Paralelización y mejoramiento de la métrica *clustering*.

Erick Muñiz Morales

Seminario de programación en paralelo.

August 18, 2021

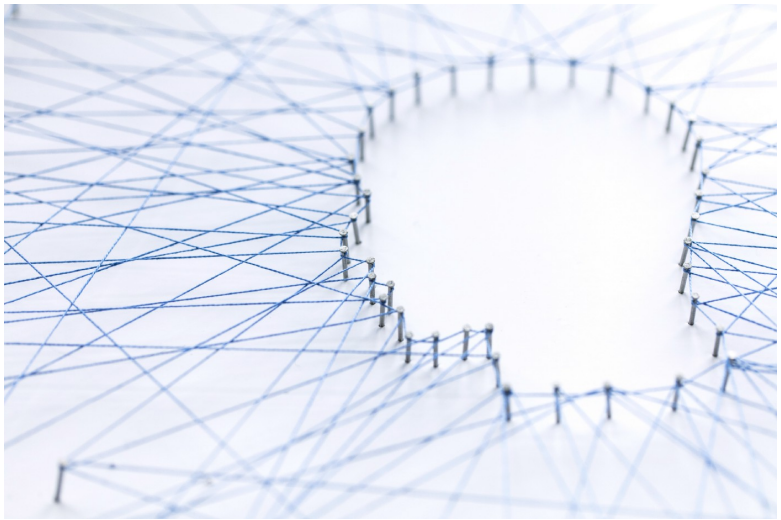
Resumen

- El tiempo computacional requerido para las métricas básicas de análisis de redes es bastante tardío.
- En específico, se hace una optimización para el cálculo del número de triángulos.
- Se obtiene un rendimiento bastante bueno, mejorando por más de 10 veces el implementado en Python con la función *clustering()*.

Una humilde introducción a sistemas complejos.

Una definición etimológica.

Complejidad viene del latin *plexus* que significa entrelazado.
Entonces, podríamos decir que, un sistema complejo es un sistema al cual no se le puede separar.



Evidentemente, esta defnición es muy escueta...

Lo que sí, tenemos propiedades.

Aunque, a veces depende del autor, en general son estas 3.

Propiedades generales.

1. Auto-organización y emergencia.
2. Independencia
3. Interdependencia

Auto-organización y emergencia.

Figure: clap,clap,clap

Independencia.

Figure: Un árbol

Interdependencia

Figure: La importancia de cierto jugadores.

Interdependencia/Independencia.

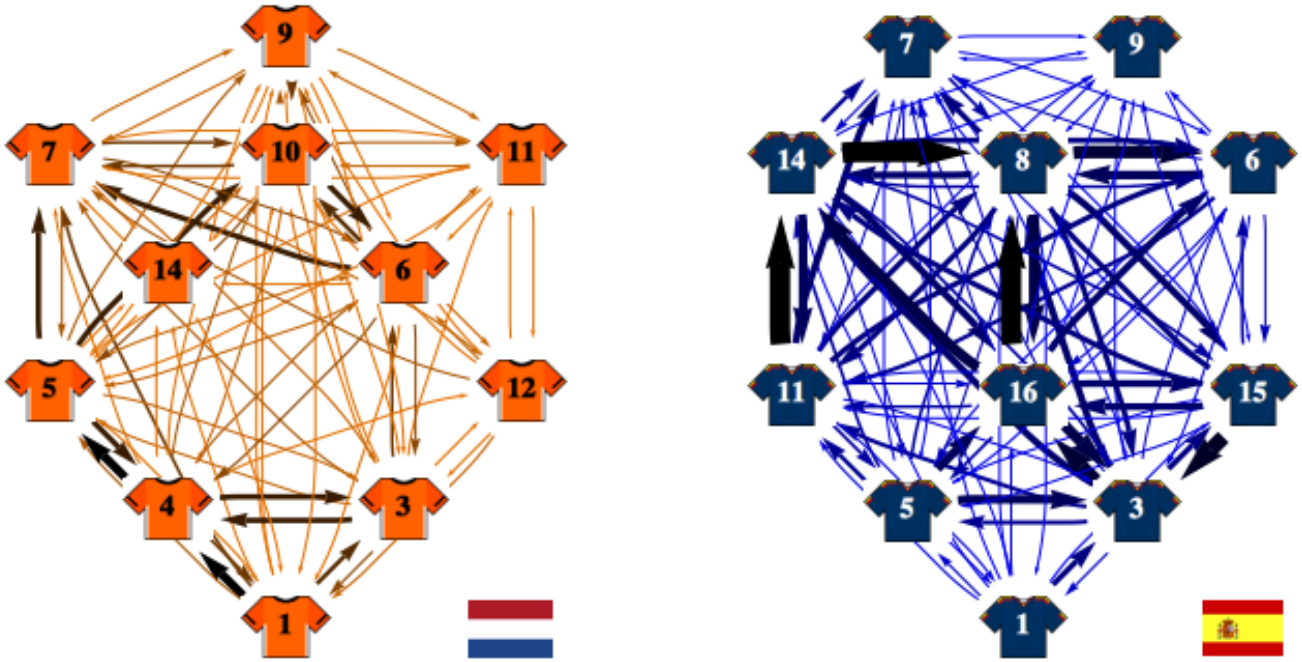


Figure: Ejemplo de red. Notamos los dos propiedades. Obtenida del artículo *A network theory analysis of football strategies*. [2]

Algunas referencias interesantes.

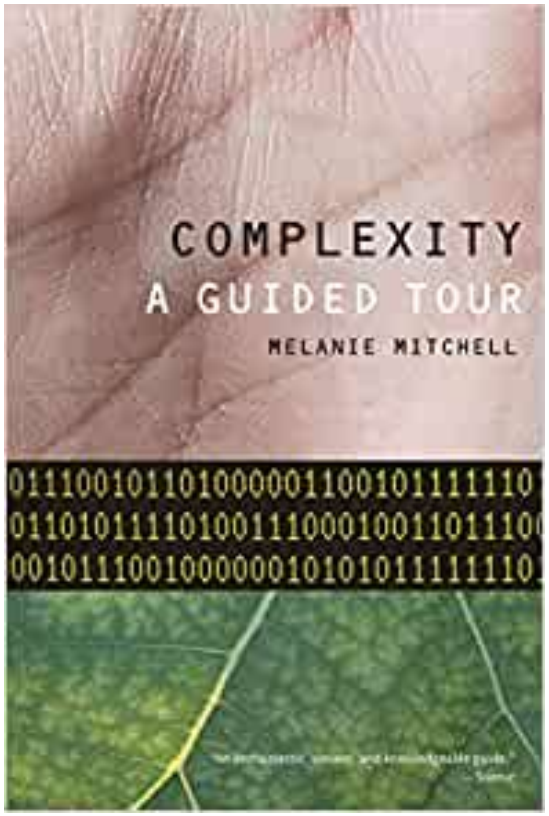


Figure: Algunas referencia interesantes. [1]

Algunas referencias interesantes.

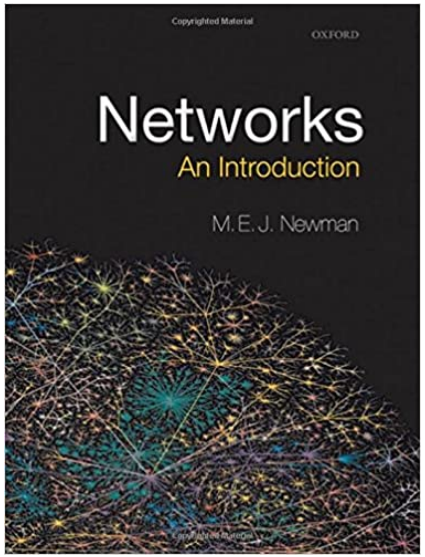


Figure: Algunas referencia interesantes. [1]

Algunas referencias interesantes.

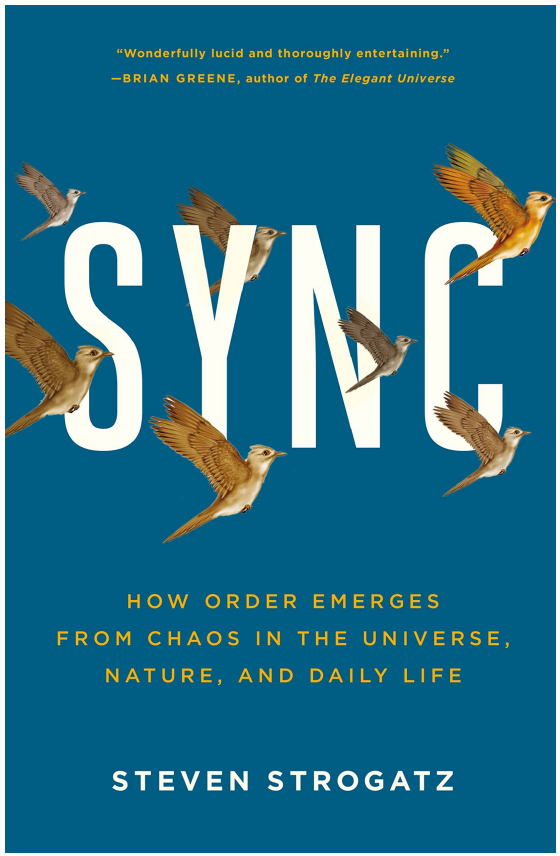
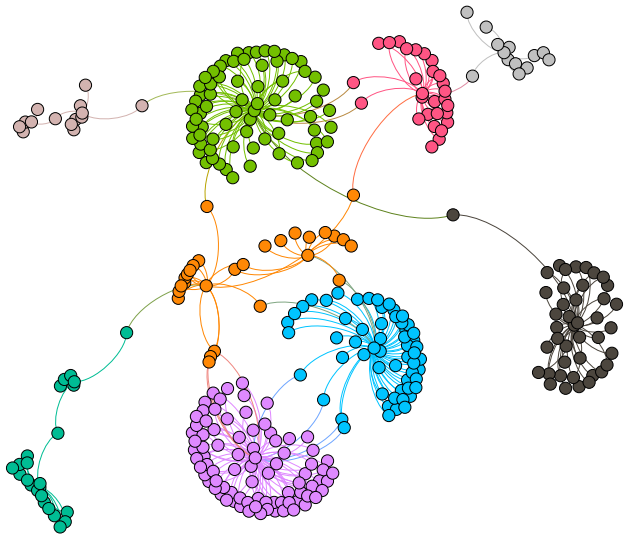


Figure: Algunas referencia interesantes.

Hay algo siempre constante: Las relaciones.

De todas la imágenes, en ningún momento se evalúa al sistema en función del comportamiento individual de los elementos del mismo.
¿Qué objeto nos permite recuperar y darle importancia a las relaciones y no a los objetos en sí?: **Las redes.**



Una red

Definición.

Una red es una gráfica con interpretación. En esencia, deja de ser abstracto.

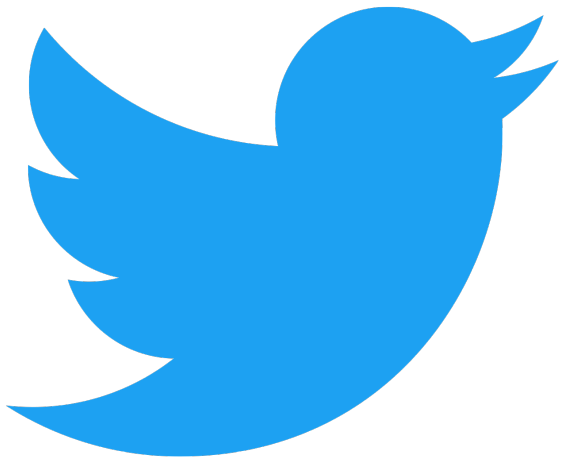
Así, de manera técnica, no dista de una gráfica. Por lo que, muchos métodos se heredan de gráficas.
Entre estos: camino más corto, diámetro, ciclos, etc...
Sólo se les da una interpretación.

Mi caso particular: Análisis de tendencias en *Twitter*.

- Una red de media social con más de 350 millones de usuarios activos cada mes.
- Un centro de discusión de diverso temas.
- Una dinámica simple de comunicación: Interacciones de costo mínimo.

Obejtivo:

Caracterizar a tendencias con comportamiento explosivo en función de los usuarios que se comunican.



Algunos gráficos bonitos.

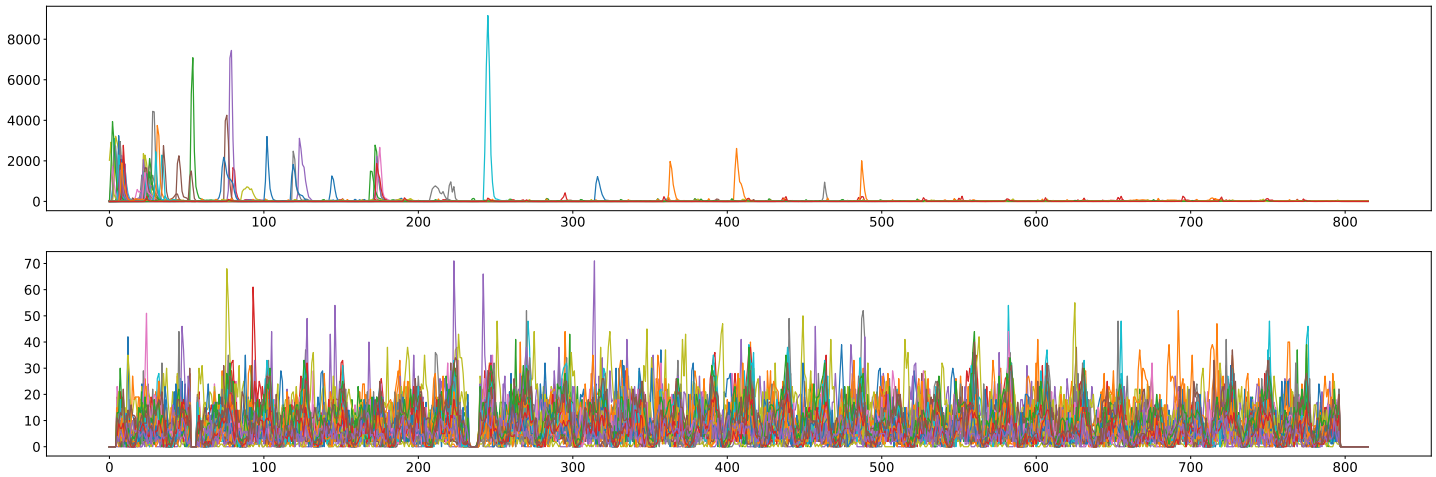


Figure: Ejemplo de series de tiempo.

Algunos gráficos bonitos. En una escala menor.

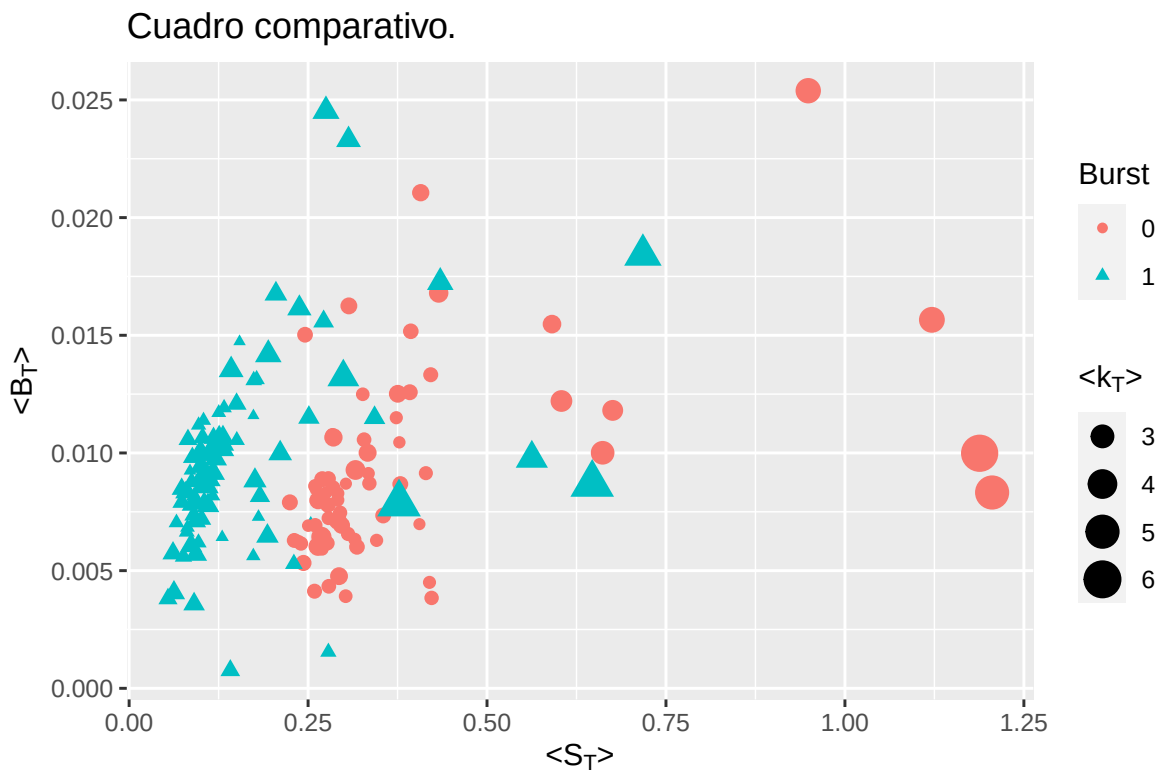


Figure: Una pequeña caracterización a nivel local. Un comportamiento esperable en el momento de mayor comunicación.

Algunos gráficos bonitos. En una escala menor.

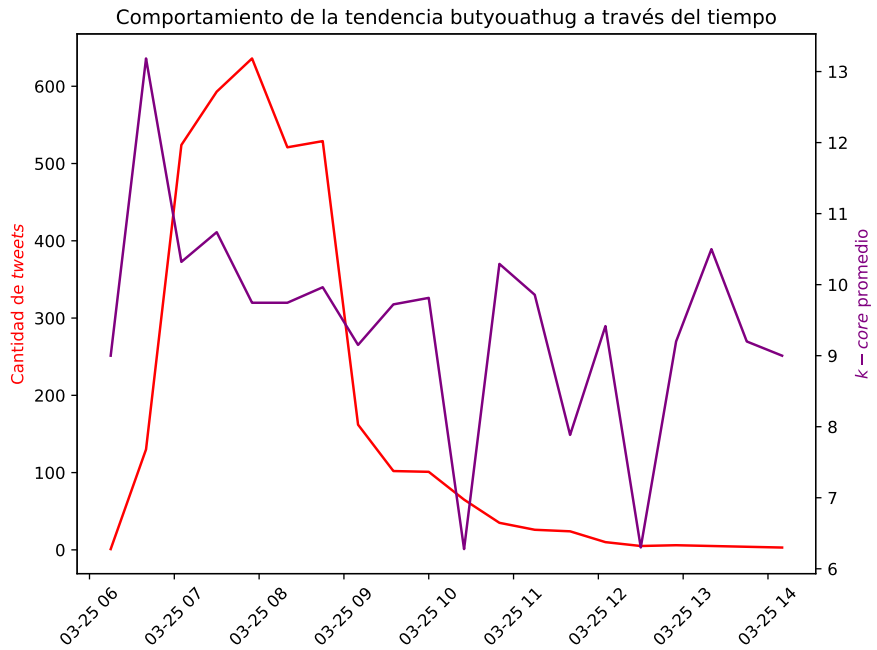


Figure: Ejemplo de un *ataque* grupal.

Algunos gráficos bonitos. En una escala menor.

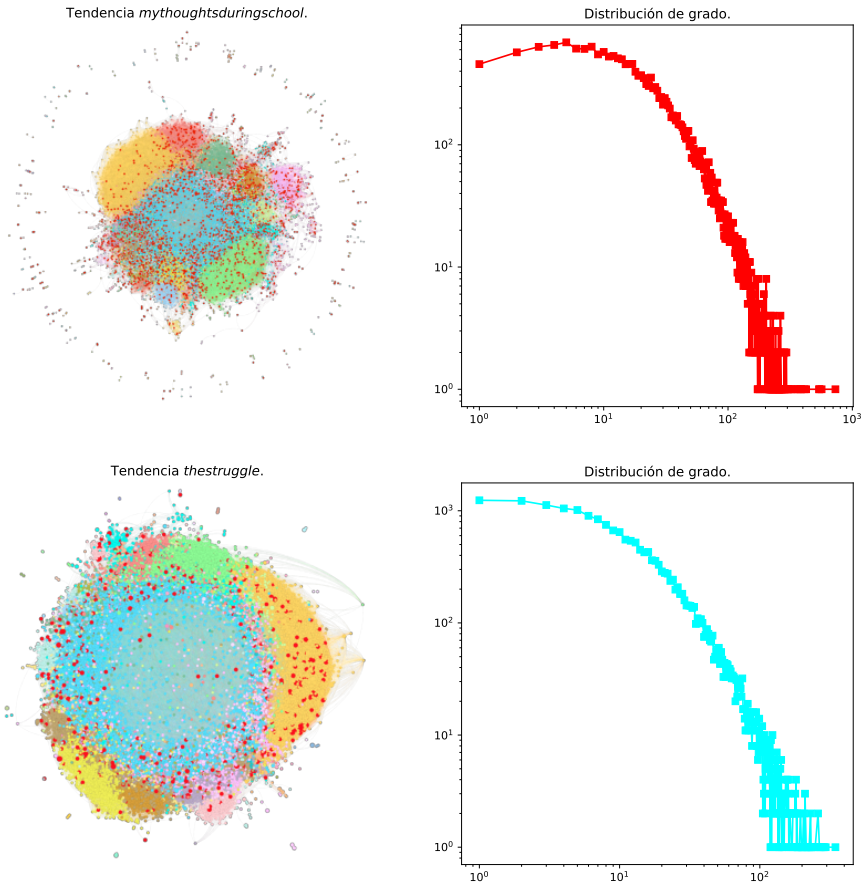


Figure: Redes de más de 10,000 nodos. Esta es una escala mayor a las anteriores.

Sobre gráficas.

Una gráfica es el par $G = (V, E)$ donde V es el conjunto de nodos y E el conjunto de enlaces o relaciones. Notemos que aquí se hace **énfasis en la topología del sistema por la construcción de los enlaces**.
En todo los que sigue, definimos a $n \in \mathcal{N}$ como el número de nodos.
Importante: Esta construcción se hará para redes simples: sin rizos, ni pesos.

Matriz de adyacencia (A).

Es la representación más simple de una red donde cada columna se puede abstraer como el estado de un nodo respecto a los demás.
Matemáticamente, se define como

$$A = \begin{cases} 1, & \text{Si } (i,j) \in E \\ 0, & \text{Si } (i,j) \notin E \end{cases}$$

Matriz de adyacencia (A).

Esta matriz es **simétrica**. Notemos que esta matriz se puede generar en función de la adyacencia de un nodo con los otros; esto eso, por cada nodo, tener un vector con entradas similares. De manera formal,definamos el vector columna $\Delta_u \in \mathcal{N}^{n\times 1}$ para algún nodo $u \in V$,

$$(\Delta_u)_j = \begin{cases} 1, & \text{Si } (u,j) \in E \\ 0, & \text{Si } (u,j) \notin E \end{cases}$$

Matriz de adyacencia (A).

En consecuencia de esta definición, tenemos que

$$\delta_u = \Delta_u^T \Delta_u \tag{1}$$

donde δ_u es el grado del nodo u. Así, la matriz de adyacencia se puede ver como

$$A = \begin{pmatrix} \Delta_{u_1} & \Delta_{u_2} & \cdots & \Delta_{u_n} \end{pmatrix}$$

Sobre la métrica *Clustering*.

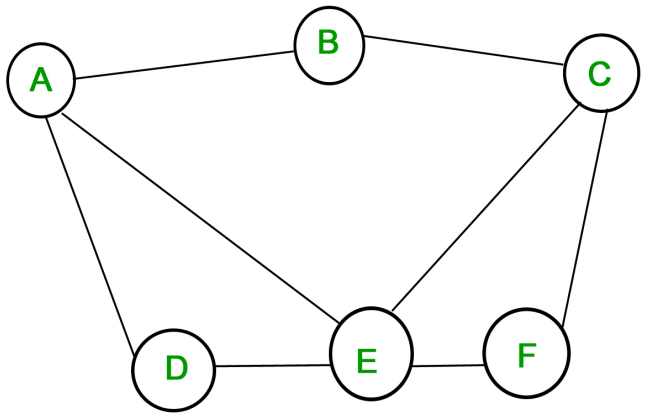
- Centralidad local.
- El *valor* del nodo en función de su primera vecindad. En esencia, calcula la razón de triángulos que están en la gráfica contra los ideales.

$$c_u = \frac{\tau(u)}{\binom{\delta_u}{2}} \tag{2}$$

$$= \frac{2\tau(u)}{(\delta_u)(\delta_u - 1)} \tag{3}$$

Clustering a través de la matriz A.

- Dicha métrica considera el conteo de triángulos.
- Un triángulo es un ciclo de longitud 3.
- la diagonal de A^3 nos dará el doble de estos valores ya que consideramos una red simple.



Si definimos a $diag(A^3) \in \mathcal{N}^{n \times 1}$ como un vector, tenemos que (3) se puede definir como

$$c_u = \frac{diag(A^3)_u}{(\delta_u)(\delta_u - 1)} \tag{4}$$

Resumen en pseudocódigo.

Algorithm 1 Obtención del *clustering* en secuencial.

Require: Matriz de adyacencia A de dimensión $n \times n$

Definimos Vector_grados, Vector_diagonal, vector_clust.

for nodo **do**

Sumamos la entradas del vector Δ_{nodo}.

Guardamos la suma en vector_grados.

end for

Calculamos A^3 tal cual la definición formal con un hilo. ▷ Esta es la parte más tardada.

for nodo **do**

Extraemos el valor de la diagonal de A^3 asociado al nodo.

Guardamos el valor en vector_diagonal.

end for

for nodo **do**

Extraemos los valores en vector_grados (δ_{nodo}) y vector_diagonal asociados a el nodo (D_{nodo}).

Calculamos $\frac{D_{nodo}}{(\delta_{nodo})(\delta_{nodo}-1)}$.

Guardamos el valor en el vector vector_clust.

end for

Figure: Algoritmo secuencial.

Lo que hace Python...

```python
@not_implemented_for("multigraph")
def _triangles_and_degree_iter(G, nodes=None):
    """Return an iterator of (node, degree, triangles, generalized degree).

    This double counts triangles so you may want to divide by 2.
    See degree(), triangles() and generalized_degree() for definitions
    and details.

    """
    if nodes is None:
        nodes_nbrs = G.adj.items()
    else:
        nodes_nbrs = ((n, G[n]) for n in G.nbunch_iter(nodes))

    for v, v_nbrs in nodes_nbrs:
        vs = set(v_nbrs) - {v}
        gen_degree = Counter(len(vs & (set(G[w]) - {w})) for w in vs)
        ntriangles = sum(k * val for k, val in gen_degree.items())
        yield (v, len(vs), ntriangles, gen_degree)
```

Figure: Cálculo de triángulos en Python.

Lo que hace Python...

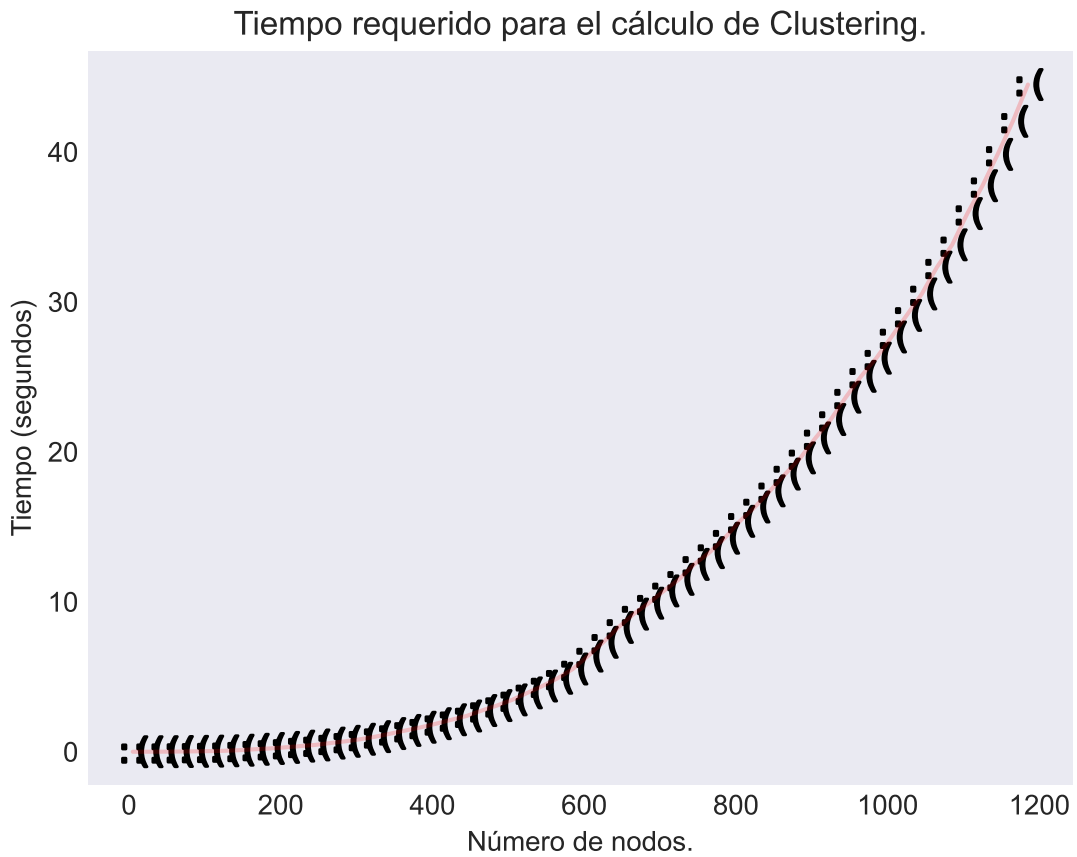


Figure: Tiempos de ejecuación en Python.

Una mejor optimización.

Hay una cosa realmente importante con el algoritmo anterior.

- Tratamos a la matriz A como cualquier otra matriz.

La simetría ayudará a reducir la mitad de los cálculos.

La metodología presentada:

- Optimización del algoritmo anterior.
- Posteriormente, se hará uso de la interfaz OpenMP. Toda la codificación se hará en C.

Otra vez con la matriz A.

Primera propiedad: $(AA)^T = A^T A^T = AA$. Entonces, sólo se requieren $\frac{n(n+1)}{2}$ productos interiores.

$$A^2 = \begin{pmatrix} \delta_{u_1} & \Delta_{u_1}^T \Delta_{u_2} & \Delta_{u_1}^T \Delta_{u_3} & \dots & \Delta_{u_1}^T \Delta_{u_n} \\ \dots & \delta_{u_2} & \Delta_{u_2}^T \Delta_{u_3} & \dots & \Delta_{u_2}^T \Delta_{u_n} \\ \dots & \dots & \delta_{u_3} & \dots & \Delta_{u_3}^T \Delta_{u_n} \\ \dots & \dots & \dots & \delta_{u_{n-1}} & \Delta_{u_{n-1}}^T \Delta_{u_n} \\ \dots & \dots & \dots & \dots & \delta_{u_n} \end{pmatrix} \tag{5}$$

Estas $\frac{n(n-1)}{2}$ entradas serán calculadas por medio de una estrategia de paralelización.

Ahora tenemos A^3 con $A2^2$ y A.

$$A^2A = \begin{pmatrix} \delta_{u_1} & \Delta_{u_1}^T\Delta_{u_2} & \Delta_{u_1}^T\Delta_{u_3} & \cdots & \Delta_{u_1}^T\Delta_{u_n} \\ \cdots & \delta_{u_2} & \Delta_{u_2}^T\Delta_{u_3} & \cdots & \Delta_{u_2}^T\Delta_{u_n} \\ \cdots & \cdots & \delta_{u_3} & \cdots & \Delta_{u_3}^T\Delta_{u_n} \\ \cdots & \cdots & \cdots & \delta_{u_{n-1}} & \Delta_{u_{n-1}}^T\Delta_{u_n} \\ \cdots & \cdots & \cdots & \cdots & \delta_{u_n} \end{pmatrix} \begin{pmatrix} \Delta_{u_1} & \Delta_{u_2} & \cdots & \Delta_{u_n} \end{pmatrix} \tag{6}$$

De esto, sólo ocupamos la diagonal. Así, sólo necesitamos n productos interiores. Aquí se vuelve a paralelizar con un detalle para el resultado final.

Resumen de algoritmo.

Algorithm 2 Optimización y mejoramiento del algoritmo 1.

Require: Matriz de adyacencia A de dimensión $n \times n$

Definimos matriz_a2[n][n], vector_clust[n].
Empezamos una región de paralelización. ▷ Notemos que aquí distribuimos estos cálculos entre los hilos permitidos.
for nodo u_i hasta $nodo_n$ **do**
  for nodo $v_j = u_i + 1$ hasta $nodo_n$ **do**
    if $u_i == v_j$ **then**
      Obtenemos su grado
      Sumamos los valores de Δ_{u_i}
      Lo guardamos en la entrada (i, i) del arreglo matriz_a2
    else if $u_i \neq v_j$ **then**
      Calculamos $\Delta_{u_i}^T \Delta_{v_j}$
      Lo guardamos en la entrada (i, j) y (j, i) del arreglo matriz_a2
    end if
  end for
end for
Terminamos una región de paralelización.
Empezamos una región de paralelización.
for nodo u_i hasta $nodo_n$ **do**
  Calculamos el producto interior de Δ_{u_i} con el renglón i del arreglo matriz_a2. Definimos este valor como D_{u_i}.
  Obtenemos su grado con el valor de la diagonal de matriz_a2 asociado a u_i. Digamos a este valor como δ_{u_i}.
  Calculamos $\frac{D_{u_i}}{(\delta_{u_i})(\delta_{u_i}-1)}$ y lo guardamos en vector_clust.
end for
Terminamos una región de paralelización.

Figure: Algoritmo dos, con paralelización.

Un pequeño cálculo de su complejidad.

- Por parte del algoritmo 1 (secuencial), dado que calculo principal es sobre A^3, tenemos que el orden de complejidad $\mathcal{O}(n^3)$.
- Por otro lado, por parte del algoritmo 2, si bien reducimos el número de operaciones a más de la mitad, tenemos la misma complejidad. Pues, el semi cálculo de A^2 simplificar $\frac{n(n-1)}{2}(2n-1)$ operaciones.

Lo esperable es que notemos una mejora al implementar este último algoritmo en su versión paralela.

Para el testeo de los algoritmos.

- Se usarán redes binomiales.
- Una red binomial es una red donde tenemos una probabilidad p de que exista un enlace.

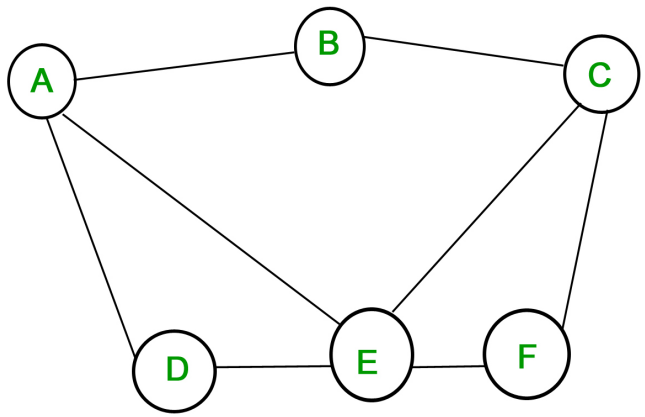


Figure: Ejemplo de red.

Nota Si bien es un tipo de red, la única limitante es la dirección y los pesos.

Para este caso no sirve.

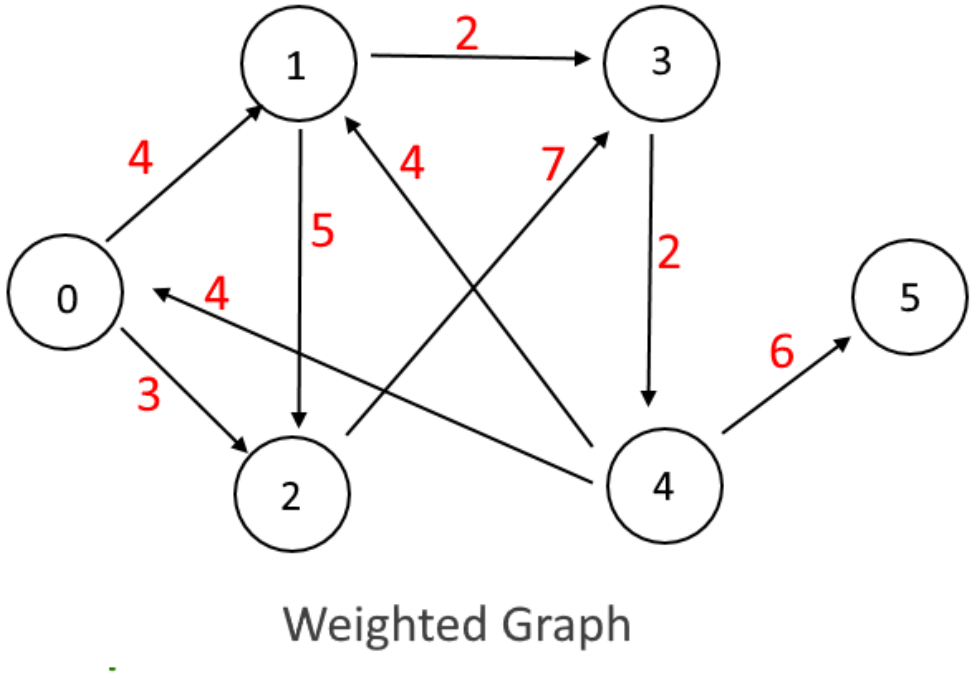


Figure: Ejemplo de una red que no sirve. Tenemos dirección y pesos. Por lo tanto, A puede no ser simétrica.

Resultados: *Rendimiento en tiempo.*

En todos los casos, se hizo con una máquina virtual con 3 hilos; tiene 1 cpu con 3 hilos disponibles.

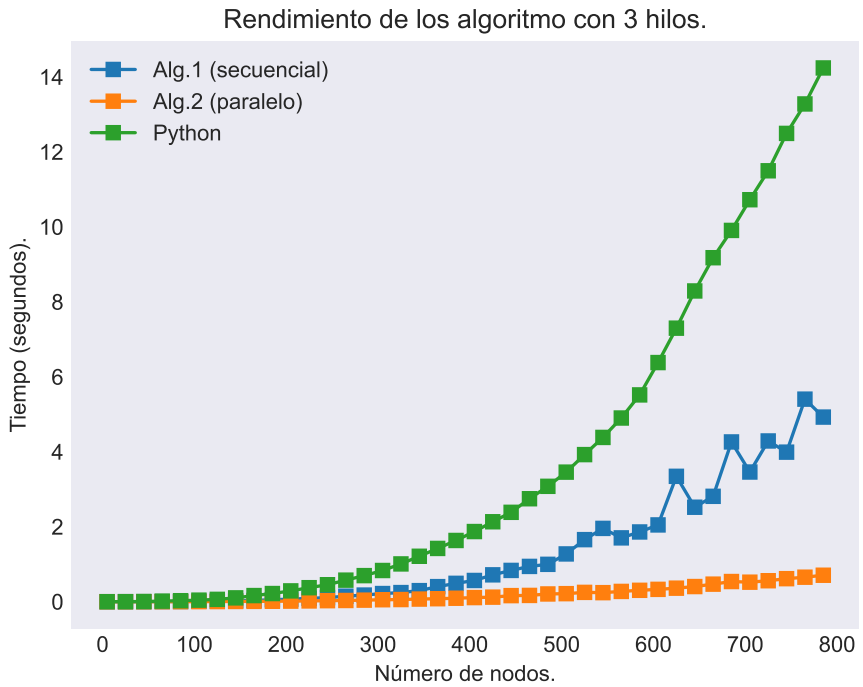


Figure: Comparativa en tiempos.

Resultados: *Speed up*; comparación entre algoritmos.

$$SpeedUp = \frac{\text{Tiempo requerido de una rutina con el algoritmo (a)}}{\text{Tiempo requerido de una rutina con el algoritmo (a)}}$$

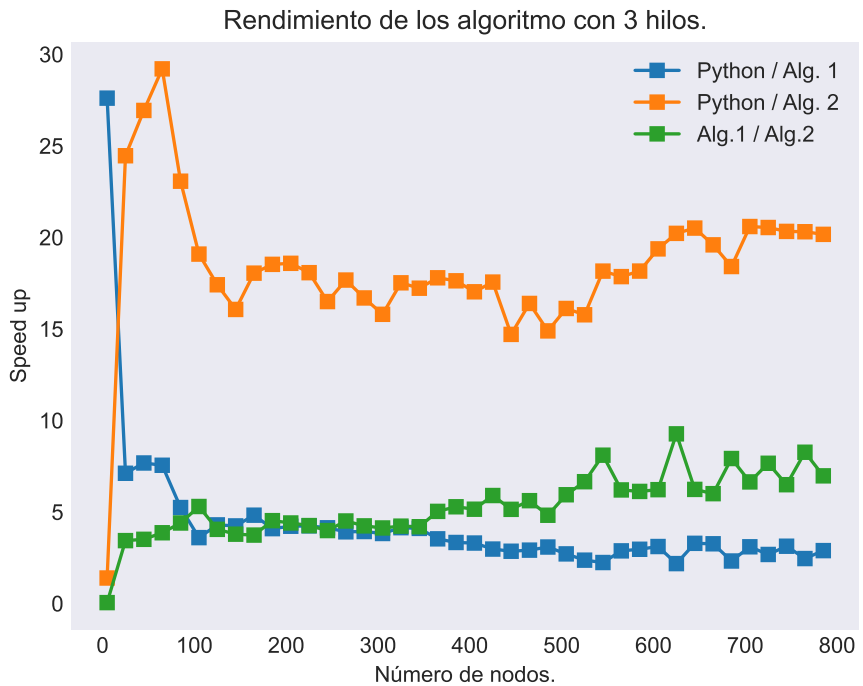


Figure: Comparativa en tiempos.

Conclusiones.

- A experiencia personal, se logra el objetivo que ganarle al algoritmo de Python 10 veces.
- Si bien siempre será bueno paralelizar, será mucho mejor analizar el algoritmo en sí y optimizarlo.
- Este algoritmo se puede escalar a otro tipo de red (dirigidas y con pesos); sólo hay que jugar con la matriz de adyacencia.

Referencias

Melanie Mitchell.

Complexity : a guided tour.

Oxford University Press, Oxford England New York, 2009.

Javier López Peña and Hugo Touchette.

A network theory analysis of football strategies, 2012.