# 1 Ejercicio 1

## 1.1 Explicación Heurística de búsqueda local1

Pudimos observar cuando probamos el algoritmo goloso con grafos chicos que la solución no era óptima por poco, y si sólo movíamos algunos nodo a otros conjuntos sí lo era. De esta forma se nos ocurrió que podíamos mover un nodo a otro conjunto para ver si era mejor la solución.
Así pensamos que la vecindad de una solución podía ser todas las soluciones que difieran en un nodo.

Para explicar mejor el algoritmo que crea la vencindad utilizaremos el siguiente pseudocódigo, en el cual se usan auxiliares:

Una *solución* es un conjunto de conjuntos de nodos, costo_Solución es un entero

```
1: procedure CrearVecindad((solución, int costo_Solución)→ lista(solución))
2:     para todos los conjuntos de nodos del conjunto:
3:     tomar de a uno los nodos dentro del conjunto de nodos
4:     para todo conjunto distinto de donde tomé el nodo:
5:     hacer una copia de la solución
6:     sacar el nodo de la copia
7:     poner el nodo en el conjunto distinto
8:     calcular el costo de la copia modificada.
9:     if (el nuevo costo < costo_Solución) then
10:        armar una tupla (copiaModificada,el nuevo costo)
11:        guardar la tupla en una lista
12:    end if
13:    devolver la lista que armé
14: end procedure
```

```
1: procedure ElegirMejor((lista((solución,costo_Solución)))→ tupla(solución,costo_solución))
2:     Buscar en la lista la túpla que tiene máximo costo_Solución
3: end procedure
```

Grafo es una matriz de $nXn$, donde $n$ es la cantidad de nodos, y en la posición $(i, j)$ está el costo de la arista $(i, j)$

```
1: procedure KPMP_Busqueda_Local((solución solu, entero costo_solu ,entero K)→ tupla(solución))
2:     lista(solución) lsolu= crearVecindad(solu,costo_solu)
3:     devolver eslegirMejor(lsolu)
4: end procedure
```

En el pseudocódigo de KPMP_Busqueda_Local la entrada es una solución, y dos enteros, pero en la implementación la entrada es un grafo (matriz) y un entero. Esto es porque dentro del algoritmo aplicabamos la implementación de la heurística golosa para obtener la solución a la que ibamos a buscar los vecinos o sólo metíamos todos los nodos en un conjunto dejando los otros vacíos. De esta manera podíamos experimentar sin tener que darle a la entrada una solución y nos resultó más comodo.

## 1.2 Complejidad Heurística de búsqueda local1

Para calcular la complejidad de CrearVecindad vamos a calcular la complejidad de los "auxiliares" que utiliza.
En CrearVecindad tenemos:
Línea 2 : La cantidad de conjuntos es $k$, entonces tengo costo $k$, porque usamos vector que tiene costo constante para el acceso(*http://www.cplusplus.com/reference/vector/vector/operator%5B%5D/*).
Línea 3: Recorrer los nodos dentro del conjunto en peor caso es $n$, cantidad de nodos y asumiendo que todos los conjuntos tienen esa cantidad. Para representar los conjuntos utilizamos vector,como el acceso es $O(constante)$(*http://www.cplusplus.com/reference/vector/vector/operator%5B%5D/*),entonces tengo costo n.

Línea 4: Lo mismo que en línea 2, sólo que ahora tengo una comparación (es o no el nodo que saqué) que tiene costo 1. Entonces tengo costo $k$.
Línea 5: Tengo $k$ conjuntos con $n$ nodos, con lo cual tengo costo $n*k$, también porque usamos vector y hacer una copia tiene costo la cantidad de elemntos(*http://www.cplusplus.com/reference/vector/vector/operator=/*).
Línea 6: Buscarlo en el conjunto tiene costo $n$(por usar vector), luego borarlo tiene costo 1(tiene costo la cantidad de elementos por usar un vector *http://www.cplusplus.com/reference/vector/vector/erase/*), con lo cual tengo costo $n + 1$.
Línea 7: Agregarlo en el conjunto a modificar tiene costo 1
(*www.cplusplus.com/reference/vector/vector/push_back/*).
Línea 8: Recorro cada uno de los conjuntos (costo $k$) y calculo "las aristas que se generan dentro" para sumarlas (en peor caso tengo costo $n^2$ todos los nodos con todos). Entonces tengo costo $k * n^2$.
Línea 9: Una comparación de números, costo 1.
Línea 10 y 11: Armar la túpla y meterla en una lista tiene costo 1, porque usamos un vector para representar la lista, usando push_back que tiene costo constante.

Como tenemos muchos "fors" anidados, el costo total es la multiplicación del costo de cada "for" Con lo cual la complejidad de $CrearVecindad$ es $k^3 * n^3$ .

ElegirMejor tiene costo la cantidad de elemenos que hay en la lista: Para cada conjunto, cuya cantidad es $k$, voy a tomar cada uno de los elementos, en peor caso tengo $n$, y los voy a meter en cada uno de los conjuntos. Con lo cual en peor caso son $k^2 * n$ elementos dentro de la lista, y como suamos vector para representarla tenemos costo la cantidad de elementos, con lo cual tengo costo $k^2 * n$

Finalmente el costo de *KPMP_Busqueda_Local* es la suma del costo de $CrearVecindad$ y $ElejirMejor$,ya que devolver la tupla tiene costo 1 y la forma de crear la solución no es relevante porque , entonces tenemos $O(k^3 * n^3)$

## 1.3 Explicación Heurística de búsqueda local2

Para la resolución se cuenta inicialmente con una K-partición con sus elementos (nodos) distribuidos de alguna forma, la llamaremos *solucion*. Lo que se va a proceder a hacer es tratar de mejorar el peso total de esta K-partición (con peso total nos referimos a la suma de los pesos de todos los conjuntos de dicha partición). Reubicando sus nodos de manera que se minimice dicho peso. Para esto y siguiendo la idea de búsqueda local se construirán varias posibles soluciones, vecinos, y nos quedaremos con el mejor de ellos (el mejor es aquel, cuyo peso total sea el mínimo de entre los vecinos que constituyen la vecindad). Para la resolución, empezaremos tomando al primero de los conjuntos que constituyen a *solucion*. Una vez tomado, se trabajara con los nodos dentro del mismo y de a par, siempre que estos sean adyacentes. Si no hay nodos adyacentes en el conjunto, entonces se devolverá como solucion y vecino, a *solucion* (no se producen cambios). En caso de existir al menos un par de nodos adyacentes. Evaluaremos el peso que se genera entre la arista que une a este par y el peso de las aristas a los nodos adyacentes dentro del conjunto actual, *peso_combinacion*. Calculado este último peso, nos quedaremos con aquel cuyo valor es máximo, *mayor_peso*. Una vez hallado este par de nodos, a los que llamaremos a uno *nodo_A* y otro *nodo_B*, crearemos otra K-partición, *nuevo_vecino*. El mismo sera una copia de la *solucion* actual, solo que no contara con los nodos denominados *nodo_A* y *nodo_B*. Y lo que se va a proceder a hacer es a reubicar a ambos nodos en aquel o aquellos conjuntos tal que el peso final de *nuevo_vecino* sea mínimo. Y tendremos una posible solución. Finalizado esto, se procede a operar de la misma manera pero, ahora trabajando con el siguiente conjunto. De esta forma vamos a obtener un nuevo vecino. Luego de haber trabajado con los k conjuntos elegiremos a aquel vecino de todos los formados cuyo peso total sea el mínimo. Como *solucion* es hasta el momento la solución para el problema, se comparara si el peso del vecino elegido es menor que este. Si no lo es, no se pudo obtener una mejor ubicación que la que ya se tenia para los nodos y se finaliza el algoritmo siendo *solucion* la solucion a nuestro problema. En caso de serlo *solucion* pasara a ser el vecino seleccionado. Y se comenzara a ejecutar nuevamente lo propuesto inicialmente, pero ahora con la K-partición *solucion* modificada. Para poder seguirla optimizando.

1: **procedure** VECINDAD2(*..solucion, total de conjuntos,..*)
2: *suma_solucion* ← *sumar el peso de la solucion*
3: **if** (*solucion no usa los k conjuntos*) **then**

```
        agregar conjuntos restantes vacios a solucion
    end if
    vecino_solucion ← crear una nueva k_particion
    posible_solucion ← −1
    while (se pueda mejorar suma_solucion) do
        for i= 1,...,total de conjuntos do
            if (solucion[i] tiene mas de un nodo) then
                nuevo_vecino ← crear una nueva k_particion
                peso_vecino ← crear_vecino(...,i, ,solucion, vecino_actual,..)
            end if
            if (peso_vecino < posible_solucion o posible_solucion == −1) then
                posible_solucion ← peso_vecino
                vecino_solucion ← nuevo_vecino
            end if
        end for
        if (posible_solucion < suma_solucion) then
            suma_solucion ← posible_solucion
            solucion ← vecino_solucion
            posible_solucion ← −1
        else
            return solucion
        end if
    end while
    Return solucion
end procedure
```

```
procedure CREAR_VECINO(..i, solucion, vecino_actual..)
    for cada combinacion entre par de nodos adyacente dentro de solucion[i] do
        nodo_A ← tomar un nodo de solucion[i]
        nodo_B ← tomar un nodo adyacente a nodo_A en solucion[i]
        peso_combinacion ← sumar el peso de la arista entre el nodo_A y el nodo_B y el peso de las aristas
    de los nodos adyacentes a nodo_A y nodo_B dentro de solucion[i]
        mayor_peso ← el mayor peso_combinacion
        separo_A ← nodo_A con mayor peso_combinacion
        separo_B ← nodo_B con mayor peso_combinacion
    end for
    nuevo_vecino ← solucion
    sacar a los nodos separo_A y separo_B del conjunto nuevo_vecino[i]
    ubicar a los nodos separo_A y separo_B entre los conjuntos de nuevo_vecino de manera de minimizar
    el peso total
    return peso total de nuevo_vecino
end procedure
```

*tanto solucion[i] como nuevo_vecino[i] hacen referencia al i-esimo conjunto de cada K-partición. Con los conjuntos enumerados de 1 al total de conjuntos.*

## 1.4 Complejidad Heurística de búsqueda local2

Para representar la K-partición *solucion*, se utilizo un vector de tuplas. donde la segunda componente emula un conjunto, por medio de un vector de int, y la primera a la suma del peso de dicho conjunto. El algoritmo cuenta con dos funciones principales que son las mencionadas *vecindad*2 y *crear vecino*, como el mismo se ejecuta hasta que ya no se pueda mejorar al vector *solucion*. Analizaremos la complejidad de cada iteración.

1) Complejidad de *vecindad*2:
a) Como se trabaja a partir de una soluciónón dada, la misma quizás no utiliza la cantidad total de conjuntos

con que dispone. Entonces se procede a completar al vector *solucion* con las tuplas restantes. O(k)
b) Se crear un vector vacío, que almacenara al mejor de los vecinos. O(1) y se comienza con las mencionadas iteraciones.
c)Por cada conjunto de *solucion* se llama a la función crear vecino. Es decir se la llama k veces (siendo k la cantidad de conjuntos).
d) Cada vez que se obtiene un *nuevo_vecino* y su peso, se verifica si es menor que el peso de los anteriores vecinos o si es el primer vecino creado. En caso de serlo se guarda este peso 0(1) y tambien a *nuevo_vecino* en *vecino_solucion* O(k+n) (n cantidad de nodos en total). Como esto se realiza a lo sumo para todos los conjuntos tenemos O(k*(k+n))
e) Finalizada estas iteraciones, se verifica si el peso de *vecino_solucion* es menor que el peso de *solucion* en caso de serlo se guarda el peso y se reemplaza a *solucion* O(k+n). Y se vuelve a realizar una nueva iteración.

2) Complejidad de *Crear vecino*:
a) Esta función comienza por buscar a los pares de nodos que sean adyacentes y calculando su *peso_combinacion*. Para obtener todas las combinaciones se emplearon dos for anidados, el primero de ellos va desde 1 al total de nodos en el conjunto que se esta utilizando, denominaremos t a esta cantidad con 0 <= t < n. El segundo comienza desde el siguiente nodo que se este utilizando en el primero de los for hasta t. De esta forma obtenemos todas las combinaciones. Pero, por cada una hay que sumar el peso de la arista que los une y el peso de sus adyacentes. Para calcular este último, se vuelve a recorrer a todos los nodos y si son adyacentes se suma el peso de la arista. El mismo se encuentra en una matriz a la que podemos acceder en O(1). De esta manera se realizan, para generalizar, dos for de 1 hasta t. y en cada iteracion una suma de t elementos en total O($t^3$). Comparando cada *peso_combinacion* realizado para guardarnos el máximo O(1).
b) Procedemos a copiar al vector *solucion* a *nuevo_vecino* O(k + n), y a retirar del i-esimo conjunto (i-esima tupla) de este último vector a *nodo_A* y *nodo_B* O(t). Y se actualiza la primer componente con la suma correspondiente. Que es la cantidad de aristas, en el peor de los casos O($t^2$).
c) Para encontrar la mejor ubicación del par de nodos es necesario ver todos las combinaciones con los k conjuntos y ubicarlo donde el peso total generado sea el mínimo que con alguna otra combinación. Para representar esto, se crearan dos vectores de longitud k (cada posición hace referencia a un conjunto de *nuevo_vecino*). El primero posee en cada posición el peso que se adiciona al meter al *nodo_A* en cada conjunto y el segundo al adicionado por insertar a *nodo_B*. De esta manera, se busca la combinacion de estos vectores donde el valor de la posición de uno mas la del otro sea la mínima O($k^2$). Y se guarda las posiciones elegidas.
d) Por último se inserta al par de nodos en las posiciones elegidas dentro de *nuevo_vecino* y se calcula el peso total de este vector O(k)

El costo total de esta función es por lo tanto O($t^3$) + O(k + n) + O(t) + O($t^2$)+ O($k^2$) + O(k) = O($t^3$) + O($k^2$) + O(n)
t es como máximo igual a n. Acotamos t por n y obtenemos:
O($n^3$ + $k^2$)
Pero esta función como se menciono en el punto 1.c es llamada k veces entonces tenemos:
O(k*($n^3$ + $k^2$))
Agregando los costos de la función *vecindad*2 tenemos en total:
O(k) + O(k*($n^3$ + $k^2$)) + O(k*(k + n)) + O(k + n) = O(k*$n^3$ + $k^3$)

# 2 Experimentación de las heurísticas locales

Primero vamos a corroborar empíricamente que las complejidades que calculamos teóricamente son correctas. Para esto generamos grafos completos de 1 a 100 nodos, con los pesos de las aristas un número aleatorio entre 0 y 50(en $\Re$).Luego aplicamos las heuristicas locales 5 veces, calculamos promedio, máximo y mínimo tiempo de las 5 veces por cada instancia. Obtuvimos los siguientes gráficos:
***Para la primera heurísica local:***

Aplicamos el algoritmo con $k = 10$ y las instancias de grafos mencionadas anteriormente.

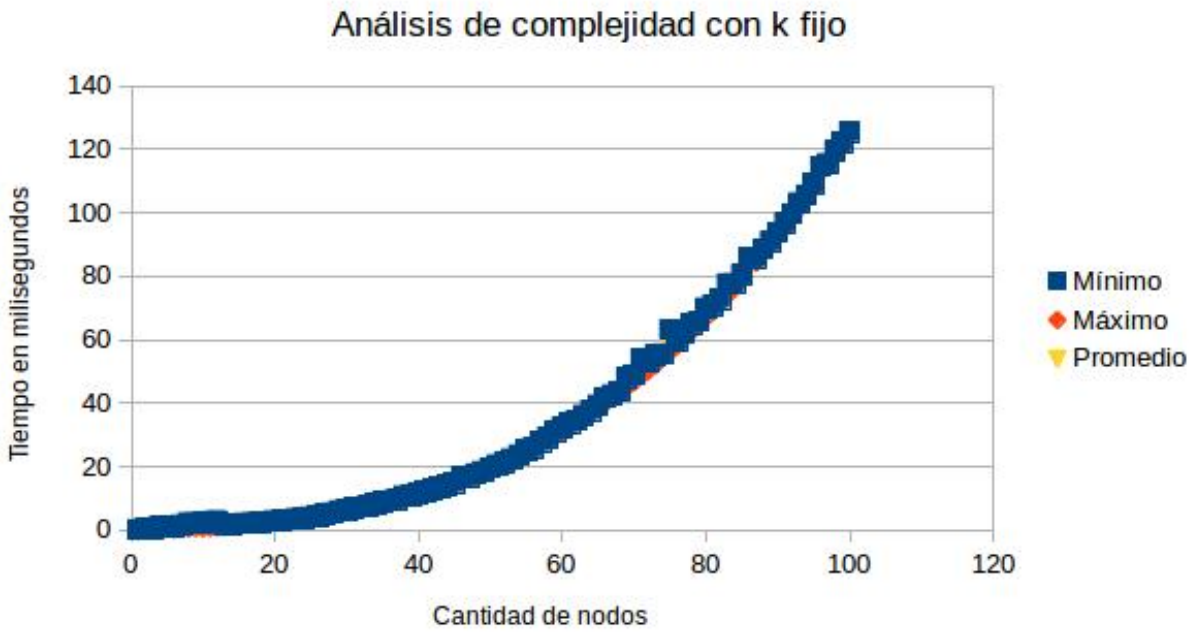


**Figure 1:** Con este grafo no podemos ver si realmente es $O((k^3 * n^2) + (k^2 * n^3))$, para poder evidenciarlo mejor vamos a dividir el tiempo por $O((k^3 * n^2) + (k^2 * n^3))$ y deberíamos obtener una constante.
Además podemos comentar que no hay diferencia notable entre el máximo, mínimo y promedio.

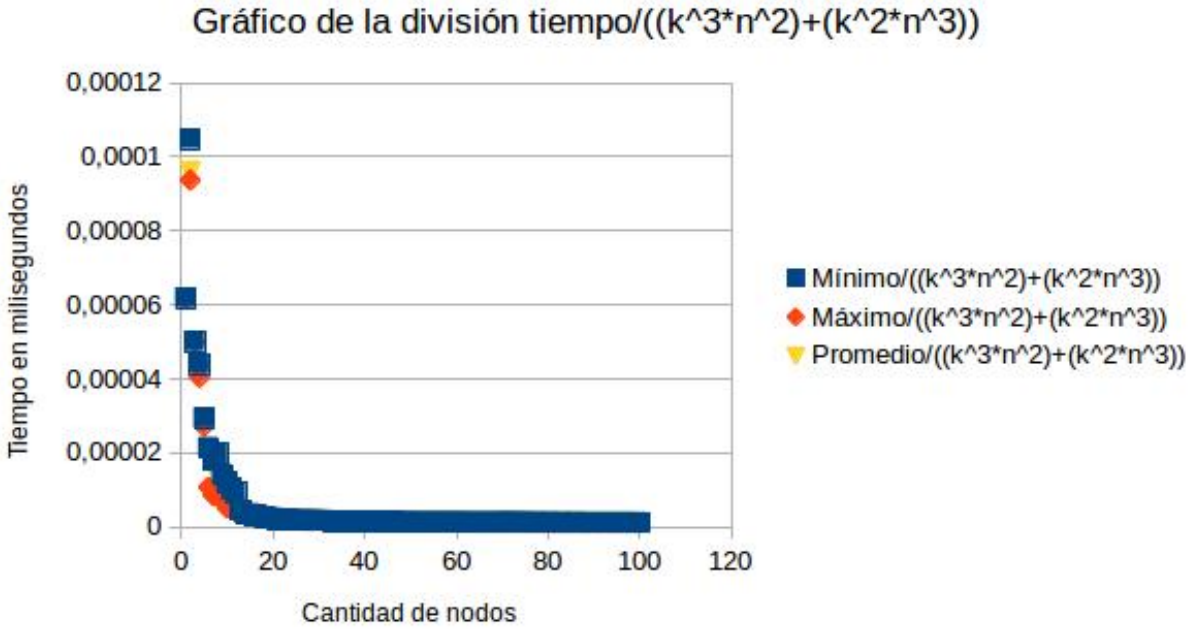


**Figure 2:** Vemos que tiende a una constante, pero esta es 0 a simple vista, y si realmente tiende a 0 no es nada bueno, veamos que la constante es distinta a 0. Para esto vamos a hacer "zoom" en 0 y ver que las líneas no están ahí.

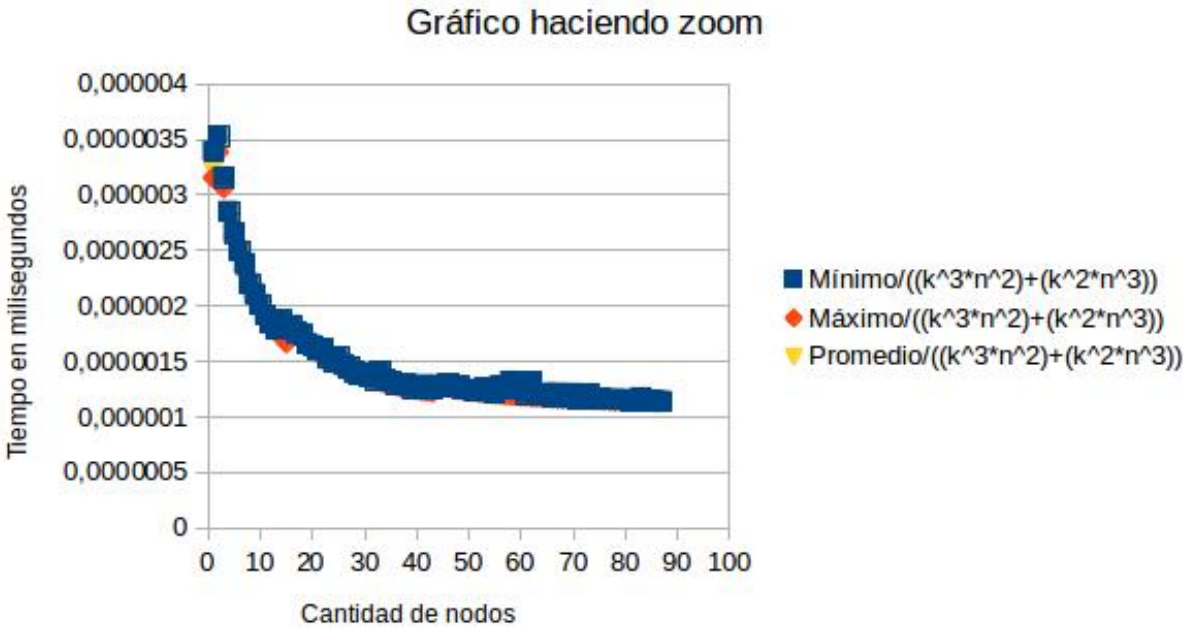


**Figure 3:** Podemos ver así que la constante a la que tiende está entre o,000002 y 0,000005 que es distinta de 0. Con esto podemos concluir que la complejidad explicada teóricamente, empíricamente también lo es.

***Para la segunda heurísica local:***

Aplicamos el algoritmo con $k = 10$ y las instancias de grafos mencionadas anteriormente.

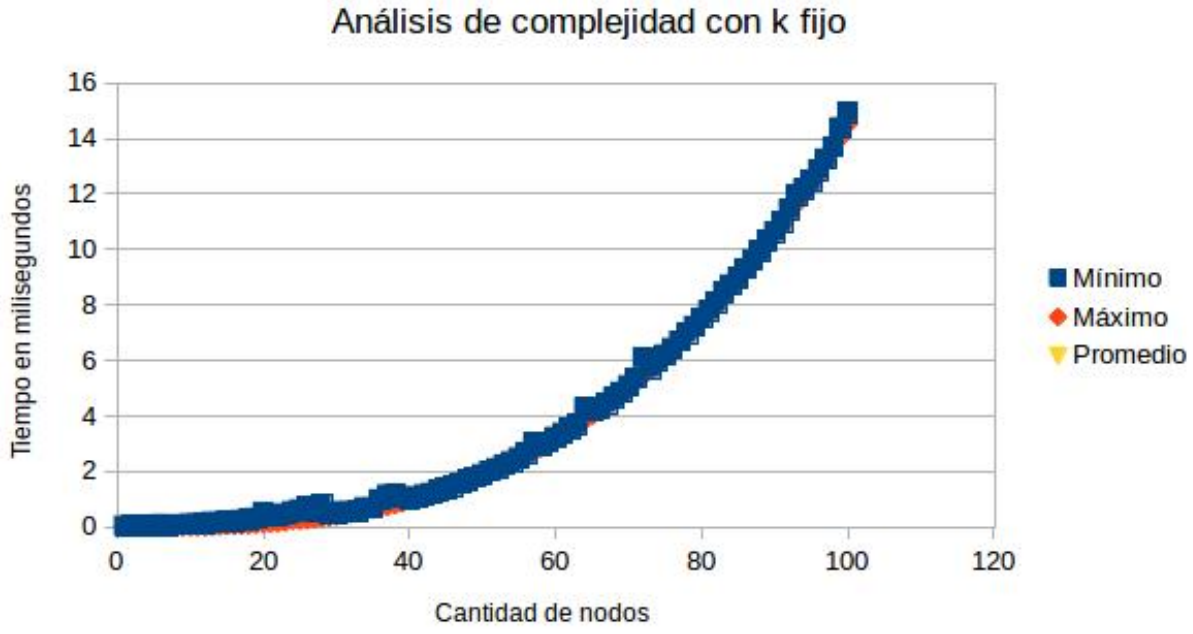


**Figure 1:** Con este grafo no podemos si realmente es $O(k^3 + k * n^3)$, para poder evidenciarlo mejor vamos a dividir el tiempo/$(k^3 + k * n^3)$ y deberíamos obtener una constante.
Además podemos comentar que no hay diferencia notable entre el máximo, mínimo y promedio.

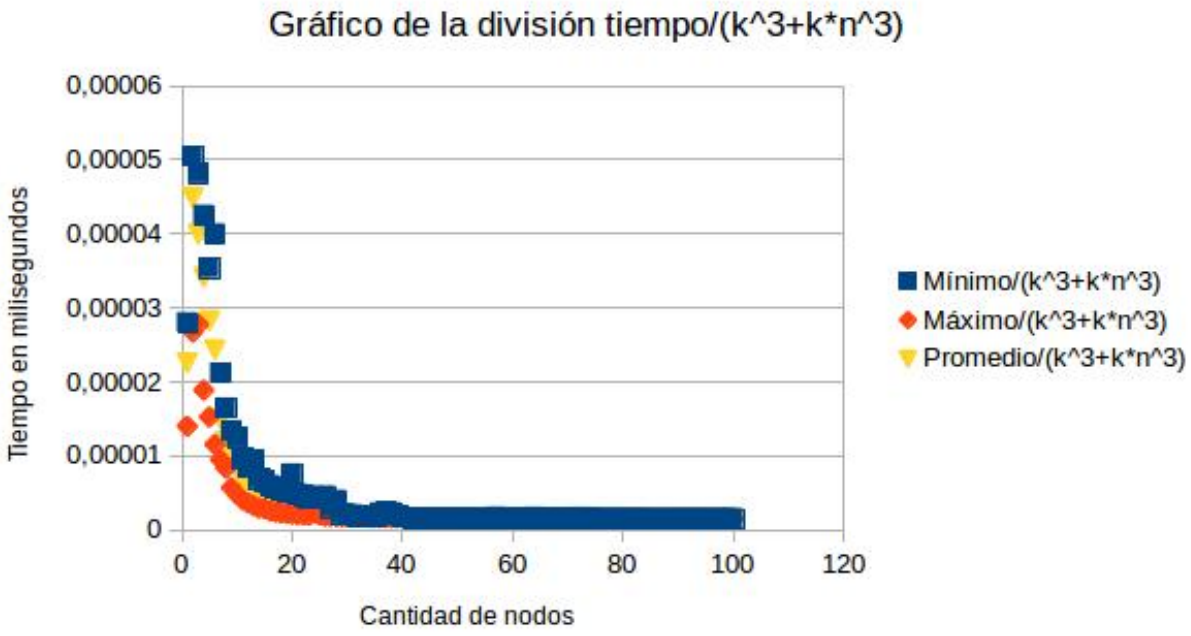


**Figure 2:** Vemos que tiende a una constante, pero esta es 0 a simple vista, y si realmente tiende a 0 no es nada bueno, veamos que la constante es distinta a 0. Para esto vamos a hacer "zoom" en 0 y ver que las líneas no están ahí.
Aquí si se puede ver la diferencia entre promedio,máximo y mínimo pero es despreciable.

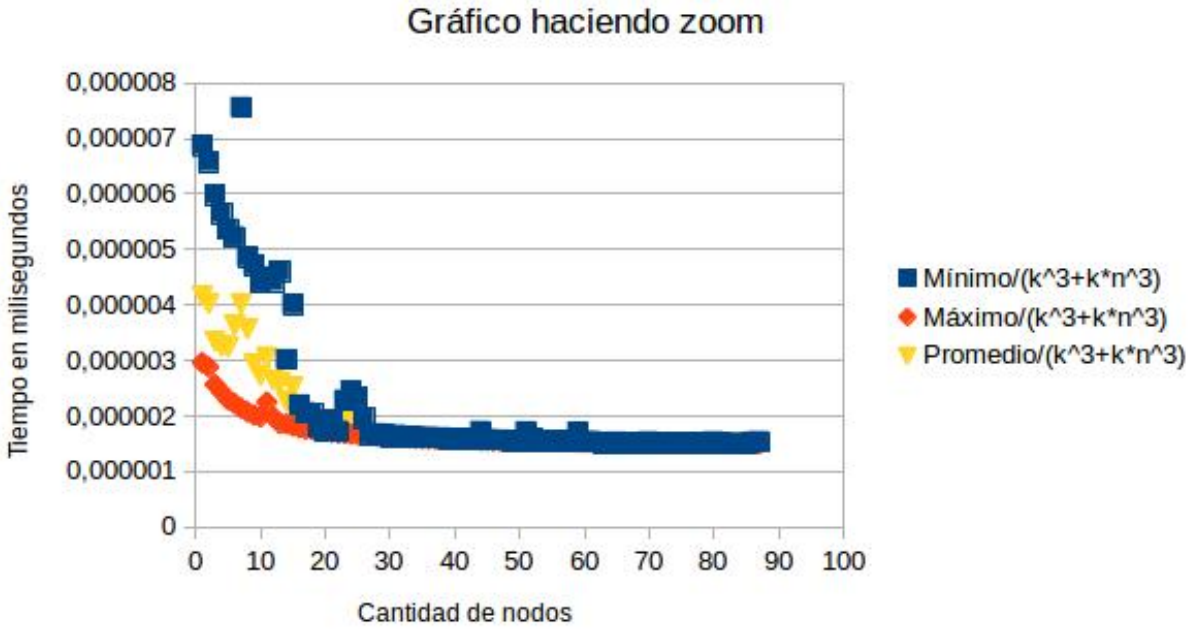


**Figure 3:** Podemos ver así que la constante a la que tiene está entre o,000002 y 0,000001 que es distinta de 0. Con esto podemos concluir que la complejidad explicada teóricamente, empíricamente también lo es.

***Experimentación para ambas locales:***
Para experimentar usamos gráfos aleatoizados, es decir, la forma que tiene el grafo es aleatoria al igual que

el peso de las aristas. Generamos 50 instancias de estos grafos con aristas en un rango de 0 a 50(en $\Re$) y la cantidad de nodos en un rango de 1 a 50 y otras 50 con el mismo rango de aristas pero con 50 nodos. Luego ejecutamos las heurísticas locales 5 veces para cada una de las 50 instancias, quedándonos con el promedio del tiempo de las 5 para cada instancia. Así obtuvimos los tiempos con los que realizamos los siguientes gráficos.

Notamos como *local*1 a la aplicación del algoritmo de la primera vecindad, donde la solución que recibe, es meter todos los nodos en un conjunto y dejar los otros vacíos. Lo mismo para *local*2 pero aplicando el algoritmo de la segunda vecindad. Por otra parte *local*1*Golo* tomo como solución la aplicación del algoritmo goloso y luego a la solución modificada le aplica la primera vecindad, lo mismo para *local*2*Golo* pero aplicando la segunda vecindad. Además cabe aclarar que los tiempos no son de una iteración del algoritmo, sino, de toda la ejecución.

Para $k$ aleatorio en un rango de 2 a $i$, donde $i$ es la cantidad de nodos del grafo sobre el que se aplica el algoritmo.Y las instancias de grafos son los que varían en la cantidad de nodos:

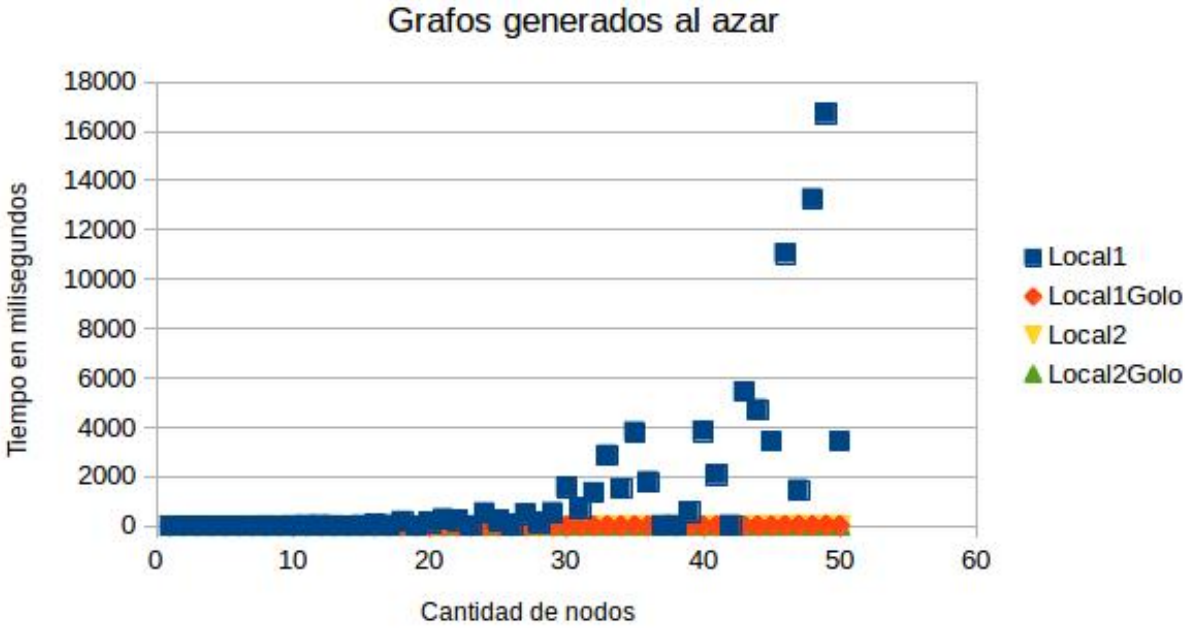


**Figure 4:** Vemos claramente que *local*1 tarda mucho más en comparación con las demás que tardan práctiamente lo mismo a simple vista, veamos que pasa cuando hacemos "zoom" para ver los tiempos más chicos.

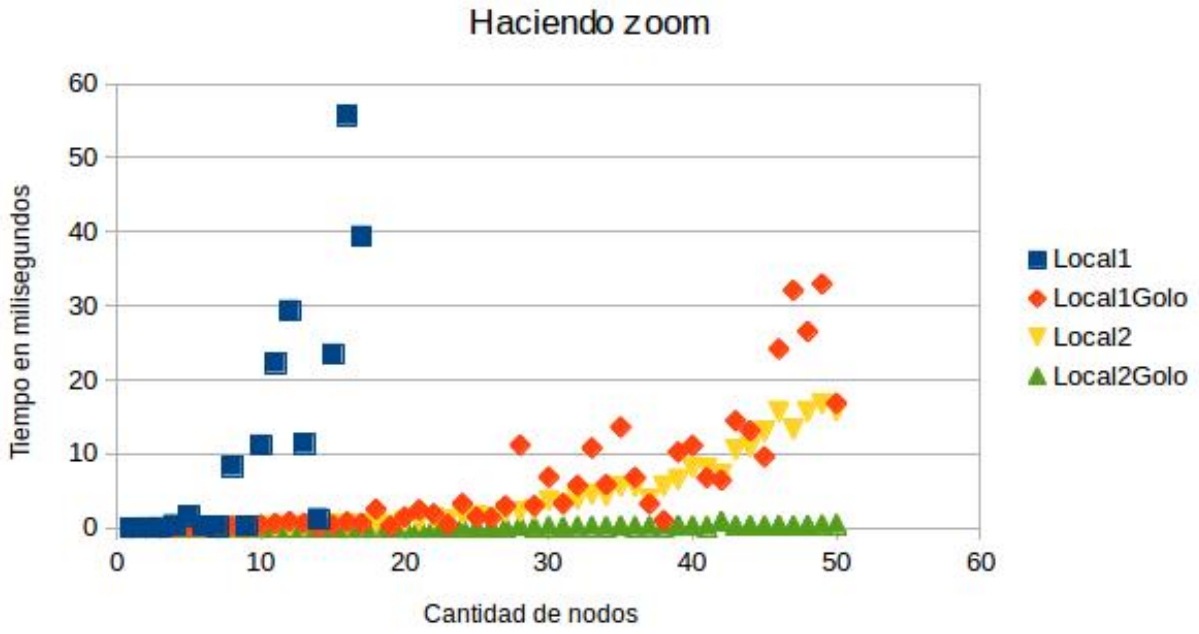


**Figure 5:** Se puede observar que local2Golo es la más rápida, entre local1Golo y local2 hay altos y bajos, pero pareciera que local1Golo se hace más lenta cuando n aumenta.

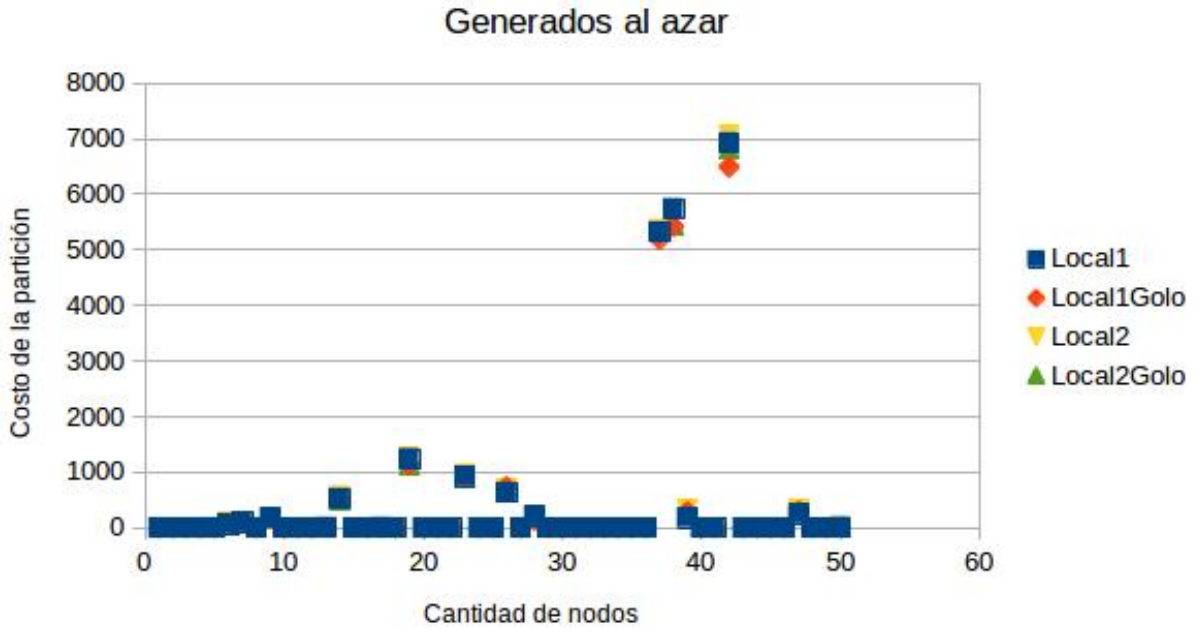


**Figure 6:** En cuanto al costo de la partición se puede ver que son prácticamente iguales, para todos los algoritmos. En los grafos anteriores vimos que *local*1 tarde notablemente mucho más que los otros, y acá podemos ver que el resultado es prácticamente el mismo. Entonces no vale la pena invertir tanto tiempo en *local*1 si con los otros podemos obtener casi lo mismo en poco tiempo, en comparación.

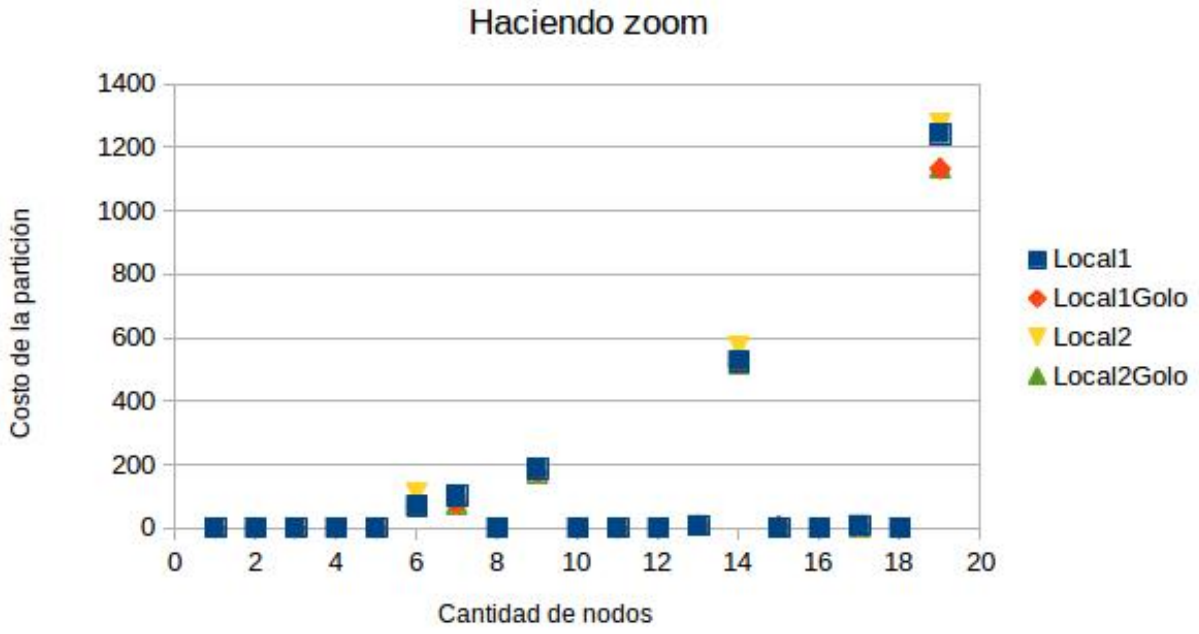


**Figure 7:** Hicimos algo de "zoom" para ver si realmente la diferencia era muy poca, y se ve claramente que la mayoría de las veces es casi lo mismo.

Para $k = 3$, y las instancias de grafos son los que varían en la cantidad de nodos:

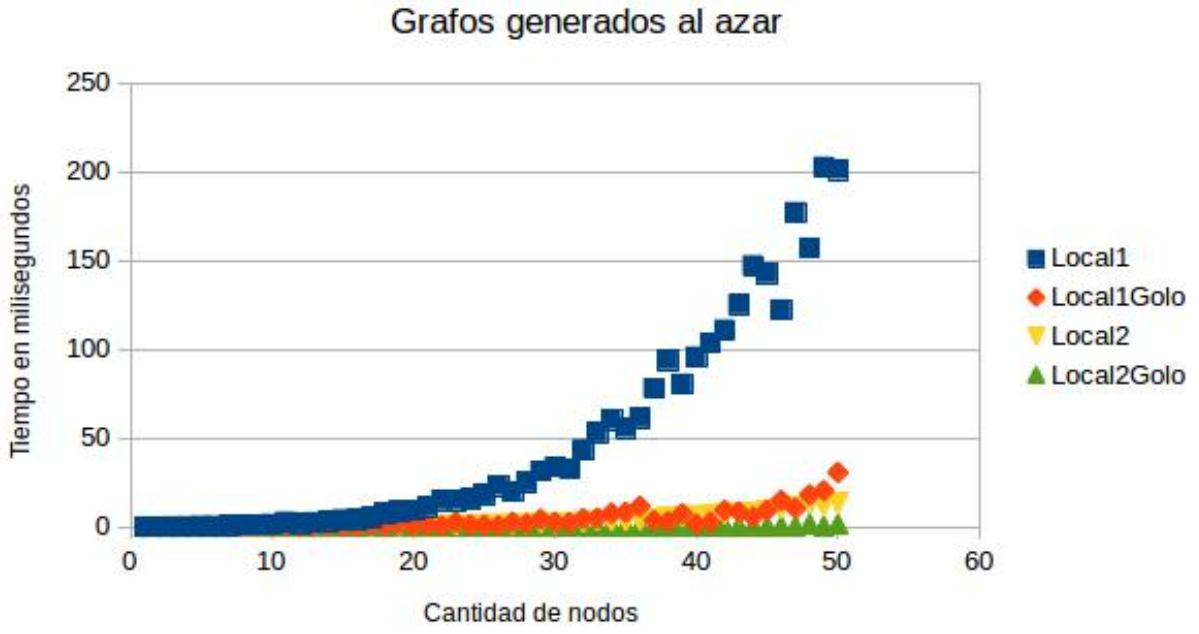


**Figure 8:** No podemos decir más que pasa lo mismo que en las instancias anteriores, *local*1 es más lenta que todas las demás en comparación. Cabe mencionar que al ser k fijo y no aleatorio como antes, el tiempo de ejecución es mucho menor.

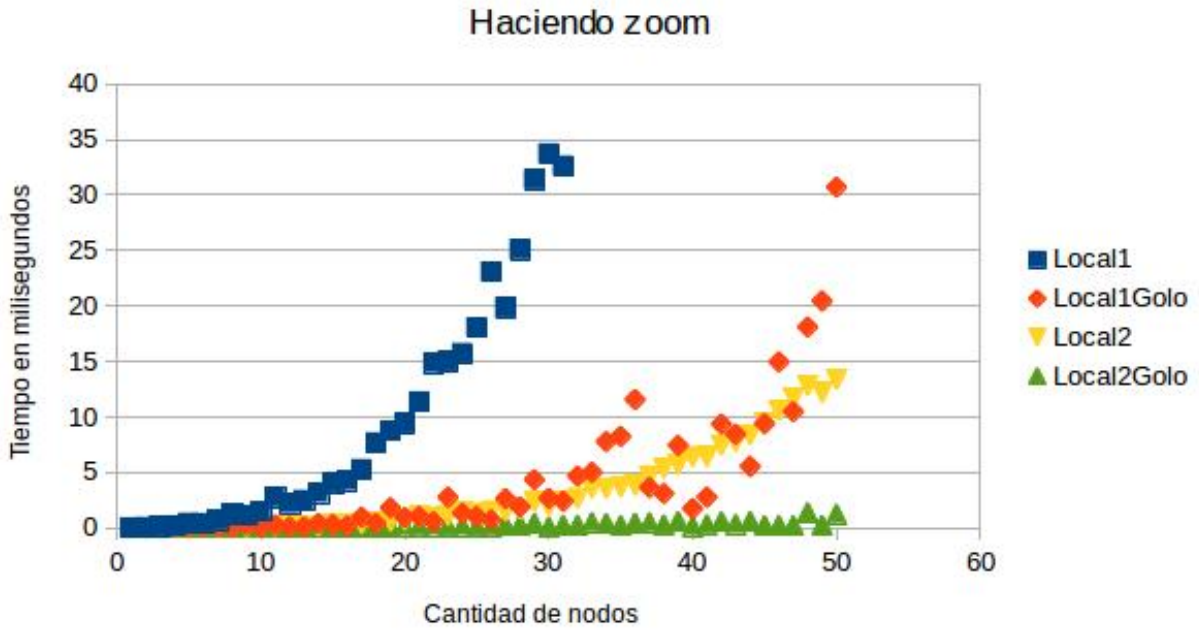


**Figure 9:** Vemos que *local*1*Golo* y *local*2 tienen un intercambio de "mejor" al principio, pero luego de $n = 45$ la mejor es *local*2.

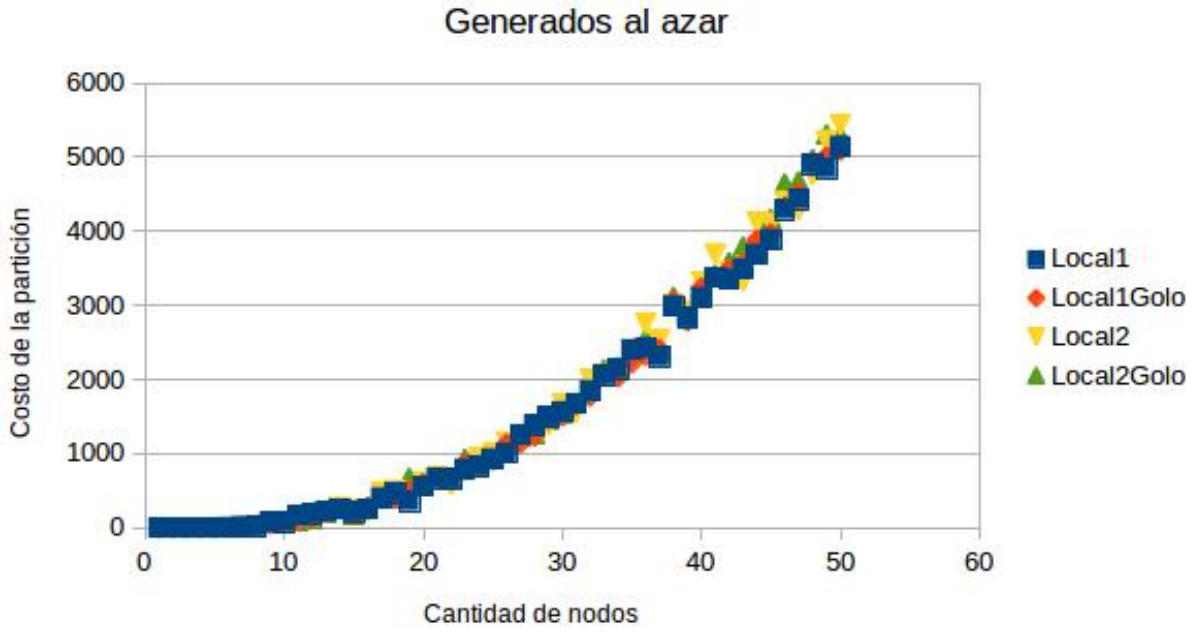


**Figure 10:** Como era de esperar el costo de la partición aumenta cuando n aumenta, porque k es fijo, entonces "hay menos lugar" para poner los nodos sin que estos generen aristas que suman al costo de la partición.

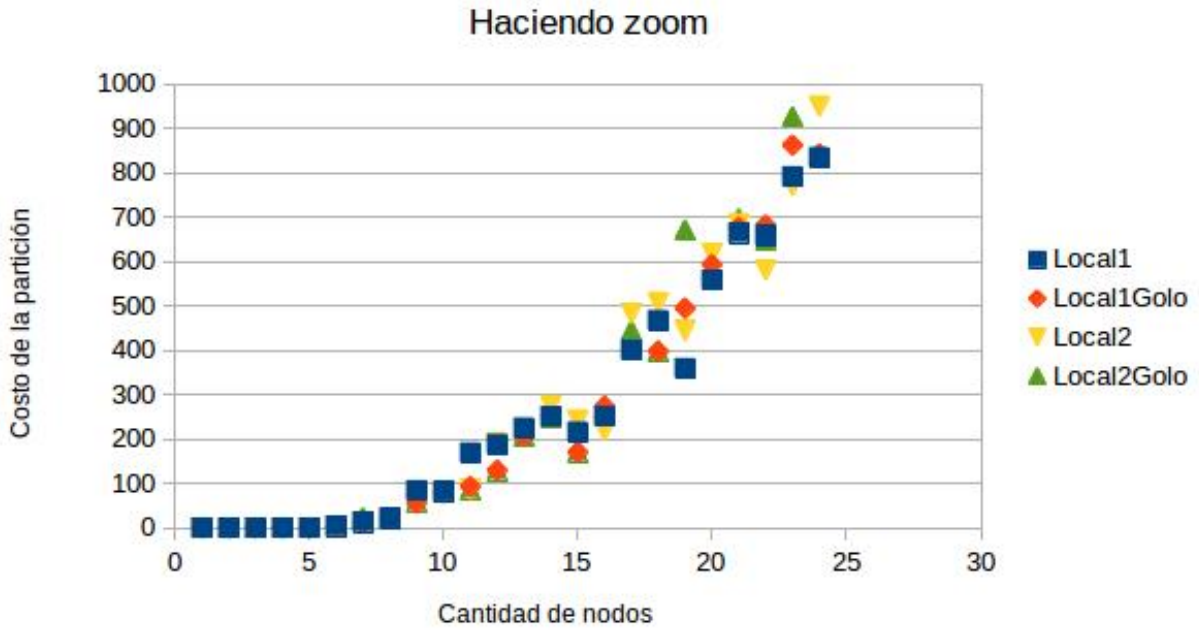


**Figure 11:** Algoritmos son más eficientes, es decir, dan resultados con menos costo. La diferencia entre ellos es casi despreciable cuando n es chico pero a medida que n aumenta se hace más notable la brecha entre ellos.

Finalmente la última experimentación es con $k$ en el rango de 1 a 50 y los grafos con las intancias con n fijo:

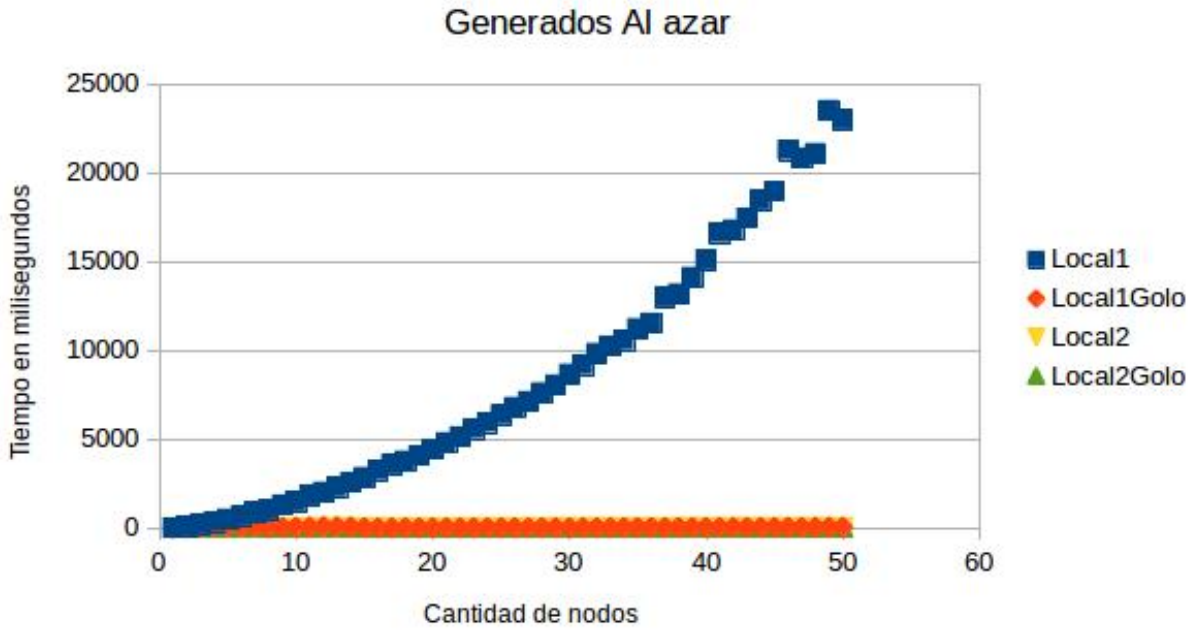


**Figure 12:** Como era de esperar aumenta en función de k, claramente *local*1 es la "perdedora" en toda esta experimentación. Las demás locales son iguales a simple vista, veamos un poco más los tiempos menores.

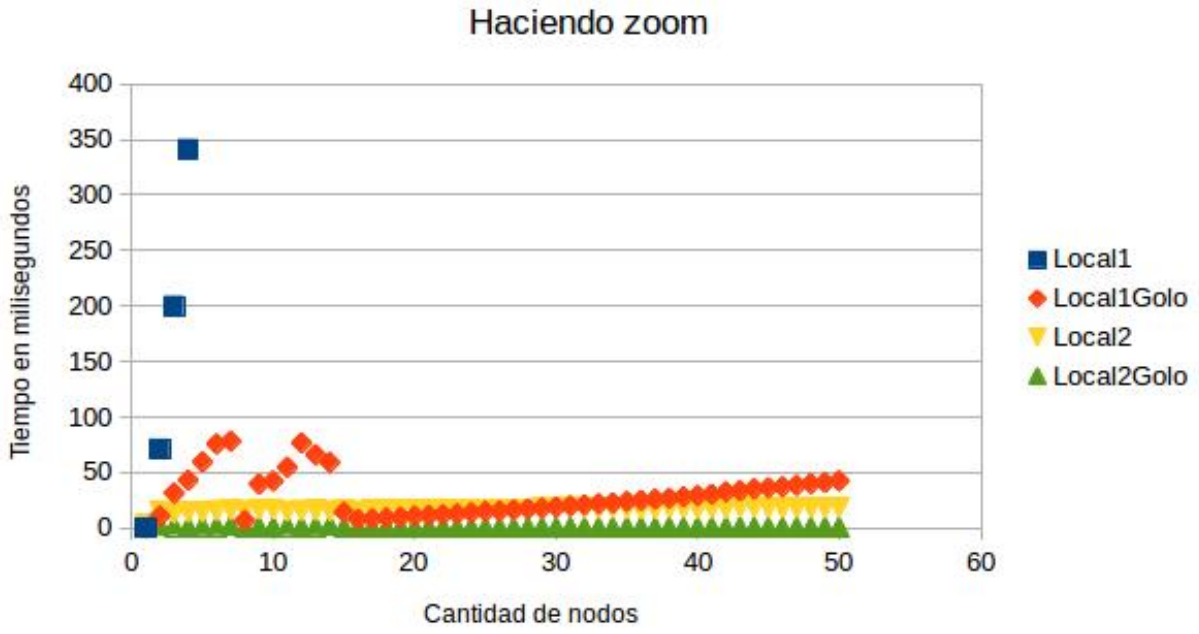


**Figure 13:** Si bien en el grafo anterior parecía que todas menos *local*1 eran iguales acá podemos ver que no es así.


**Figure 14:** Si la cantidad de nodos es fija, y aumento la cantidad de conjuntos para separarlos, se ve claramente que tengo más posibilidades para ponerlos sin que se generen aristas en los conjuntos. Este grafo deja en evidencia que esto es así, pero veamos que sucede en el medio, cuando no tengo ni pocos ni muchos conjuntos($k$).

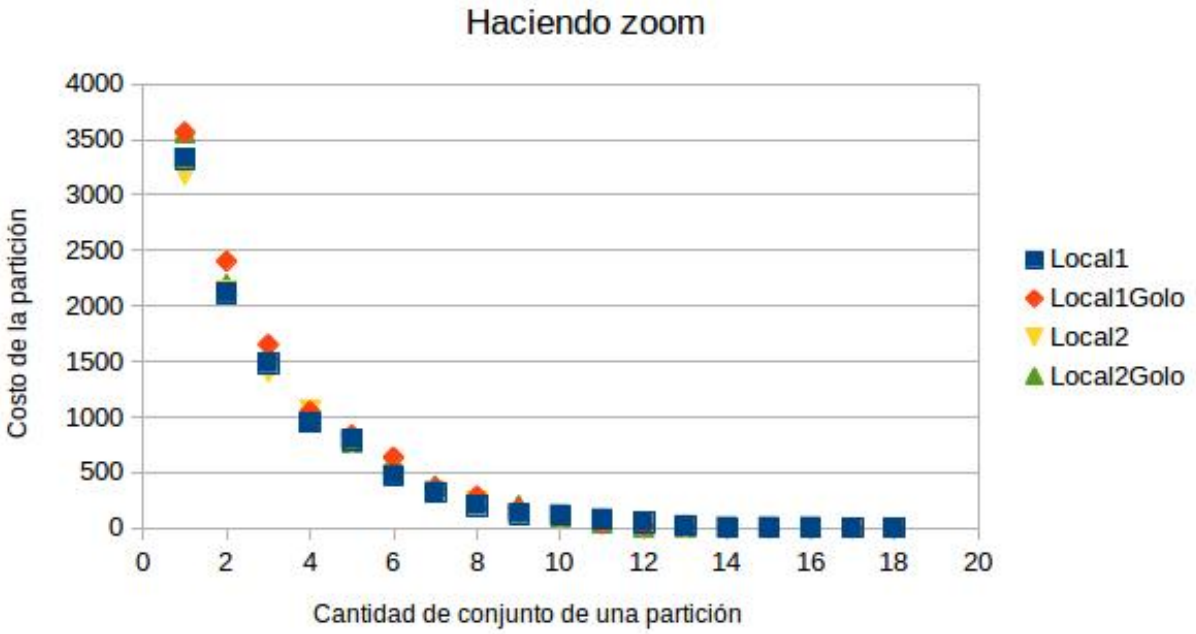


**Figure 15:** Para todas las locales los resultados son casiiguales, e iguales al "final"

En conclusión como vimos a lo largo de toda la experimentación *local*1 es la que más tarda en comparación con las otras 3 y para colmo esta inversión en tiempo no da una solución mejor. Entre las otras no hay una diferencia drástica en cuanto a tiempo y menos en costo, porque la siempre estaban "pegadas".
En fin, no usaríamos *local*1 teníendo alguna de las otras 3, y como mejor elegimos a la *local*2*Golo* que fue la mejor en tiempo en los 3 experimentos que realizamos.