slice toma el valor de la duración del segundo curso. Es evidente que no tiene sentido tomar un time slice menor a ese. Min time slice: 2 horas.

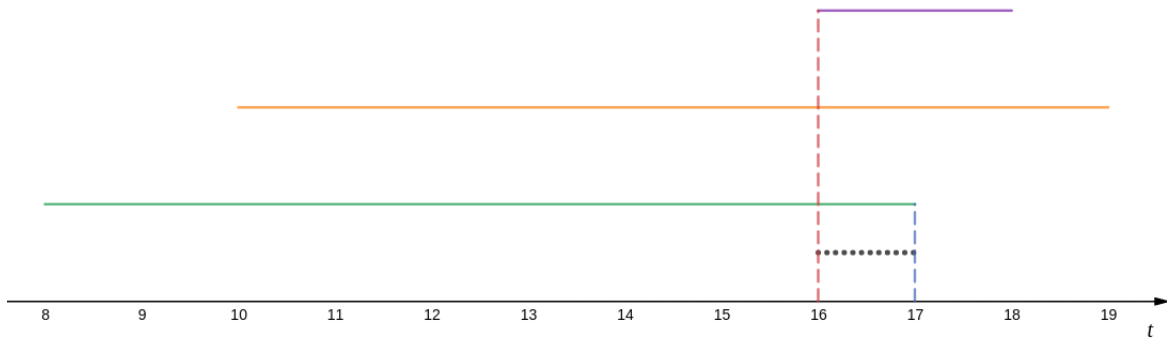


Figura 5: Min time slice: 1 hora.

En el `while` se analizan todos los instantes de tiempo, arrancando con el primer tiempo de inicio, aumentando de a intervalos de tiempo acorde a lo calculado anteriormente. Para cada instante de tiempo se analiza la cantidad de cursos que se dictan. De esta forma, una vez analizados todos los instantes de tiempo, se obtiene la cantidad máxima de superposiciones.

2.3.2. Función variante

El **foreach** del algoritmo termina en a lo sumo n^2 iteraciones. Por definición, el **foreach** ejecutará una iteración por T_i . En cada iteración, además, se compara a cada T_i con cada $T_j, \forall i \neq j$. Como la cantidad de T_i s es n , queda probado que habrá máximo n^2 iteraciones.

$$n^2 - \text{cantidad de comparaciones } T_i \text{ vs } T_j \quad (1)$$

Se puede observar que esta función es estrictamente decreciente.

El **while** termina una vez que se analizó una cantidad finita de tiempos, dependiendo ésta del mínimo intervalo de tiempo en que se presenta algún cambio en el *scheduling*.

Queda así demostrado que el algoritmo propuesto está en P (la complejidad es: cuadrática + lineal en las divisiones del intervalo de tiempo. Si se hace tender el mínimo time slice a cero, la cantidad de intervalos tenderá a infinito. Pero dado que se estudia el caso de un solo día, y el time slice real no ha de ser tan chico - podríamos suponer como algo chico 5 minutos, lo que daría en 24 hs 288 tiempos a analizar).

2.3.3. Reducción a problema de coloreo de grafos

Considérese cada una de las materias como un nodo. Dados dos nodos, se establece una conexión entre ellos mediante una arista únicamente en caso de superposición horaria entre los cursos que representan dichos nodos. De esta forma, si se encuentra la forma de pintar todos los vértices de forma que no queden dos adyacentes con el mismo color, utilizando la cantidad mínima de colores, se habrá resuelto el problema propuesto.

2.3.4. ¿P=NP?

No se pueden sacar conclusiones al respecto. El algoritmo propuesto es P , mientras que el coloreo de grafos es NP . La reducción no proporciona suficiente información para sacar conclusiones: lo único que se puede concluir es que el problema de coloreo es igual o mas difícil que el problema de aulas, lo cual no es suficiente para probar nada.