

---

## Práctica 4

MIKEL BARRENETXEA Y AMAIA LOBATO  
DISEÑO DE ALGORITMOS

---

### El problema de los semáforos

El objetivo de esta práctica es diseñar un algoritmo para proponer soluciones a un problema de optimización mediante un algoritmo voraz e implementarlo en el ordenador. Hemos escogido el problema de los semáforos que aparecía en la hoja 5 de problemas. Se trata de diseñar el patrón de semáforos de un cruce de calles como el de la Figura 1 y que minimice el tiempo de espera. El problema se representa por un grafo en el que cada nodo representa un giro posible y los arcos conectan giros incompatibles. Con esta representación este problema es equivalente al problema del coloreado de grafos en el que el objetivo es encontrar el número mínimo de colores necesario para colorear los nodos de un grafo de forma que ningún arco conecte nodos del mismo color.

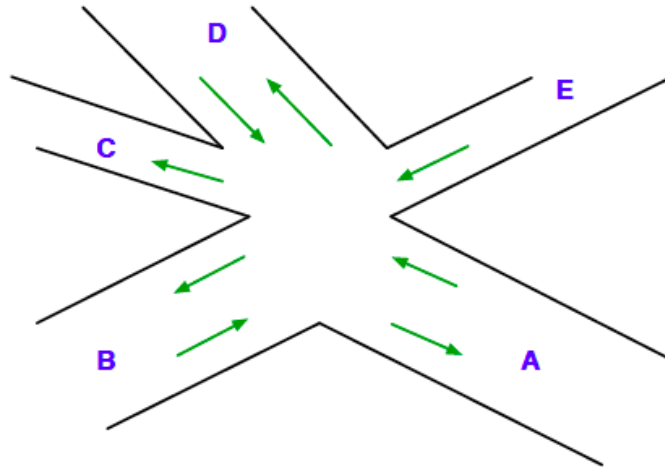


Figura 1: Cruce

En primer lugar, hemos visto cuáles son los giros posibles: AB, AC, AD, BA, BC, BD, DA, DB, DC, EA, EB, EC y ED. Por tanto, en total hay 13 giros posibles. Estos 13 giros serán los 13 nodos del grafo. El primer nodo será el giro AB, el segundo nodo será el giro AC, y así sucesivamente. A continuación hemos construido una matriz de tamaño  $13 \times 13$  cuyos elementos son ceros y unos. Cada columna y cada fila están asociadas a un nodo del grafo, es decir, la primera columna y fila de la matriz están asociadas al nodo AB, la segunda columna y fila de la matriz están asociadas al nodo AC y así sucesivamente siguiendo el orden de nodos que hemos fijado previamente. Si el  $(i, j)$ -ésimo elemento de la matriz es un uno, el  $i$ -ésimo nodo y el  $j$ -ésimo nodo estarán conectados, es decir, los giros asociados a dichos nodos no serán compatibles. En cambio, si el  $(i, j)$ -ésimo elemento de la matriz es un cero, el  $i$ -ésimo nodo y el  $j$ -ésimo nodo no estarán conectados, es decir, los giros asociados a dichos nodos serán compatibles. La matriz que hemos logrado es la siguiente:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Con esta matriz ya tenemos toda la información necesaria para dibujar el grafo que representaría nuestro problema. Pero no vamos a dibujarlo, ya que al haber tantos nodos y tantas aristas, no sería fácil distinguir qué nodos están conectados entre sí.

## Algoritmo voraz

El objetivo del problema es diseñar un patrón de semáforos que minimice el tiempo de espera. Considerando la representación del problema como un grafo, se trata de usar el menor número de colores de tal manera que todos los nodos del grafo estén coloreados y que ninguna arista conecte dos nodos del mismo color. Es decir, nuestra función objetivo es minimizar el número de colores de los nodos (teniendo en cuenta que ninguna arista puede conectar dos nodos del mismo color).

La función de selección que hemos usado consta de dos partes. En primer lugar, se escoge el nodo que vamos a colorear. Una vez elegido el nodo que vamos a colorear, para cada color que ya haya sido usado previamente, miramos si el nodo elegido está conectado con algún nodo de dicho color. En el momento en el que se encuentra un color tal que ningún nodo de este color esté conectado con el nodo elegido, se le asigna al nodo elegido dicho color. En caso de que el nodo elegido esté conectado con algún nodo de cada uno de los colores que hayan sido usados previamente, se le asigna al nodo elegido un nuevo color que no se ha utilizado todavía.

Para elegir el nodo que vamos a colorear hemos usado tres criterios diferentes:

1. Escoger el nodo con más conexiones.
2. Escoger el nodo con menos conexiones.
3. Escoger un nodo cualquiera al azar (utilizando el comando `random`).

Con la función de selección que hemos utilizado (utilizando cualquiera de los tres criterios para elegir el nodo que vamos a colorear), no podemos garantizar que la solución que vayamos a obtener sea la solución óptima del problema. De hecho, el problema de coloreado de grafos es uno de los problemas NP-completos clásicos (no existe ningún algoritmo de tiempo polinómico que lo resuelva de forma óptima). Lo que sí podemos dar es una cota superior de los colores que se necesitarán para colorear el grafo. Si un grafo es de grado máximo  $\Delta$  (el número de aristas que incidan sobre cualquier nodo del grafo será menor o igual que  $\Delta$ ), el grafo es  $(\Delta + 1)$ -coloreable.

Aunque para muchos grafos utilizando esta función de selección obtendremos muy buenos resultados, hay grafos para los que esta función no siempre da buenos resultados, como por ejemplo los "grafos de la corona" (crown graphs). Estos grafos tienen  $2n$  nodos que se dividen en dos conjuntos:  $U = \{u_1, \dots, u_n\}$  y  $V = \{v_1, \dots, v_n\}$ . Los nodos  $u_i$  y  $v_j$  están unidos por una arista para cada  $i \neq j$ . Veamos algunos ejemplos:

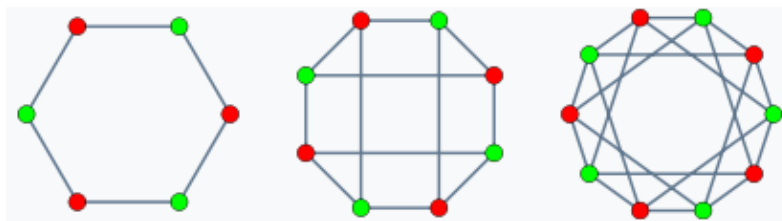


Figura 2: 'Grafos de la corona' con 6, 8 y 10 nodos

Si coloreamos de un color el nodo  $u_i$  y después seleccionamos el nodo  $v_i$  (podría pasar ya que todos los nodos de estos grafos tienen el mismo número de conexiones), como los nodos  $u_i$  y  $v_i$  no están conectados, le asignaríamos al nodo  $v_i$  el mismo color que a  $u_i$  y esto nos llevaría a tener que usar  $n$  colores para colorear el grafo, cuando en realidad solo se necesitan dos colores (coloreando todos los nodos de  $V$  con un color y todos los nodos de  $U$  con otro color).

# Algoritmo

---

## Algoritmo 1: Algoritmo voraz

---

**Entrada:** *matriz*: Una matriz cuadrada compuesta por zeros y unos. Si el  $(i, j)$ -ésimo elemento de la matriz es 1, significa que el nodo  $i$  y el nodo  $j$  están conectados. En cambio si el  $(i, j)$ -ésimo elemento de la matriz es 0, significa que el nodo  $i$  y el nodo  $j$  no están conectados.

$C$ : Conjunto de nodos.

**Salida:** Una partición del conjunto  $C$ , cuyos elementos son conjuntos que contienen nodos del grafo que se pueden dibujar del mismo color.

**Función** semáforos (*matriz*,  $C$ )

**principio**

```
1: nodo_inicial  $\leftarrow$  elegir_nodo(matriz,  $C$ )
2: colores  $\leftarrow$  {{nodo_inicial}}
3:  $C \leftarrow C - \text{nodo\_inicial}$ 
4: mientras  $C \neq \emptyset$  hacer
5:    $n_0 \leftarrow$  elegir_nodo(matriz,  $C$ )
6:    $C \leftarrow C - n_0$ 
7:   insertado  $\leftarrow$  False
8:   para color  $\in$  colores hacer
9:     si insertado = False entonces
10:      cont  $\leftarrow$  0
11:      para nodo  $\in$  color hacer
12:        si matriz[ $n_0$ ][nodo] = 1 entonces
13:          cont  $\leftarrow$  cont + 1
14:        fin si
15:      fin para
16:      si cont  $\leftarrow$  0 entonces
17:        color  $\leftarrow$  color  $\cup \{n_0\}$ 
18:        insertado  $\leftarrow$  True
19:      fin si
20:    fin si
21:  fin para
22:  si insertado  $\leftarrow$  False entonces
23:    colores  $\leftarrow$  colores  $\cup \{n_0\}$ 
24:  fin si
25: fin mientras
26: devolver colores
```

---

---

Algoritmo 2: elegir\_nodo\_max

---

**Entrada:** matriz: Una matriz cuadrada compuesta por zeros y unos. Si el  $(i, j)$ -ésimo elemento de la matriz es 1, significa que el nodo  $i$  y el nodo  $j$  están conectados. En cambio si el  $(i, j)$ -ésimo elemento de la matriz es 0, significa que el nodo  $i$  y el nodo  $j$  no están conectados.

$C$ : Conjunto de nodos.

**Salida:** El nodo con más conexiones del conjunto  $C$

**Función** elegir\_nodo\_max (matriz,  $C$ )

**principio**

```
1: maximo  $\leftarrow$  0
2: nodo_max  $\leftarrow$  0
3: para  $i \in C$  hacer
4:   cont  $\leftarrow$  0
5:   para  $j \in [0, 1, \dots, \text{longitud}(\text{matriz})]$  hacer
6:     si matriz[i][j]=1 entonces
7:       cont  $\leftarrow$  cont + 1
8:     fin si
9:   fin para
10:  si cont > maximo entonces
11:    maximo  $\leftarrow$  cont
12:    nodo_max  $\leftarrow$  i
13:  fin si
14: fin para
15: devolver nodo_max
```

---

---

Algoritmo 3: elegir\_nodo\_min

---

**Entrada:** matriz: Una matriz cuadrada compuesta por zeros y unos. Si el  $(i, j)$ -ésimo elemento de la matriz es 1, significa que el nodo  $i$  y el nodo  $j$  están conectados. En cambio si el  $(i, j)$ -ésimo elemento de la matriz es 0, significa que el nodo  $i$  y el nodo  $j$  no están conectados.

$C$ : Conjunto de nodos.

**Salida:** El nodo con menos conexiones del conjunto  $C$

**Función** elegir\_nodo\_min (matriz,  $C$ )

**principio**

```
1: minimo  $\leftarrow$  longitud(matriz)
2: nodo_min  $\leftarrow$  0
3: para  $i \in C$  hacer
4:   cont  $\leftarrow$  0
5:   para  $j \in [0, 1, \dots, \text{longitud}(\text{matriz})]$  hacer
6:     si matriz[i][j]=1 entonces
7:       cont  $\leftarrow$  cont + 1
8:     fin si
9:   fin para
10:  si cont < minimo entonces
11:    minimo  $\leftarrow$  cont
12:    nodo_min  $\leftarrow$  i
13:  fin si
14: fin para
15: devolver nodo_min
```

---

Para escoger al azar el nodo, no hará falta otra función, bastará con usar el comando *random*.

# Análisis de complejidad computacional

En primer lugar, analizaremos la complejidad temporal de nuestro algoritmo voraz. Para ello, supondremos que los nodos ya nos los dan ordenados de alguna manera. Por lo tanto, a la hora de hacer el análisis, no vamos a tener en cuenta cuánto tardamos en ordenar los nodos, pero ya hemos visto en el curso, que para ordenar los elementos de una lista es algo que podemos hacer en tiempo  $\Theta(n \log(n))$ .

En el algoritmo, el primer bucle lo abrimos con un *mientras*. El *mientras* lo vamos a hacer hasta que no queden nodos por dibujar, es decir,  $n$  veces. Dentro del *mientras*, abrimos otro bucle donde recorremos la lista *colores*. En la lista *colores* tenemos tantos elementos como colores hemos usado hasta el punto donde nos encontramos. Ahora, lo que necesitamos es saber la cantidad de colores que tenemos.

Nuestro algoritmo coge un nodo en cada iteración y lo intenta colorear con uno de los colores ya usados si es posible. Si no es posible, usará un nuevo color. Por lo tanto, nuestro algoritmo usará a lo más  $\Delta + 1$  colores siendo  $\Delta$  el grado máximo del grafo. En el peor caso, si todos los nodos están conectados entre sí, harían falta  $n$  colores.

Por último, dentro del bucle de *colores* abrimos otro bucle que depende de la cantidad de nodos que hay en ese color. El peor caso en este bucle se da cuando no existe ninguna conexión entre los nodos. En este caso habrá  $n$  nodos de un único color. Pero éste es el caso contrario del que hemos comentado antes para el bucle *colores*. Por lo tanto, tener  $n$  colores y  $n$  nodos de un color es un suceso incompatible.

Por lo tanto, la complejidad temporal de nuestro algoritmo voraz es  $T(n) \in \Theta(n^2)$ .

Hemos hecho algunas pruebas para analizar la complejidad temporal. Para ello, hemos creado un programa en el que íbamos aumentando el tamaño del problema y lo resolvíamos 100 veces para cada valor de  $n$ . Los resultados son los siguientes:

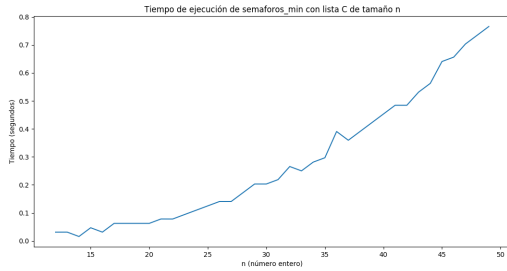


Figura 3: Ordenando los nodos de más conexiones a menos.

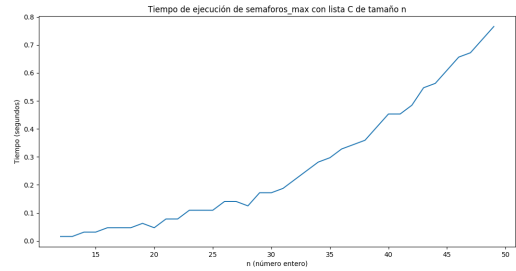


Figura 4: Ordenando los nodos de menos conexiones a más.

Como podemos ver en las Figuras (3) y (4) sí que se puede decir que el tiempo crece de forma cuadrática respecto al tamaño del problema.

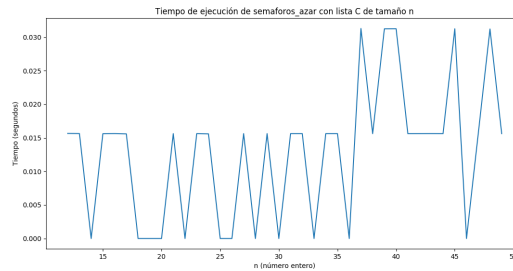


Figura 5: Cogiendo los elementos de la lista al azar, sin ningún orden.

En la Figura (5) no podemos ver ninguna relación. Esto se debe a que los tiempos de ejecución son muy pequeños en este caso. Hay una gran diferencia de tiempo comparando con las otras dos funciones de selección. Estas diferencias se dan porque en este caso no tenemos que escoger los nodos siguiendo un orden, entonces, como lo hacemos aleatoriamente, nos ahorramos un tiempo.

Para ver mejor la relación entre el tiempo y el tamaño del problema en el caso de la función selección de azar, vamos a repetir el problema 1000 veces en lugar de 100.



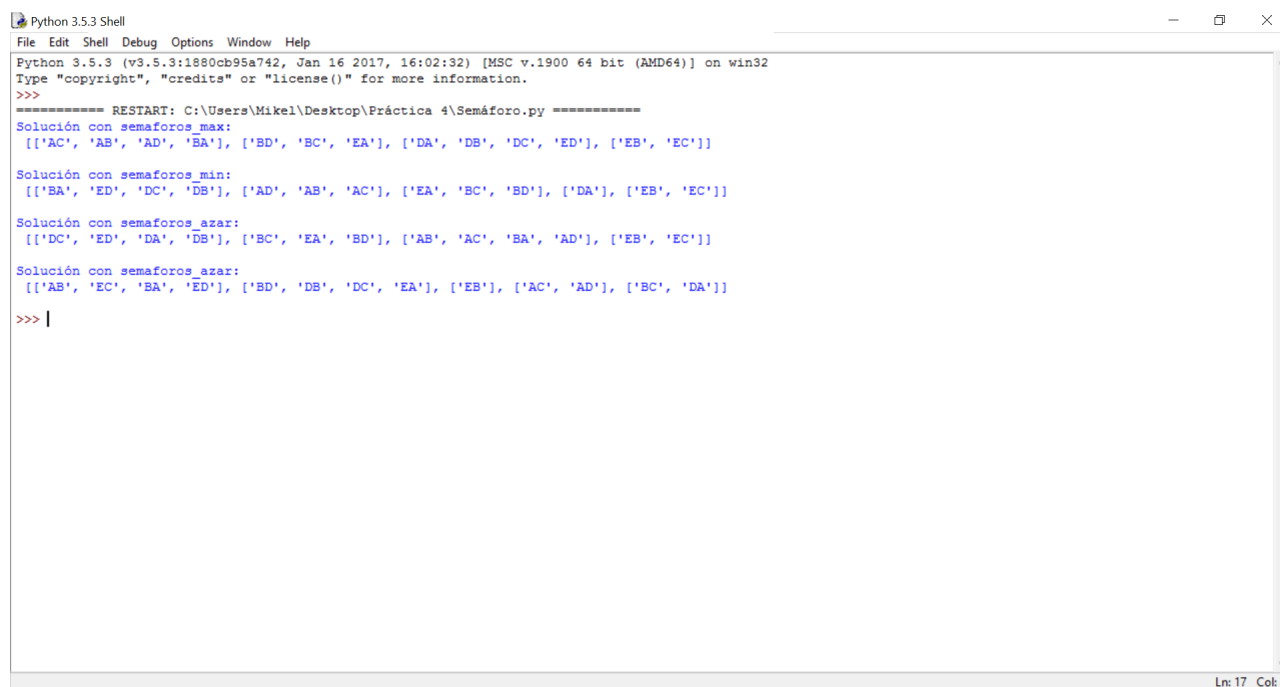
Figura 6: Cogiendo los elementos de la lista al azar, sin ningún orden.

En la Figura (6) ya podemos observar que existe una relación entre el tiempo y el tamaño del problema y sí que se podría decir que es cuadrática.

Calcular la complejidad espacial del algoritmo es bastante sencillo. Los datos que necesitamos son los nodos (vector C) y una matriz que nos informa de las conexiones entre los nodos. Esta matriz es de tamaño  $n \times n$  y es el elemento más grande del algoritmo. Por lo tanto,  $S(n) \in \Theta(n^2)$ .

## Análisis de soluciones

Hemos resuelto el problema de los semáforos de la hoja 5 de ejercicios usando el algoritmo voraz con diferentes funciones de selección y las soluciones que hemos obtenido aparecen en la siguiente imagen:



```
Python 3.5.3 Shell
File Edit Shell Debug Options Window Help
Python 3.5.3 (v3.5.3:1880cb95a742, Jan 16 2017, 16:02:32) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Mikel\Desktop\Práctica 4\Semáforo.py =====
Solución con semaforos_max:
[['AC', 'AB', 'AD', 'BA'], ['BD', 'BC', 'EA'], ['DA', 'DB', 'DC', 'ED'], ['EB', 'EC']]

Solución con semaforos_min:
[['BA', 'ED', 'DC', 'DB'], ['AD', 'AB', 'AC'], ['EA', 'BC', 'BD'], ['DA'], ['EB', 'EC']]

Solución con semaforos_azar:
[['DC', 'ED', 'DA', 'DB'], ['BC', 'EA', 'BD'], ['AB', 'AC', 'BA', 'AD'], ['EB', 'EC']]

Solución con semaforos_azar:
[['AB', 'EC', 'BA', 'ED'], ['BD', 'DB', 'DC', 'EA'], ['EB'], ['AC', 'AD'], ['BC', 'DA']]
>>> |
```

Figura 7: Soluciones del problema de los semáforos.

Como podemos ver, en todos los casos se usan menos de  $n = 13$  colores. Aún así, podemos observar que hay diferencias entre las soluciones de las distintas funciones de selección.

Mientras que con la función de selección que elige siempre el nodo con el mayor número de conexiones conseguimos usar solamente cuatro colores (solución óptima); con la función de selección que hace lo contrario, elegir el mínimo, usamos cinco colores. Por lo tanto, la segunda función de selección nos da un resultado peor.

Con la función de selección que escoge los nodos aleatoriamente conseguimos resultados diferentes. Algunas veces logramos una solución que usa cuatro colores y otras veces una de cinco.

Hemos podido observar que en ningún caso hemos usado más de cinco colores. Además, el grado máximo de grafo del problema es  $\Delta = 8$  y nuestro algoritmo voraz nunca usará más de  $\Delta + 1$  colores.

Por lo tanto, queda claro que la función selección que escoge el nodo con el mínimo número de conexiones no es una buena función de selección para llegar a lograr una solución óptima. En cambio, la función de selección que escoge el nodo con el mayor número de conexiones puede llegar a dar la solución óptima, aunque no la garantiza.



## Ejemplos de ejecución de cada programa

En la Figura (8) tenemos un ejemplo de ejecución del programa *Semáforo*.

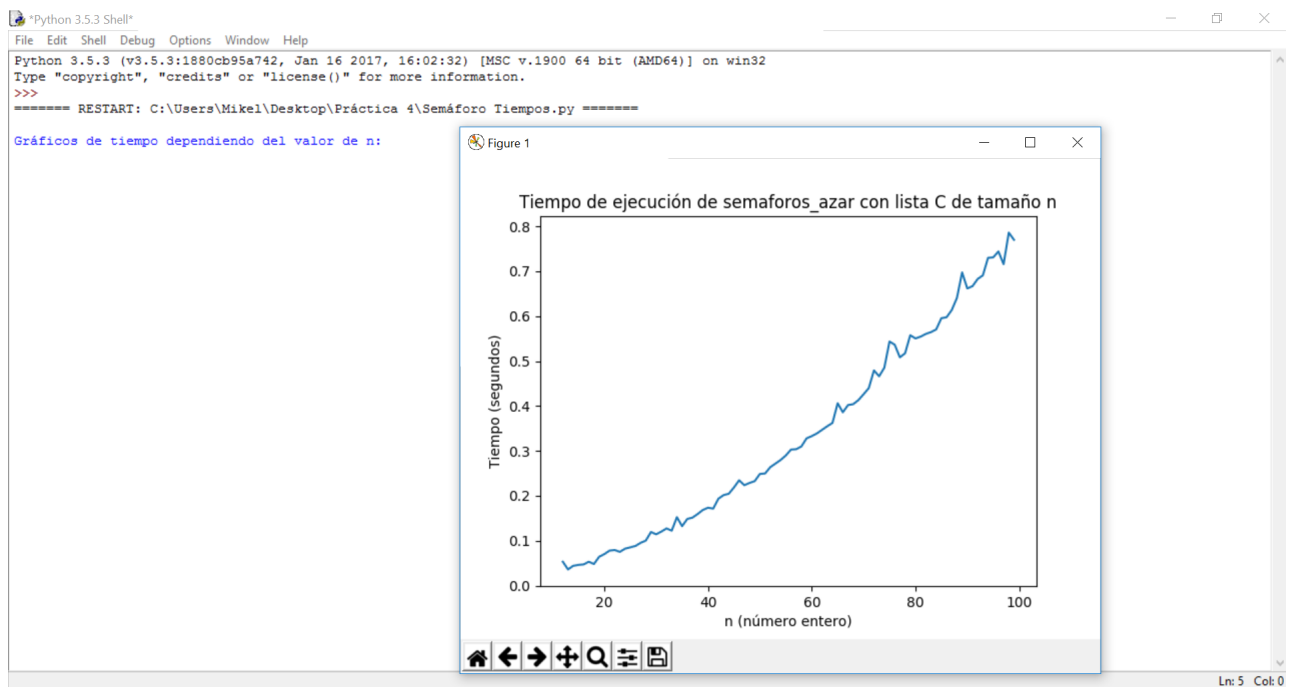


Figura 8: Ejemplo de ejecución de *Semáforo tiempos*.

## Bibliografía

- <http://www.tdx.cat/bitstream/handle/10803/5844/01Fjog01de01.pdf;jsessionid=60A91FE486CF5AE4F9501sequence=1>
- <http://making-code.blogspot.com.es/2015/08/algoritmos-de-coloracion-de-grafos.html>
- [https://en.wikipedia.org/wiki/Crown\\_graph](https://en.wikipedia.org/wiki/Crown_graph)
- [https://en.wikipedia.org/wiki/Greedy\\_coloring](https://en.wikipedia.org/wiki/Greedy_coloring)