Trabajo Práctico 3

Viernes 19 de Diciembre de 2014 — Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Florencia Lagos	313/12	cazon_88@hotmail.com
Martín Caravario	470/12	martin.caravario@gmail.com
Federico Hosen	825/12	fhosen@hotmail.com
Bruno Winocur	906/11	brunowinocur@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellon I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autonoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Método de experimentación.

Para experimentar en todos los ejercicios utilizamos la libreria *chrono* de c++, la cual nos brindó funciones para medir los tiempos en microsegundos.

El método utilizado consistió en correr nuestro algoritmo una cantidad fija de veces, quedandonos con el tiempo minimo que tardo nuestro algoritmo en resolver la instancia.

2. Punto 1

2.1. a)

El problema 3 del TP 1 consistía en minimizar la cantidad de camiones necesarios para transportar n productos químicos sabiendo que entre cada par de productos existe una cierta *peligrosidad* y que los camiones poseen un *umbral* de peligrosidad determinado.

Pensando en el problema de k-PMP, vamos a redefirnir el Problema 3 en términos de grafos.
Los *productos* van a ser los *nodos*. El peso w, de la arista estaría represenando al coeficiente de *peligrosidad* que existe entre dos productos. y por último la cantidad de camiones a utilizar estaría siendo representado por el numero k, de las k-particiones siendo las particiones los camiones con los productos.

La diferencia en estos dos problemas, es que en el Problema 3 TP 1, lo que se busca es minimazar el k, y no la sumatoria de las peligrosidades en total. Además las particiones deberían tener un límite de peso a soportar, a diferencia del k-PMP.

2.2. b)

Dado un grafo un G k-coloreable, si generamos una particion $G = v_1, v_2...v_k$, en donde v_j contiene a los nodos coloreados con j, podemos asegurar que el peso de v_j es cero, pues el conjunto de aristas del subgrafo es vacío.

Entonces si para el grafo G de la instancia del problema, podemos encontrar un k-coloreo, entonces la solución óptima es la generada a partir del coloreo, pues ésto nos asegura que la suma de pesos de la partición será 0.

El problema es que quizás no existe un k-coloreo del grafo G.

2.3. c)

Primer ejemplo:

En un colegio secundario los docentes se encuntran frecuentemente con grupos de alumnos que generan disturbios en clase, pues como todos sabemos algunos alumnos se potencian entre sí.

Para solucionar ésto, las autoridades del colegio le pidieron al departamento de informática que realizen un algoritmo que reparta a los alumnos en los distintos cursos, minizando el nivel de *conflicto*, considerando que dos alumnos tienen un coeficiente de maldad asociado.

¿Por qué es un k-PMP?
Este ejemplo podemos relacionarlo con el problema de k-PMP, ya que si hacemos una analogia entre los nodos y los alumnos, uniendolos con aristas cuyo peso representa el coeficiente de maldad, podríamos resolver el problema ubicandolos en particiones distintas (en este caso cursos) con el objetivo de reducir los conflictos por aula.

Ejemplo 2: Supongamos que tenemos una cantidad de materias, las cuales debemos cursar para finalizar la carrera. Sabemos que cursar dos materias en el mismo cuatrimestre tiene una cierta dificultad, por lo tanto se pide distribuir las materias en k cuatrimestres, con el objetivo de minimizar la dificultad total de la carrera

¿Por qué es un k-PMP?
Si tomamos a las materias como nodos y definimos el peso de las aristas como la dificultad entre materias, estaríamos frente al problema de k-PMP, pues deberíamos distribuir las materias en los k cuatrimestres minimizando la dificultad total de la carrera.

Ejemplo 3: Tenemos que organizar una fiesta de 15 a la que asistiran n personas, las cuales sabemos que entre cada par de personas hay un coeficiente de enemistad. Queremos sentar a las personas en k mesas (mesas con tamaño a lo sumo n), minimizando la enemistad total de la mesa.

¿Por qué es un k-PMP?
En este ejemplo podemos tomar a los nodos como si fuesen las personas y el peso de las aristas sería el coeficiente de enemistad entre las personas. También podriamos resolver el problema como si fuese el de k-PMP, ya que debemos ubicar a las personas en k mesas distintas, minimizando el coeficiente de enemistad por mesa, generando asi el menor conflicto posible.

3. Punto 2

3.1. a)

Para elaborar el algoritmo exacto, utilizamos la técnica algorítmica de *backtracking*, pues debemos explorar todas las posibles soluciones para compararlas entre sí, ya que ésta es la única forma de asegurarse que la solución obtenida sea la óptima. El algoritmo consiste en ir armando soluciones de forma recursiva, hasta que no queden más posibles soluciones para armar.

Para ésto, se comienza con k-conjuntos[1] vacíos, que representan la solución, y con una lista de candidatos. De esta última, se elige un elemento y se lo agrega a cada uno de los conjuntos, llamando recursivamente con cada instancia.

A continuación se presenta el pseudocódigo del algoritmo exacto.

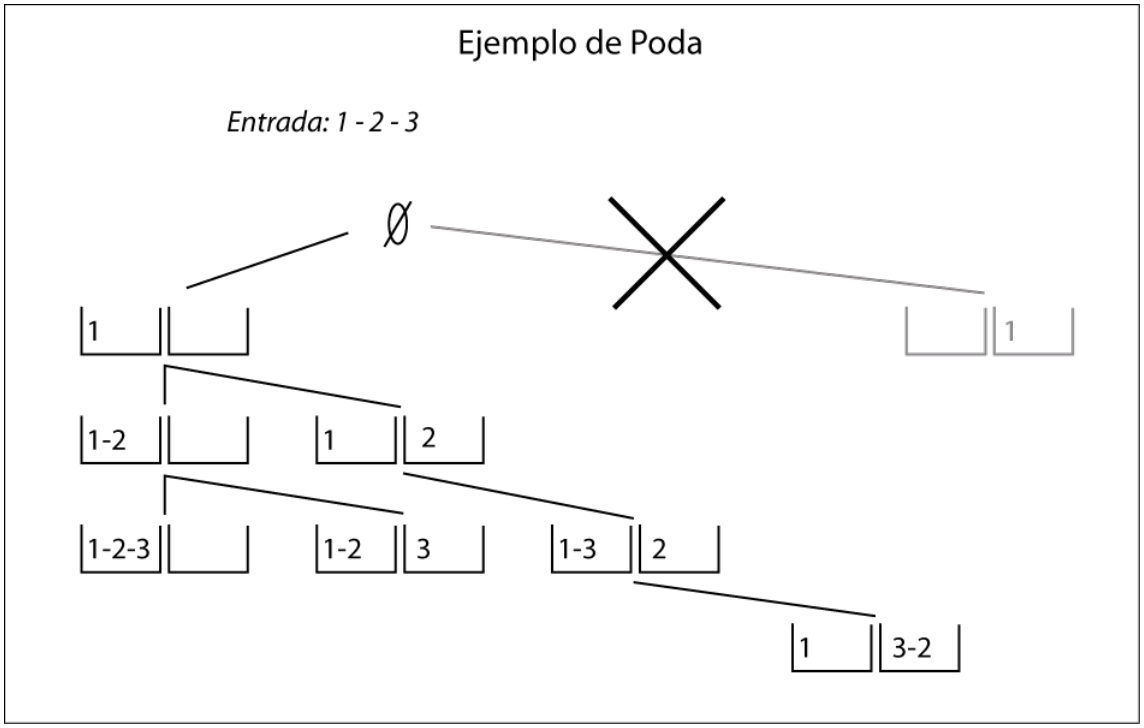
```
PARTICION(in/o arreglo(conj(nodo)): res, in/o arreglo(conj(nodo)): particion, in conj(nodo): candidatos)
 1  if candidatos es vacia
 2     then res ← particion
 3     else  nodo elegido ← elegir uno y sacarlo de candidatos
 4           for i = 0 hasta tamanio de particion
 5             agregar a elegido en particion[i]
 6             if al agregarlo sigue siendo subsolucion
 7                then particion(res, particion, candidatos)
 8             endif
 9             sacar a elegido de particion[i]
10           endfor
11  endif
```

Uno de los problemas de este algoritmo es que repite soluciones, aumentando considerablemente el tiempo de resolución.

Para mitigar ésto, elaboramos una poda que elimina las ramas del árbol de recursión que llevan a hojas (soluciones) ya revisadas. A continuación se presenta el pseudocódigo con dicha poda.

```
PARTICION(in/o arreglo(conj(nodo)): res, in/o arreglo(conj(nodo)): particion, in conj(nodo): candidatos)
 1  if candidatos es vacia
 2     then res ← particion
 3     else  nodo elegido ← elegir uno y sacarlo de candidatos
 4           for i = 0 hasta tamanio de particion
 5             agregar a elegido en particion[i]
 6             if al agregarlo sigue siendo subsolucion
 7                then particion(res, particion, candidatos)
 8             endif
 9             sacar a elegido de particion[i]
10             if si al sacar al elegido el conjunto queda vacio
11                then detener el ciclo
12             endif
13           endfor
14  endif
```

[1] El k repsenta la cantidad máxima de conjuntos de la partición.



En el gráfico se representa brevemente cómo funciona dicha poda.
El ejemplo es con una entrada de 3 nodos y dos conjuntos.

3.2. Análisis de la complejidad.

Para analizar la complejidad del algoritmo sin la poda, plantearemos la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} k * S * T(n-1) & si\ n > 0 \\ 1 & si\ n = 0 \end{cases}$$

Siendo $T(n)$ la ecuación de recurrencia que describe el comportamiento de la complejidad del algoritmo planteado, k la cantidad de conjuntos de la partición, n la cantidad de nodos y S la función que verifica que la sub-solución es factible. Diremos que el costo de S es $O(k+n)$ por como está implementado, sin importar en qué llamada recursiva se encuentre para así simplificar el análisis.

Si desarrollamos la ecuación nos encontramos con lo siguiente:

- $T(n-1) = k * S * T(n-2)$
- $T(n-2) = k * S * T(n-3)$
- $T(n-3) = k * S * T(n-4)$
- $T(n-4) = k * S * T(n-5)$
- Y así hasta que n tome valor 0.

Luego, si reemplazamos los términos.

- $T(n) = k * S * (k * S * (k * S * T(n-3)))$

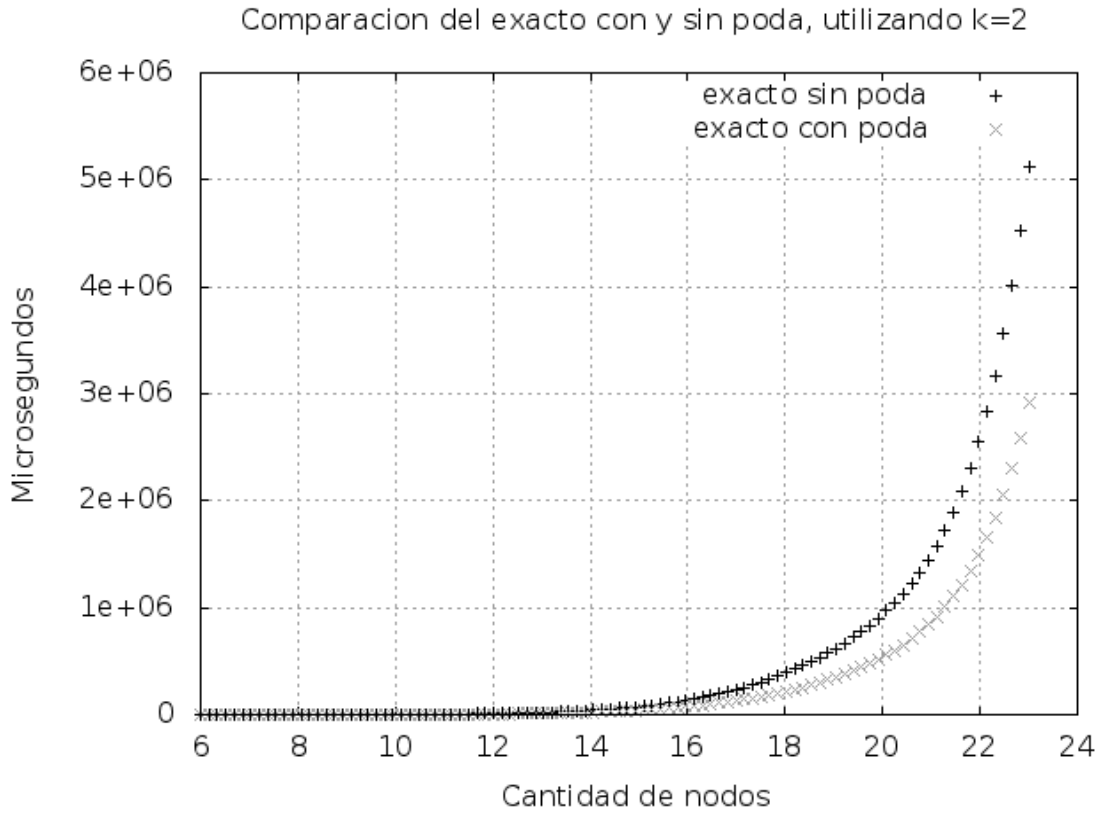
Podemos deducir por el comportamiento de la función recursiva que su formula cerrada es $T(n) = (k * S)^n$. Por como está implementada la función S, diremos que su complejidad es $O(n+k)$. Luego, la complejidad del algoritmo es $O((k * (n+k))^n)$.

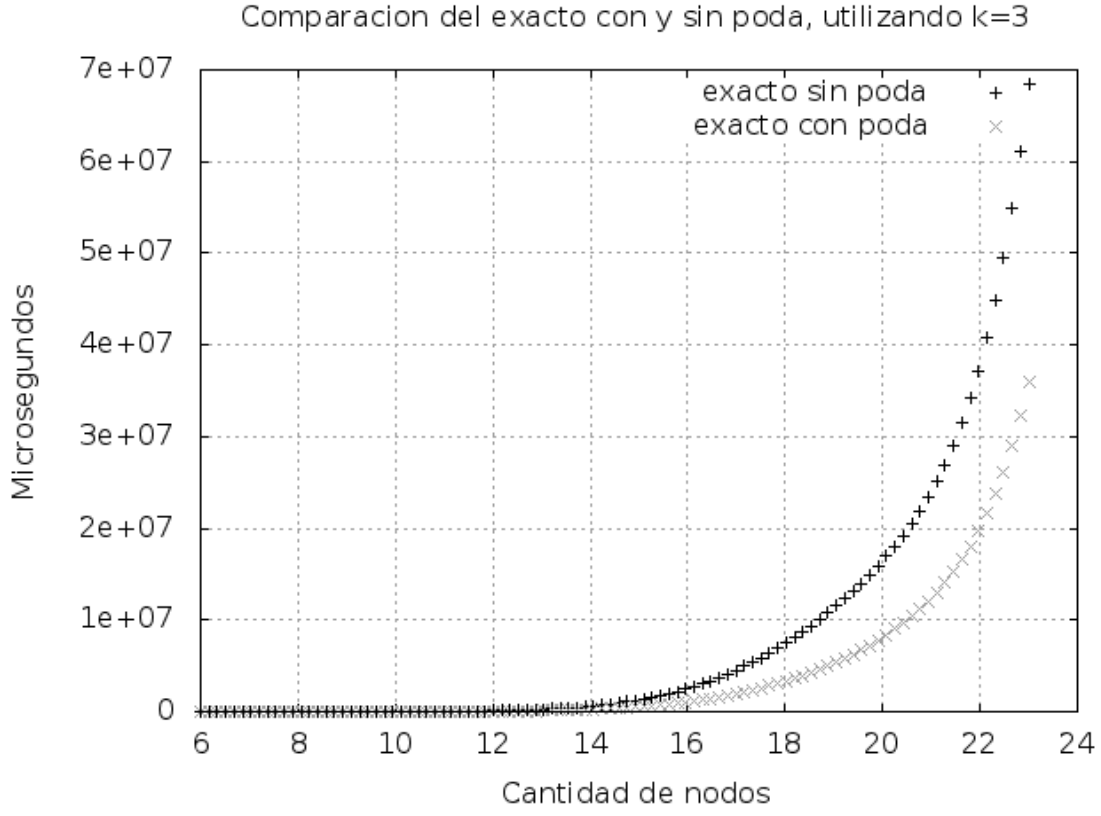
No haremos un análisis de la complejidad teórica del algoritmo con la poda, pero sí mostraremos en la experimentación cómo se reduce el tiempo de ejecución al aplicar la misma.

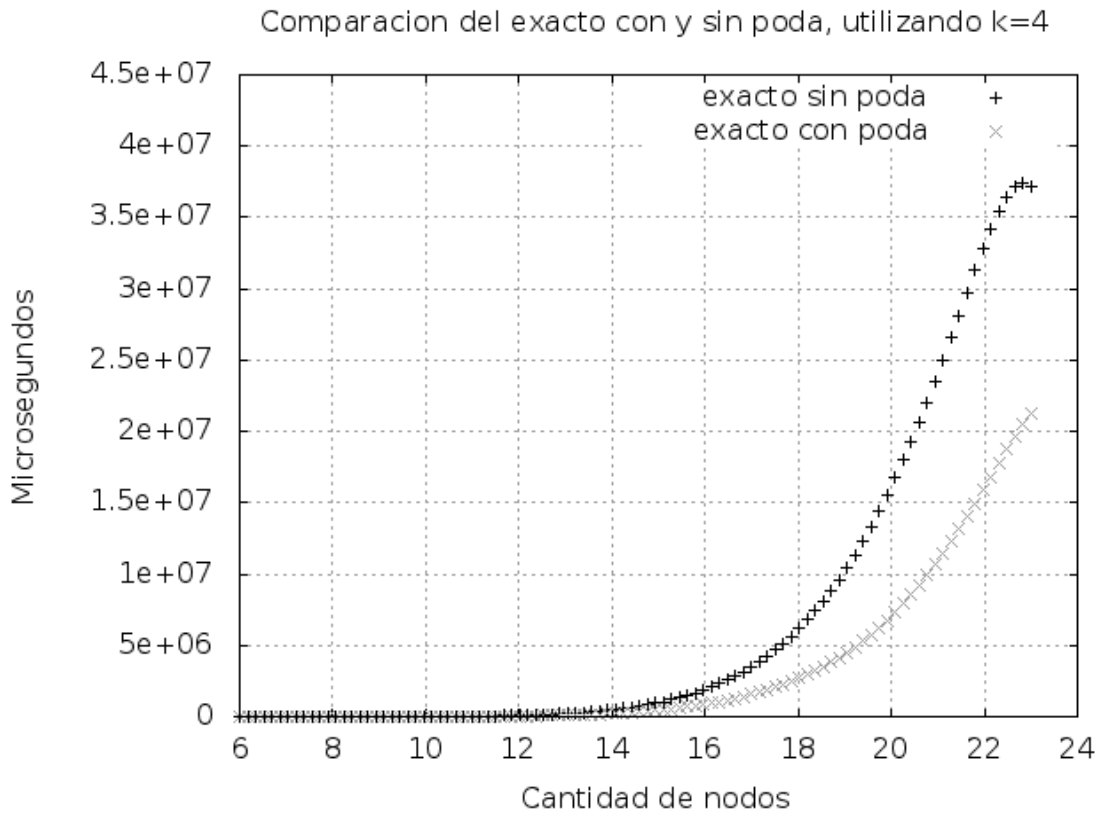
3.3. Experimentación del tiempo de ejecución.

Para corroborar el análisis de complejidad, mediremos los tiempos de ejecución del algoritmo exacto *con* y *sin* poda. El primero debería tardar más tiempo que el segundo, pues explora soluciones repetidas.

Tomaremos n desde 6 y lo incrementaremos hasta 23 (pues dada la naturaleza exponencial del algoritmo, la entrada no puede ser demasiado grande), para cada valor de n tomaremos $k = 2, 3, 4$. Para minizar el impacto de los *outliers* se ejecutará cada instancia 5 veces y nos quedaremos con el mínimo valor.







Como se puede observar en los gráficos, en primera instancia, el algoritmo con la poda implementada muestra una reducción en los tiempos de ejecución. Por otro lado vemos como para cada uno de los distintos k, las curvas en los gráficos se asimilan a n^n, lo cual se condice con la complejidad descripta, ya que al ser k una constante $O((k * (n + k))^n)$ es $O(n^n)$.

Creemos que es pertinente aclarar que no realizamos una experimentación con un k *grande*, ni un experimento para ver que la complejidad es polinomial en k si fijamos un n porque a medida que k se acerca a n el tiempo de ejecución disminuye drásticamente, debido a que el algoritmo encuentra una solución *buena* rápidamente, lo que lo lleva a no explorar la mayoría de las ramas del árbol de recursión.

4. Punto 3

4.1. a)

La idea de la heurística golosa es simple, consiste en ubicar a los vértices del grafo dentro del conjunto de la partición de forma tal que al agregarlo, impacte lo menos posible sobre el peso acumulado del conjunto, minimizando así (localmente) la suma total de pesos.

Para lograr esto, el algoritmo recorre los nodos sin un orden específico, y los ubica en el conjunto de la partición donde impacte menos. El algoritmo terminará una vez que haya ubicado a todos los nodos.

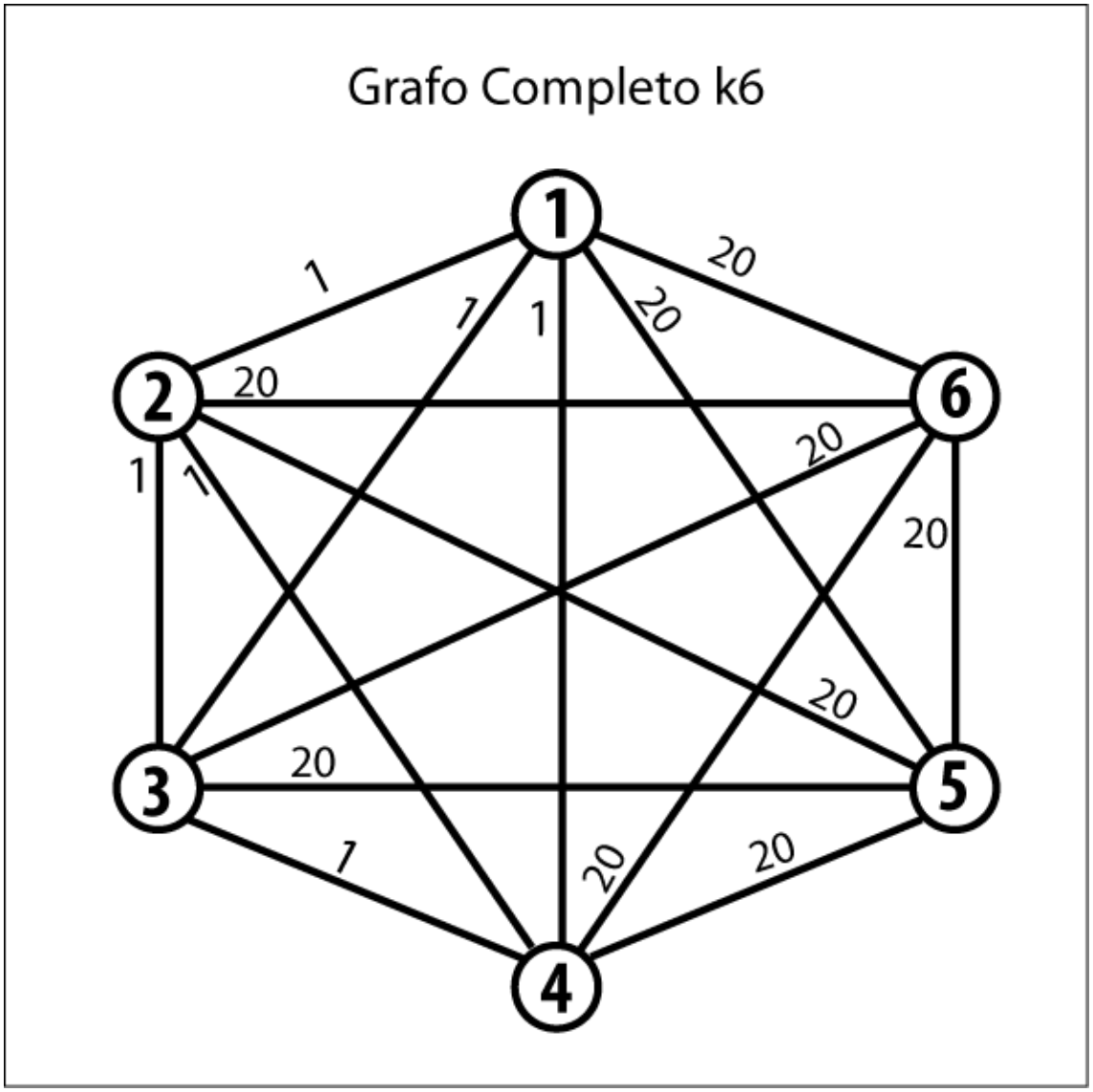
A continuación se presenta el pseudocódigo:

```
HEURISTICA_GOLOSA(Grafo (V, X) G, int k)→ Secuencia(conj(nodo))
1  Secuencia(conj(nodo)) partes ← vacia
2  Matriz(int) mz_ady ← generar_adyacencias(G)
3  while V ≠ ∅
4    nodo elegido ← elegir alguno de V
5    ubicar_en_menor(u, v, partes, mz_ady, vistos)
6    V ← V / {elegido}
7  endwhile
8  return partes
```

```
UBICAR_EN_MENOR(in nodo: u, in/o Secuencia(conj(nodo)): partes, in Matriz(int): mz_ady)
 1  int i ← 0
 2  int costo ← +∞
 3  int minimo
 4  while i < cantidad de conjuntos en partes
 5    int candidato ← calcular el costo de agregar u a partes[i]
 6    if candidato < costo
 7       then costo ← candidato
 8            minimo ← i
 9    endif
10    i++
11  endwhile
12  agregar a u en partes[minimo]
```

- Matriz(int) **generar_adyacencias**(G), crea una matriz en la que cada elemento es un entero que indica el peso de la arista entre ambos nodos. Si no hay arista entre los nodos, el valor es de -1

La presición de la heurística golosa dependerá del orden en el que se elijan a los nodos del grafo al momento de ubicarlos en un conjunto de la partición. Tengamos por ejemplo, un grafo completo de 6 nodos y una partición con 3 conjuntos.



Si el algoritmo toma los nodos de forma secuencial, primero el nodo 1, luego el 2, y así sucesivamente hasta el 6, la solución que dará será la siguiente:

- Bolsa[2] 1: nodo 1 y nodo 4.
- Bolsa 2: nodo 2 y nodo 5.
- Bolsa 3: nodo 3 y nodo 6.

Esta solución trae un peso asociado de 42. La siguiente es otra posible solución:

- Bolsa 1: nodo 1, nodo 2, nodo 3 y nodo 4.
- Bolsa 2: nodo 5.
- Bolsa 3: nodo 6.

Ésta última es la solución óptima, con un peso de 6.

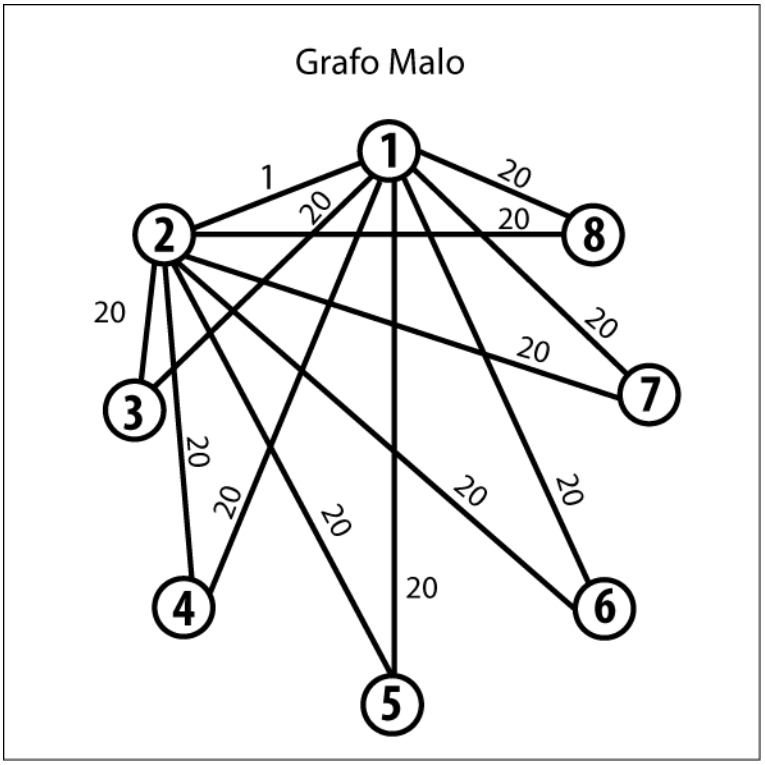
La razón por la cual el algoritmo no encuentra la solución óptima, dada la elección de nodos descripta, es porque ubica en bolsas distintas los primeros k nodos (3), siempre y cuando los primeros k nodos tengan una arista entre sí. Ésto se debe a que la heurística busca la bolsa donde agregar el nodo impacte en menor medida, con lo cual, aunque el costo de poner un nodo en una bolsa sea tan bajo como 1, si hay una bolsa vacía, lo pondrá allí, pues dicho costo es de 0.

Entonces, en las primeras k iteraciones, se distribuyen los primeros k nodos en bolsas distintas, dejando todas con un elemento. Con lo cual, a la hora de poner los nodos más *pesados* ya no existe ninguna bolsa libre, y en consecuencia, ponerlo en cualquier bolsa traerá un costo de al menos 20.

[2]Nos referimos a bolsa como un conjunto de la partición.

Es por ésto que desarrollamos un algoritmo complementario que ordena la secuencia de nodos, ubicando primeros en la lista a los nodos que tienen un mayor peso asociado. De ésta forma, casos como el anterior se solucionan, pues a la hora de elegir un nodo para ubicar en la partición, se elige al de mayor peso.

Aunque ésto soluciona el problema descripto anteriormente, al ser una heurística, sigue habiendo casos en los cuales el algoritmo no llega a la solución óptima. Veamos el siguiente ejemplo, supongamos que tenemos el siguiente grafo.



Ahora bien, aquí, si ordenamos a los nodos por su *peso asociado*[3], tendríamos la siguiente secuencia:

- 1,2,3,4,5,6,7,8

Con lo cual, el algoritmo primero eligiría al nodo 1 y lo ubicaría en la bolsa 1, luego tomaría el nodo 2 y lo ubicaría en la bolsa 2, luego los nodos $3, 4$ y 5 lo ubicaría en la bolsa 1 y los nodos $6, 7$ y 8 en la bolsa 2, obteniendo un peso de 60 en cada bolsa, lo que conlleva a un peso final de 120 para la partición.

Ahora si en vez de separar el nodo 1 y 2, estos estuvieran juntos podríamos obtener una solución (que es la óptima) en la cual el peso de la partición sería de 1, pues los nodos estarían distribuidos de la siguiente forma:

- Bolsa 1: nodo 1 y nodo 2.
- Bolsa 2: nodo 3, nodo 4, nodo 5, nodo 6, nodo 7 y nodo 8.

El problema está en que como los nodos más *pesados* están relacionados entre sí por una arista de muy bajo costo, el algoritmo los ubica en dos bolsas distintas, sentenciándose a que cualquier nodo que ubique a continuación aumente el peso acumulado de la bolsa que elija.

Lo mismo pasará con cualquier otro grafo de características similares. Es decir un grafo donde:

- Hay **solo** dos tipos de nodos, los *centrales* y los *satélites*.
- Cada nodo *central* tiene una arista que lo conecta con todos los demás *centrales*, cuyo peso es mínimo.
- Cada nodo *satélite* está **solamente** conectado a los nodos *centrales*, y el peso de estas aristas es estrictamente mayor al de las aristas que conectan a los *centrales* entre ellos.

[3] Llamamos *peso asociado* de un nodo a la suma de los pesos de las aristas que inciden sobre él.

Entonces, dada esta descripción, es fácil ver que la solución óptima siempre será mantener separados a los *satélites* de los *centrales*, aún si eso significa tener juntos a los *centrales*, pues las aristas que los conectan son de peso mínimo. Este tipo de grafos será una instancia mala siempre y cuando la cantidad de conjuntos de la partición sea mayor estricto a la cantidad de *centrales*, pues de otra forma, el algoritmo pondrá cada nodo *central* en una bolsa diferente, y a todos los *satélites* en una bolsa sin *centrales*, consiguiendo un peso total de valor 0.

Una posible instancia de ésta familia de grafos es el ejemplo mencionado anteriormente.

Éste es *un* peor caso de la heurística golosa desarrollada que encontramos, cuya solución propuesta para dicho caso no será tan mala como la peor partición posible, pero que está muy alejada de ser la óptima.
Sin embargo, no son todas malas noticias para la heurística golosa. Si el grafo a evaluar es un árbol o un bipartito, el algoritmo encuentra la solución óptima. Antes que nada, como vimos durante la cursada, tanto los árboles como los grafos bipartitos son 2-coloreables, y esto significa (como explicamos en la primer sección) que el peso total de la partición será 0.

Dicho ésto, no es difícil ver que la solución óptima para todos estos grafos es la que lleva el peso a 0, siempre y cuando la cantidad de conjuntos de la partición sea al menos 2 (si fuese 0 o 1 el problema no tendría sentido).

Entonces, sólo hay que ver que efectivamente la heurística separe bien a los nodos, y ésto sucede pues para cada nodo (sin importar cuál tome primero) el algoritmo lo pone en la **primer** bolsa donde impacte menos. Con lo cual, el primer vértice que tome, irá a la primer bolsa (pues todas las bolsas están vacías), para el segundo vértice hay dos opciones, o bien tiene una arista con el nodo ya ubicado, o no. Si tiene una arista, entonces irá a la segunda bolsa, si no la tiene, irá a la primera. Para el tercer nodo hay tres posibilidades, o bien tiene una arista con algún nodo de la bolsa 1, o tiene una arista con algún nodo de la bolsa 2, o no tiene arista con ningún nodo de ambas bolsas. En el primer y tercer caso, el vértice será ubicado en la bolsa 1, mientras que en el caso restante irá a la bolsa 2.

Éste procedimiento será repetido hasta que se agoten los nodos. Lo importante aquí es que para cualquier nodo que elija, si existe una arista con algún nodo de la bolsa 1, no existirá arista con ningún nodo de la otra bolsa.

No es nuestra intención demostrar con ésto que la herística golosa encuentra el óptimo, si no dar una breve explicación de por qué lo hace.

Dicho ésto, realizamos experimentos para ver la veracidad de la afirmación, pero no los incluimos porque son redundantes, al ser el peso siempre 0 para cualquier grafo de entrada que sea árbol o bipartito.

4.2. Análisis de la complejidad.

Para el análisis de la complejidad diremos que k representa la cantidad de conjuntos de la partición, y n la cantidad de nodos del grafo.

Antes de analizar la complejidad de la heurística, veremos cuál es el costo computacional de **ubicar_en_menor**. El algoritmo consta de un ciclo que itera k veces, calculando el costo de agregar el nodo a un conjunto. Ésta ultima operación es lineal en la cantidad de elementos del conjunto, pues para cada uno de ellos, simplemente hay que buscar su peso (si lo hay) en la matriz de adyacencias, lo cual es $O(1)$, y luego sumarlo a un acumulador, operación cuyo costo es también constante.

En un análisis superficial, parecería que el costo de **ubicar_en_menor** es $O(k * n)$, pues para los k conjuntos calculo el peso asociado del nodo con los elementos de dicho conjunto, y su tamaño en el peor caso es n. Sin embargo, ésto no es así. Si el tamaño de un conjunto es n, entonces los demás conjuntos estarán vacíos. Lo que en realidad hay que ver es la sumatoria de los costos de todos los calculos de los pesos asociados de cada conjunto, que es la sumatoria de todos los elementos de cada conjunto, que en el peor caso son n.

Por ésto, la complejidad de **ubicar_en_menor** es $O(n + k)$, pues sí o sí se realizan las k iteraciones, y sí o sí se revisan todos los elementos de la partición.

Ahora que ya sabemos la complejidad de la función auxiliar, analizemos la heurística golosa.

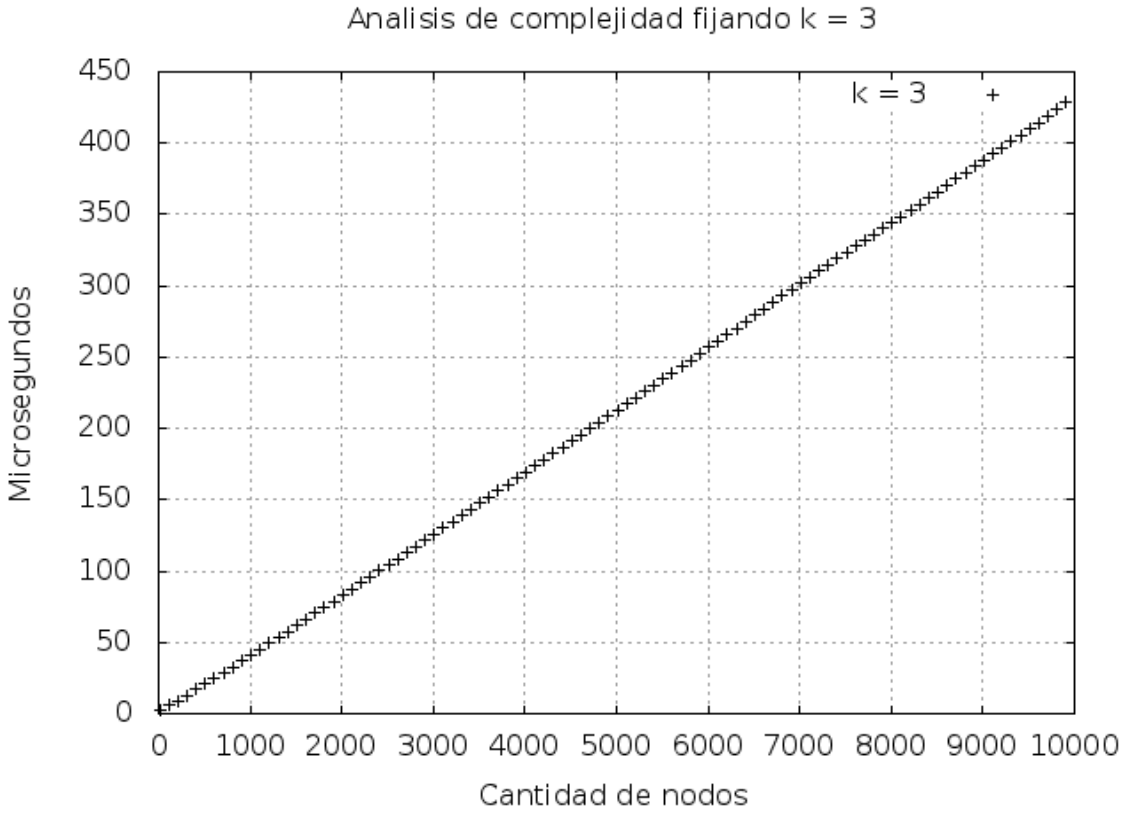
El algoritmo primero genera la matriz de adyacencias, cuyo costo es $O(n^2)$. Luego, hay un ciclo que itera n veces, llamando en cada iteración a **ubicar_en_menor**. Luego, la complejidad de la heurística es $O(n * (n + k)) + O(n^2)$, que es $O(n^2)$ si $n > k$ o $O(n * k)$ sino.

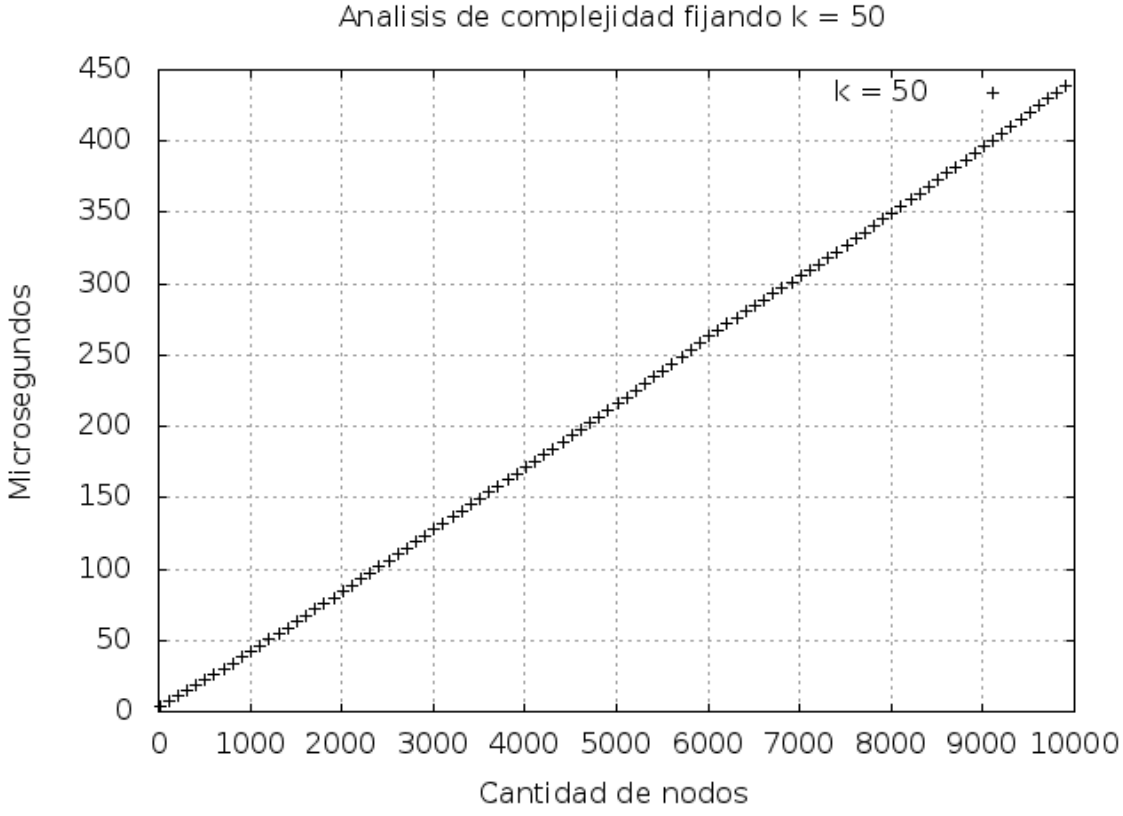
4.3. Experimentación.

En ésta sección nos interesará verificar que los tiempos de ejecución de la heurística coicidan con el análisis teórico de la complejidad. Para ésto, vamos a ver dos cosas:

1. Que, al fijar k, el algoritmo es cuadrático en n, sin importar la relación entre n y k.
2. Que, al fijar n, el algoritmo es lineal en k.

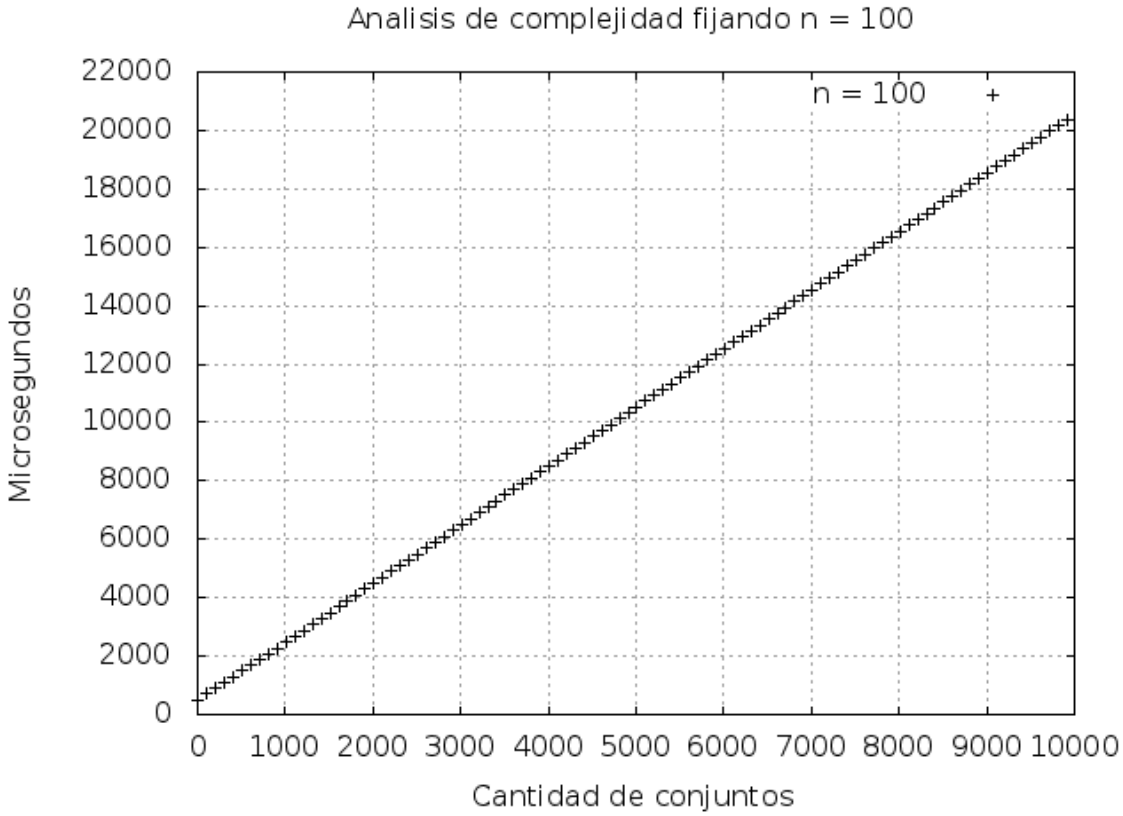
Para el primer ítem, fijamos dos k distintos, $k = 3$ y 50, y para cada valor variamos n desde 100 hasta 10000, incrementando de a 100, y corriendo 15 iteraciones de cada instancia quedándonos con el mínimo de los tiempos. La semilla utilizada fue 13.

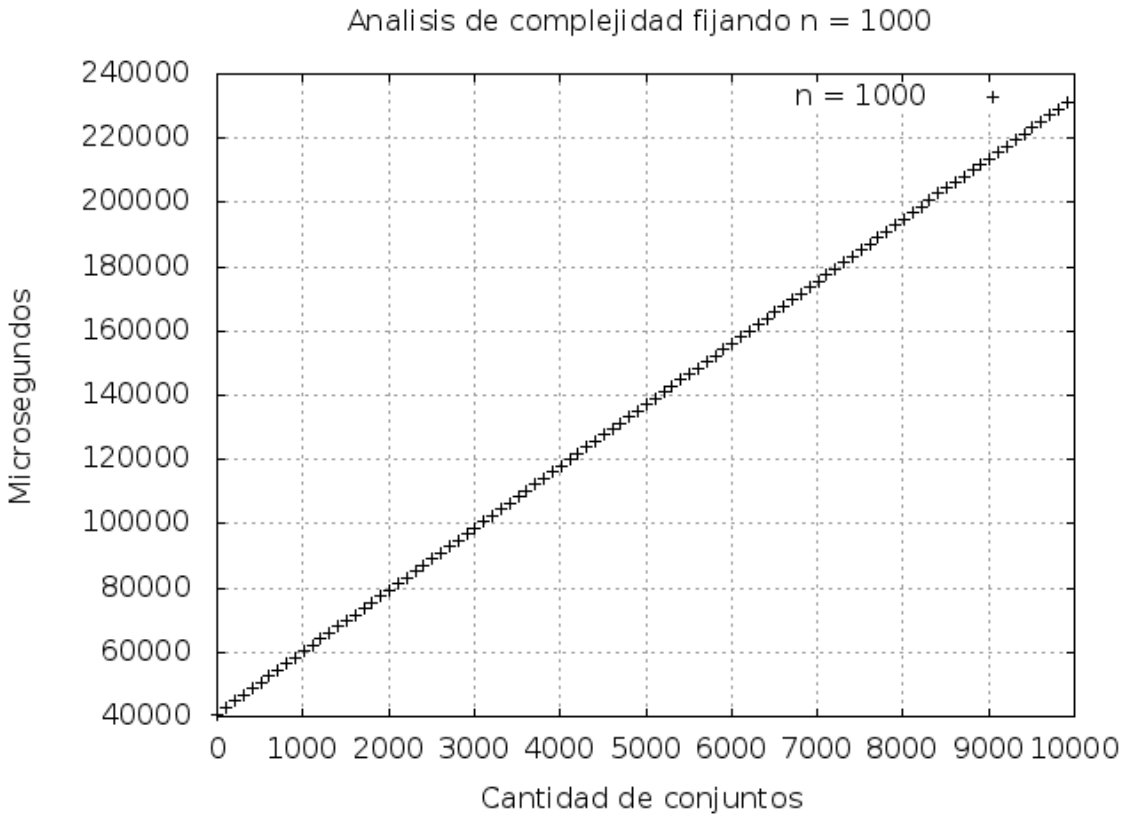




Para observar que la complejidad calculada sea cuadrática en n, decidimos dividir a los valores obtenidos en la experimentacion por la función f(x) = x. Esto nos permitió verificar gráficamente que la complejidad termina siendo lineal, ya que si realizamos la operación n^2/n, esto terminara dando una funcion lineal tal como se observa en los gráficos obtenidos.

Para ver que sea lineal en k, fijamos dos n distintos, $n = 100, 1000$. Para cada valor de n, variamos el k desde 10 hasta 1000, incrementándolo de a 10. Cada instancia se ejecuta 15 veces, quedándonos con el mínimo de los tiempos. La semilla utilizada fue 13.





Tal como fue explicado previamente, y como se observa en los gráficos la complejidad es lineal en k

5. Punto 4

5.1. a)

Nuestra heurística de búsqueda local genera alrededor de una solución una vecindad de soluciones y va tomando el mejor de los resultados para generar nuevas vecindades.
Cuando llega a un óptimo local, es decir, cuando en la vecindad generada no se consigue alguna solución de menor costo, la heurística finaliza su ejecución.

Consideraremos que una solución S es vecina a una solución H si S se puede obtener desde H al mover un vértice de algún conjunto de la parición a otro.

Para lograr generar todas las vecindades, recorremos todos los nodos, que están distribuidos en los distintos conjuntos de la partición, y por cada nodo obtenemos todas las soluciones vecinas que se generan a partir de mover dicho nodo.

Éste procedimiento se realiza de forma repetida hasta encontrar un mínimo local, es decir dada una solución no se encuentra una solución vecina menor en toda la vecindad.

A ésta vecindad la llamaremos 1-opt, pues las vecindades se consiguien al mover sólo 1 nodo de un conjunto a otro.

Pseudocódigo:

```
BUSQUEDALOCAL_1OPT(in inicial: particion, in mz_ady: matriz(int), in k: int)→ particion
 1  particion actual ← inicial
 2  while No encontré el mínimo local
 3    while Haya un conjunto i por recorrer en actual
 4      if El costo de i ≠ 0
 5        then while Haya un nodo v a evaluar en i
 6                 while Haya un conjunto j por recorrer, tal que j ≠ i
 7                   Sea H la solucion vecina dada de mover v a j
 8                   if  H es mejor solucion que la actual
 9                     then Recordar H como la mejor solucion vecina
10                   endif
11                 endwhile
12               endwhile
13      endif
14    endwhile
15    if H es mejor solucion que actual
16      then actual ← H
17      else  devolver actual
18    endif
19  endwhile
20
```

A continuación pasaremos a describir las ideas desarrolladas para una Búsqueda Local con distinta Vecindad a la anteriormente explicada.
Esta búsqueda local mantiene la misma idea que la anterior, de generar alrededor de una solución una vecindad de soluciones y tomar el mejor de los resultados para generar nuevas vecindades, y cuando llega a un óptimo local,la heurística finaliza su ejecución.

La diferencia radica en la vecindad. En esta búsqueda local consideraremos que una solución S es vecina a una solución H si S se puede obtener desde H al mover un par vértices de algún conjunto de la parición a otro.

A esta vecindad la llamaremos 2-opt.

Pseudocódigo:

```
BUSQUEDALOCAL_2OPT(in/out res: particion, in mz_ady: matriz(int), in k: int)
 1  particion actual ← res
 2  while No encontré el mínimo local
 3    while Haya un conjunto i por recorrer en actual
 4      if El costo de i ≠ 0
 5        then while Haya un nodo v a evaluar en i
 6                 while Haya un nodo u a evaluar en i, tal que u ≠ v
 7                   while Haya un conjunto j por recorrer, tal que j ≠ i
 8                     Sea H la solucion vecina dada de mover los nodos v, u a j
 9                     if  H es mejor solucion que la actual
10                       then Recordar H como la mejor solucion vecina
11                     endif
12                   endwhile
13                 endwhile
14               endwhile
15      endif
16    endwhile
17    if H es mejor solucion que actual
18      then actual ← H
19      else  devolver actual
20    endif
21  endwhile
22
```

5.1.1. Análisis de la complejidad

Para el análisis de la complejidad diremos que k representa la cantidad de conjuntos de la particón, y n la cantidad de nodos del grafo.

Primero analizaremos la complejidad de una iteración la heurística búsqueda local 1-opt.

El pseudocódigo comienza con dos ciclos anidados, en la línea 3 y 5, donde el primero parecería iterar k y el segundo n, dándonos a pensar que la cantidad de iteraciones totales es de $k \times n$. Sin embargo, analizándolo con mayor atención, se ve que la cantidad de iteraciones realizadas es la suma de los nodos de cada conjunto, lo que es n.

Ya establecido que los dos primeros ciclos anidados iteran n veces veamos qué sucede adentro.

El ciclo de la línea 6 realiza k iteraciones, pues recorre todos los conjuntos. Luego, se crea una solución vecina, cuyo costo es el de copiar la partición, lo cual es $O(n)$. Además, se recalcula el peso del conjunto a donde se movió el nodo elegido, y del conjunto origen de dicho nodo. La operación realizada tiene un costo lineal en la cantidad de nodos del conjunto afectado, como sea realiza dos veces dicha operación, una para el conjunto origen, y otra para el conjunto destino, la complejidad es $O(2 \times n)$ lo que es $O(n)$.

Luego, se verifica que la solución vecina creada sea mejor que la ya existente, diremos que ésta operación tiene un costo constante, pues en la implementación, el costo de la solución ya existente es almacenado en una variable, y el costo de la nueva solución es el resultado de dicha variable sumándole y restándole los costos asociados de mover el nodo a otra partición, éstas dos operaciones son las descriptas en el párrafo anterior. És por ésto que verificar si H es mejor solución que la *actual* es de costo constante.

Por último, en la línea 9 se copia la H como la solución actual, ésto es $O(n)$.

Ya calculada la complejidad de las operaciones realizadas dentro del ciclo, podemos decir que la complejidad de una iteración de la heurística búsqueda local 1-opt es $O(k \times n^2)$.

Analizaremos ahora la complejidad para 2-opt.

Como los pseudocódigos de ambas heurísticas son muy similares no ahondaremos en detalle sobre las operaciones salvo las que no hayan sido descriptas anteriormente.

La diferencia radica en que, en el algoritmo anterior sólo se movía de a un nodo, lo que generaba $n \times (k-1)$ iteraciones. Ahora, como hay que generar todos los pares de nodos por cada conjunto, para luego moverlos a los $k-1$ conjuntos restantes, se generan $(k-1) \times (n \times (n-1))$ iteraciones, manteniendo el costo de las operaciones internas a los ciclos.

Entonces, la complejidad de la herística 2-opt es $O((k-1) \times (n \times (n-1)) \times n)$, lo que es $O(k \times n^2 \times n)$, que es lo mismo que $O(k \times n^3)$.

5.1.2. Experimentación

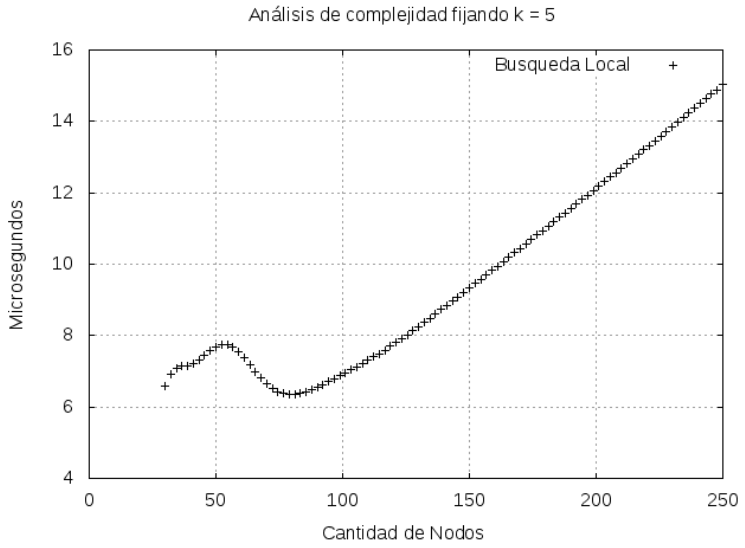
En esta sección vamos a verificar que el análsis de las complejidades de la Búsqueda Local 1-opt y Búsqueda local 2-opt sea coherente al medir sus tiempos de ejecución. Para ésto, realizaremos dos experimentos por cada vecindad:

1. Búsqueda Local 1-opt.
   - Fijar k: El algoritmo es cuadrático en n.
   - Fijar n: El algoritmo es lineal k.
2. Búsqueda Local 2-opt
   - Fijar k: El algoritmo es cúbico en n.
   - Fijar n: El algoritmo es lineal k.

Búsquda Local 1-opt

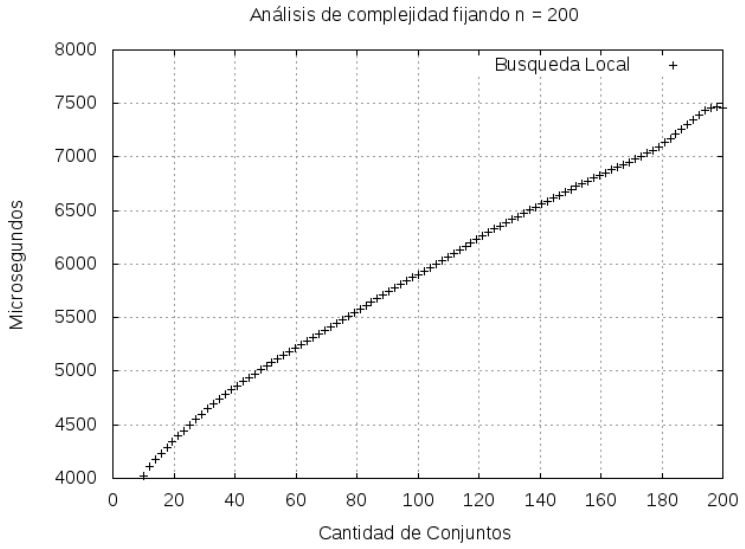
Para el primer experimento fijamos $k = 5$, y se varió el n desde 30 a 250, incrementando de a 5 nodos y corriendo 15 iteraciones de cada instancia, quedándonos con el mínimo de los tiempos. La semilla utilizada fue 13.

Aclaración: para graficar la función de este experimento, dividimos los valores por la función f(n)=n. Esto generó que la función sea de tipo lineal, ya que como se explicó anteriormente nuestro algoritmo es de orden n^2.



Podemos observar que los resultados son los esperados. En el gráfico observamos que la función obtenida es lineal por lo que concluímos que, ya que dividimos por f(n)=n, la complejidad del algoritmo es cuadrática.

Para el segundo experimento fijamos $n = 250$, y se varió el k desde 10 a 200, incrementando de a 5 conjuntos y corriendo 15 iteraciones de cada instancia, quedándonos con el mínimo de los tiempos. La semilla utilizada fue 13.

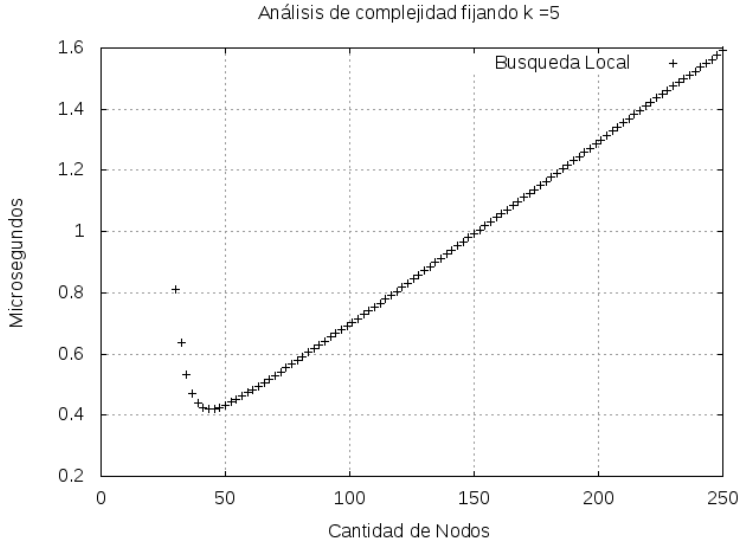


Podemos observar que los resultados son los esperados. En el gráfico observamos que la complejidad del algoritmo es lineal.

Búsquda Local 2-opt

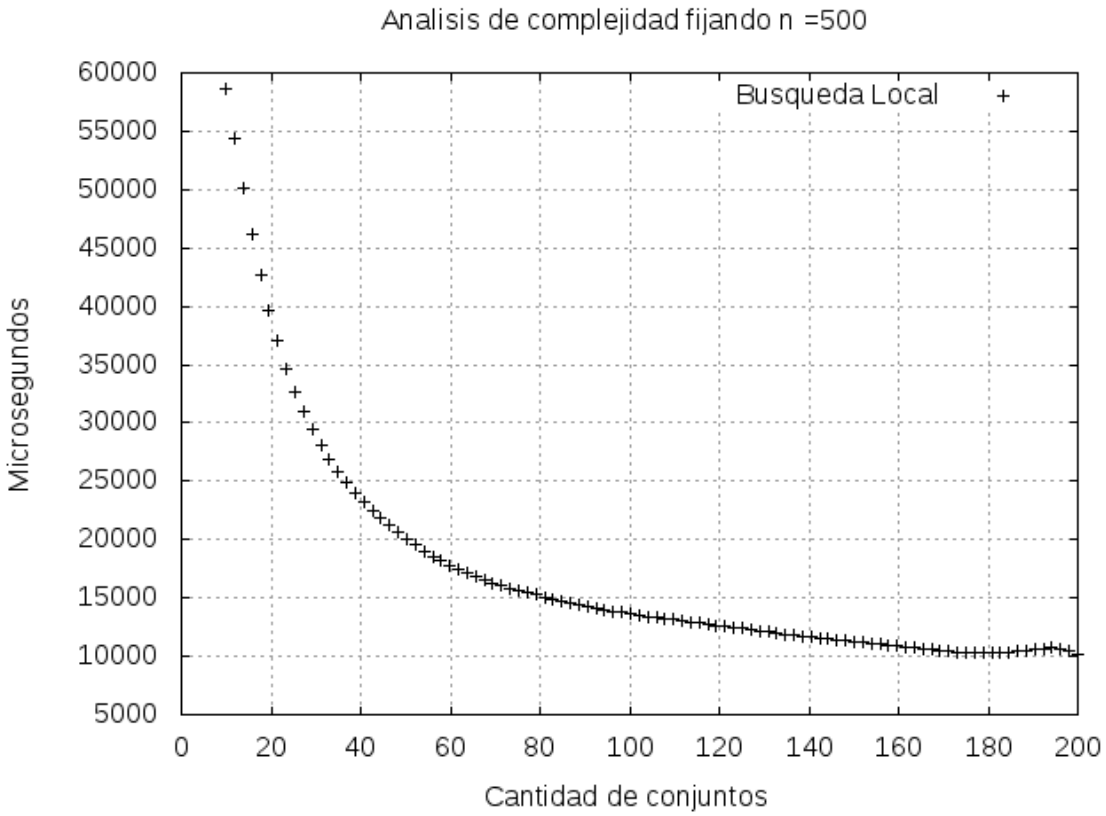
Para el primer experimento fijamos $k = 5$, y se varió el n desde 30 a 250, incrementando de a 5 nodos y corriendo 15 iteraciones de cada instancia, quedándonos con el mínimo de los tiempos. La semilla utilizada fue 13.

Aclaración: para graficar la función de este experimento, dividimos los valores por la función f(n)=n^2. Esto generó que la función sea de tipo lineal, ya que como se explicó anteriormente nuestro algoritmo es de orden n^3.



Podemos observar que los resultados son los esperados. En el gráfico observamos que la función obtenida es lineal por lo que concluímos que, ya que dividimos por f(n)=n^2, la complejidad del algoritmo es cúbica.

Para el segundo experimento fijamos $n = 250$, y se varió el k desde 10 a 200, incrementando de a 5 conjuntos y corriendo 15 iteraciones de cada instancia, quedándonos con el mínimo de los tiempos. La semilla utilizada fue 13.



El experimento realizado para verificar la complejidad teórica sobre k muestra resultados contradictorios, ya que esperabamos ver una recta y sin embargo, nos econtramos con un compartamiento que por lo menos es extraño, pues a medida que el k aumenta, los tiempos de ejecución disminuyen.

Análizando dicha anómalía, llegamos a la conclusión de que la complejidad (aún fijando el n) no depende estrictamente de k, si no de su relación con como están distribuidos los nodos en los distintos conjuntos de la partición. La vecindad planteada evalúa todas las soluciones que se desprenden de mover un par de vértices de un conjunto a todos los demás (para todos los pares de todos los conjuntos). Al aumentar el k, como los nodos están uniformemente distribuidos en los distintos conjuntos, la cantidad de pares totales para mover disminuye, con lo cual hay menos soluciones vecinas, decrementando el tiempo de ejecución.

Ahondemos un poco más formalmente en ésto. Si asumimos entonces que los nodos están uniformemente distribuidos en los conjuntos podemos decir que, siendo n la cantidad de nodos, k la cantidad de conjuntos:

- $X_i = n/k$, siendo X_i la cantidad de nodos en el conjunto i.

La cantidad de pares posibles en un conjunto de j elementos es $j \times (j-1)$, con lo cual, en un conjunto i la cantidad de pares posibles es $X_i \times (X_i - 1)$, que es lo mismo que $n/k \times (n/k) - 1$. Llamemos ahora T a la cantidad total de pares posibles en la partición.

- $T = \sum_{i=1}^{k} X_i \times (X_i - 1)$

Lo que es,

- $T = \sum_{i=1}^{k} n/k \times (n/k - 1)$, pues $X_i = n/k$

Desarrollando la sumatoria,

- $T = k \times (n/k \times (n/k - 1)) = n \times (n/k - 1)$

Ahora, como el n es una constante, pues está fijo en la experimentación, si tomamos límite cuando k tiende a n, ya que si k es mayor a n el problema deja de tener sentido pues la solución es trivial:

- $\lim_{k \to n} n \times (n/k - 1) = n \times (n/n - 1) = 0$

Con este no pretendemos decir que los tiempos de ejecución llegarán a 0, pues como sabemos ésto es imposible, si no que a medida que k se acerca a n los tiempos de ejecución disminuyen.

6. Punto 5

6.1. a)

La metaheuristica desarrollada respeta la estructura general vista de GRASP, es decir, en cada iteración del ciclo principal se obtiene una solución mediante una heurisitca golosa aleatorizada, y luego se la mejora (si es posible) con una heurística de búsqueda local.

Lo que realiza GRASP es siempre lo mismo, consigue una solución golosa (con un componente aleatorio) y la trata de mejorar con una búsqueda local. Ésto se realizada en cada iteración de un ciclo principal, recordando siempre la mejor solución encontrada. Entonces lo que hay que determinar es *cuándo* realizé suficientes búsquedas de soluciones, para ésto hay que elegir algún criterio de parada. Nosotros decidimos utilizar dos criterios de parada distintos para determinar cuántas iteraciones son suficientes:

- Detenerse al realizar k iteraciones: Este criterio consiste en iterar una cantidad prefijada de veces, donde dicho valor es un parámetro de la heurística.
- Iterar hasta no conseguir una mejora en k iteraciones : El criterio consiste detenerse solamente cuando hayan pasado k iteraciones sin conseguir una mejor solución que la mejor encontrada hasta el momento.

Con respecto a la heurística golosa aleatorizada utilizamos dos criterios para elegir la lista de candidatos (RCL) sobre los que haremos nuestra elección golosa. Ambos criterios son del tipo cantidad, ya que elegimos los β mejores dentro del conjunto original de posibles candidatos. Luego de crear esta lista, elegimos un candidato de forma aleatoria con la función *rand()* provista por la librería estándar de *c++* .

Los criterios planteados para la heuristica golosa son los siguientes:

- El primer criterio consiste en crear una lista con los β nodos más *pesados* y luego elegir aleatoriamente uno para ubicarlo en el conjunto de la partición donde menos impacte.
- El segundo criterio consiste en, dado un nodo ya elegido para ubicar en un conjunto, crear una lista con las β mejores bolsas para dicho nodo y elegir aleatoriamente el conjunto destino.

Pseudocódigo:

```
GRASP(in Grafo (V,X): G int k, int iteraciones, int β)→ Secuencia(particion)
1  Secuencia(particion) res_final ← heuristica_golosa_aleatoria(G,k,β)
2  res_final = HeuristicaBusquedaLocal(res_final,k)
3  while no se alcanza el criterio de parada
4    Secuencia(particion) res_parcial ← heuristica_golosa_aleatoria(G,k,β)
5    res_parcial ← HeuristicaBusquedaLocal(res_parcial,k)
6    if suma_total(res_parcial) ≤ suma_total(res_final)
7      then res_final ← res_parcial
8
```

- int **suma_total**(Secuencia(particion)), devuelve la suma de los pesos de todos los conjuntos de la partición.
- La heuristica golosa aleatorizada, elige los β mejores candidatos, ya sea utilizando un criterio o el otro (ambos criterios fueron previamente explicados)
- La funcion busqueda local, se comporta como la funcion previamente detallada.

6.2. Experimentación

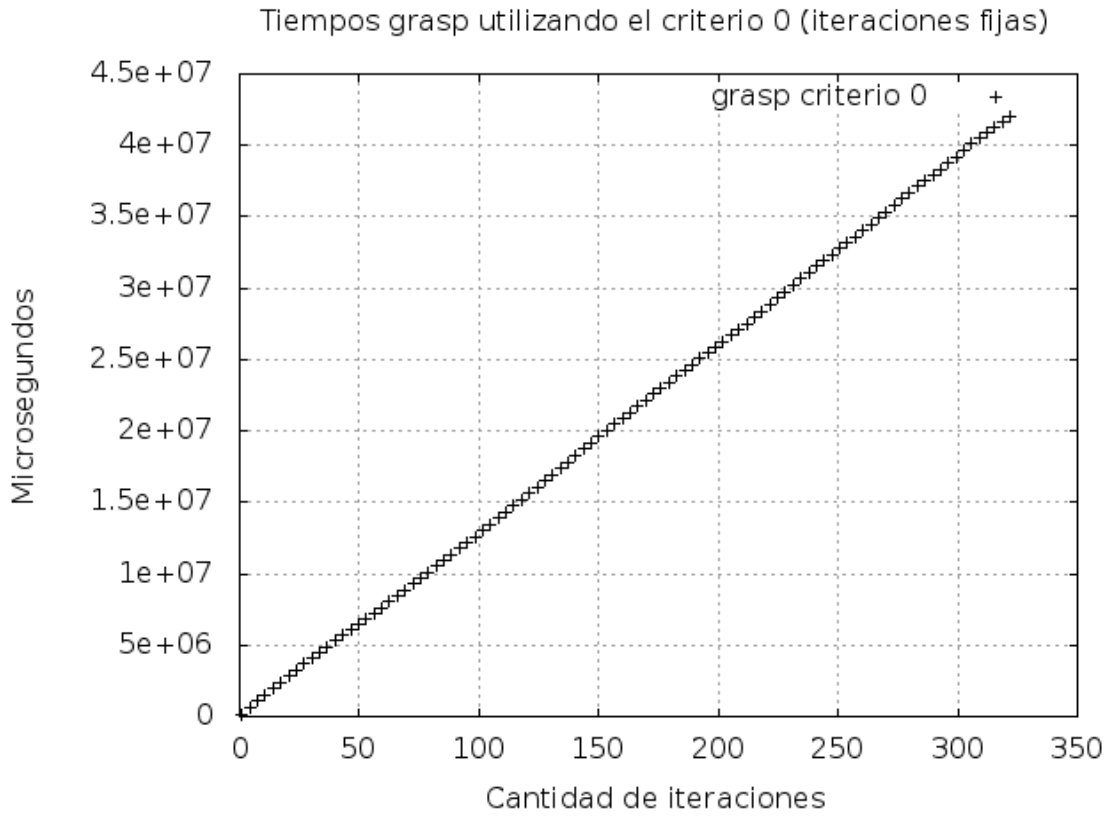
6.2.1. Complejidad

La primer experimentación de ésta heurística consiste en ver cómo afecta la cantidad de iteraciones, para cada criterio elegido, en el tiempo de ejecución.

Para cada criterio, se fijó una instancia de un grafo completo con $n = 200$ y $k = 10$, usando como semilla el número 13. La cantidad de iteraciones empieza en 1 y se incrementa hasta alcanzar las 300. El criterio de aleatoriedad utilizado es el de elegir entre los 10 nodos más pesados, aunque ésto no afecta la complejidad.

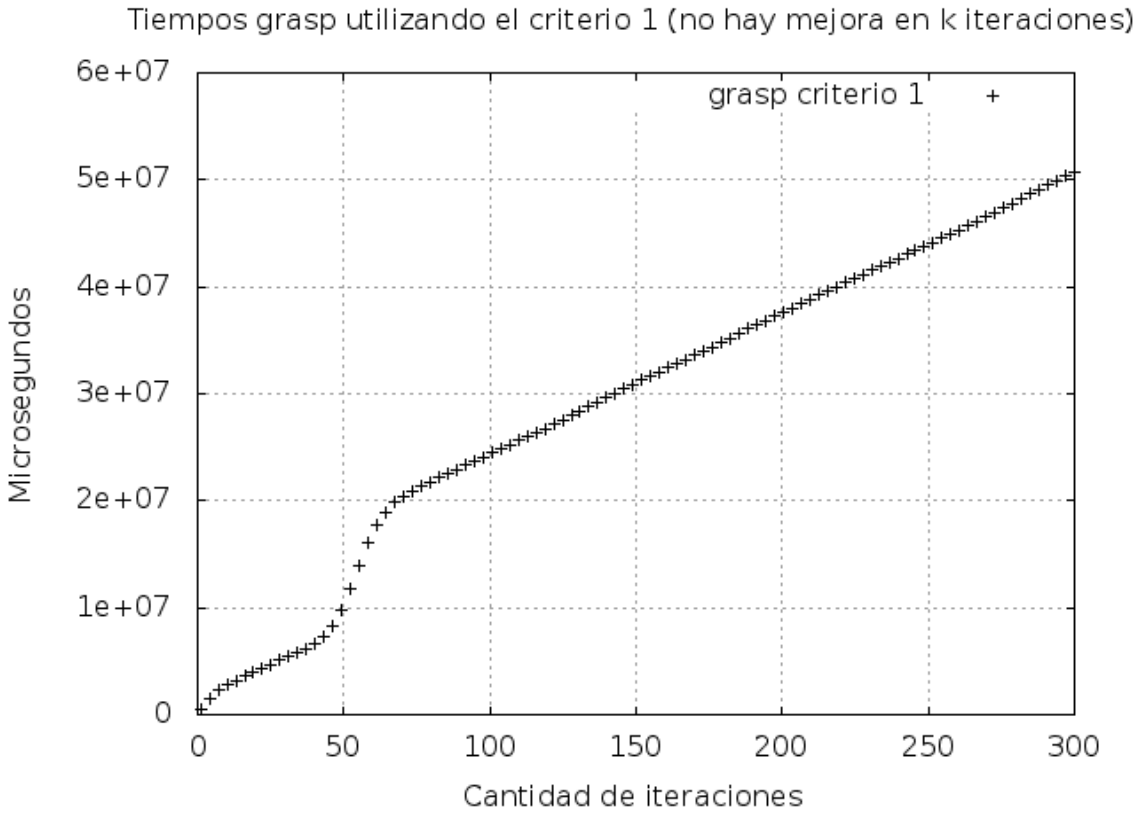
Para el criterio de k iteraciones fijas esperamos que el tiempo de ejecución aumente linealmente acorde aumenta el número de iteraciones.

El resultado del experimento es el siguiente:



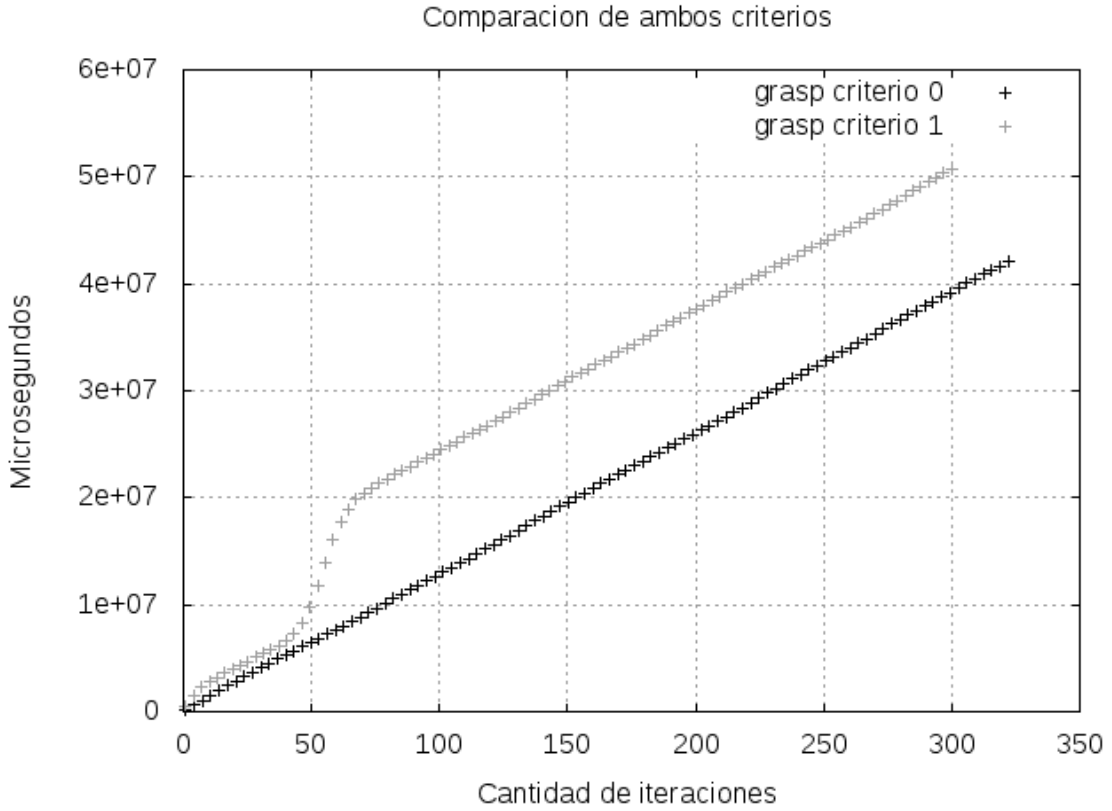
Como era de esperarse, se observa un aumento lineal del tiempo en función de la cantidad de iteraciones.

Para el criterio de *no se encuentra mejor solución en k iteraciones* esperamos que el tiempo se comporte como una función lineal con respecto a k, aunque es posible que encuentre algunas irregularidades, pues quizás a partir de algún k la herística constantemente renueve su solución al encontrar una mejor.



Al igual que el primer criterio, el tiempo de ejecución de la heurística en función al k presenta un comportamiento lineal, corroborando así el análisis previamente hecho.

Un aspecto que es bastante claro de ver, pero aún así importante de mencionar es que, el tiempo de complejidad del segundo criterio es mayor al fijar un número de iteraciones, pues si ambos toman un mismo k, el segundo criterio realizará al menos las k iteraciones, igualando así al primer criterio, pero probablemente ejecute aún más iteraciones (siempre y cuando logre mejorar la solución) superando en tiempo de ejecución al criterio de fijar las iteraciones.



6.2.2. Configuraciones

Para probar el rendimieno de GRASP en términos de los criterios de terminación y de aleatorización de la heurística golosa, tomamos las 4 posibildades variando los dos criterios de GRASP, y los dos aleatoriedades para goloso:

- **00** Tomar el criterio de grasp de k-iteraciones fijas, y aleatorizar en la elección del mejor conjunto para ubicar al nodo elegido en la heurística golosa.
- **01** Tomar el criterio de grasp de k-iteraciones fijas, y aleatorizar en la elección de los nodos más *pesados* a la hora de elegir a uno para ubicar en un conjunto.
- **10** Tomar el criterio de grasp parar si no hay mejora en k-iteraciones, y aleatorizar en la elección del mejor conjunto para ubicar al nodo elegido en la heurística golosa.
- **11** Tomar el criterio de grasp parar si no hay mejora en k-iteraciones, y aleatorizar en la elección de los nodos más *pesados* a la hora de elegir a uno para ubicar en un conjunto.

Para cada elección de criterios, variamos los valores de k (en GRASP) y el β para la heurística golosa. El β no fue fijado como constante, si no que fue variado proporcionalmente a la cantidad de nodos o conjuntos de la partición, según que critrio haya sido elegido.
Los valores para k en la configuración de k-iteraciones fijas de grasp fueron:

- 10
- 100

Para el criterio de para si no hay mejora en k-iteraciones los k utilizados fueron:

- 5
- 25

Para la heurística golosa, eligiendo de forma aleatoria los nodos se tomaron los siguientes valores:

- 1/20 de los nodos totales.
- 1/40 de los nodos totales.

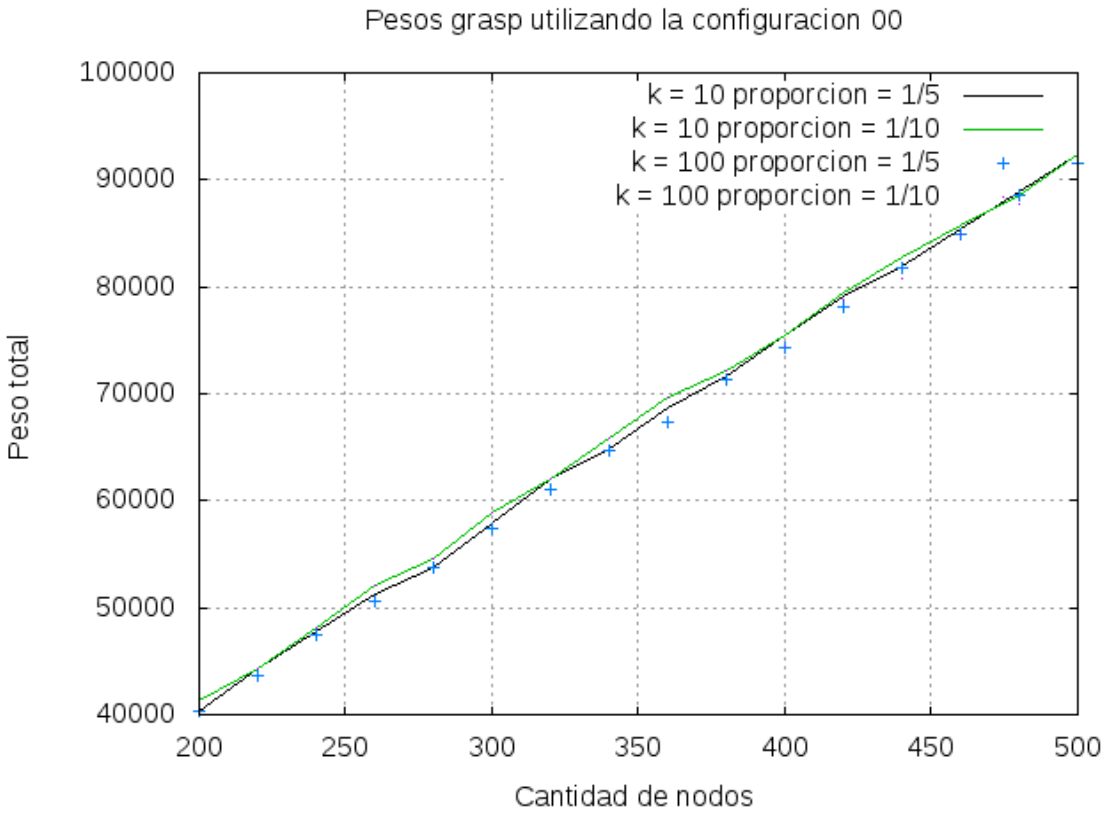
Eligiendo de forma aleatoria sobre los conjuntos de la partición se tomaron los siguiente valores:

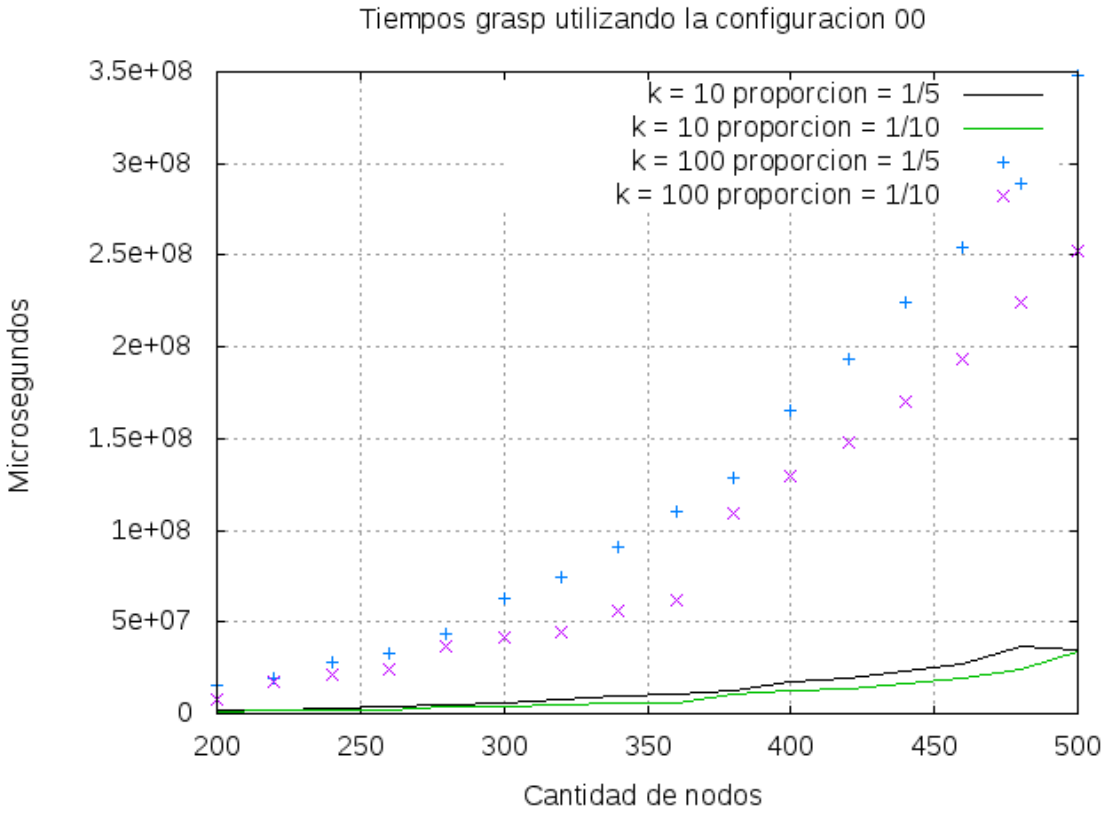
- 1/5 de las bolsas totales.
- 1/10 de las bolsas totales.

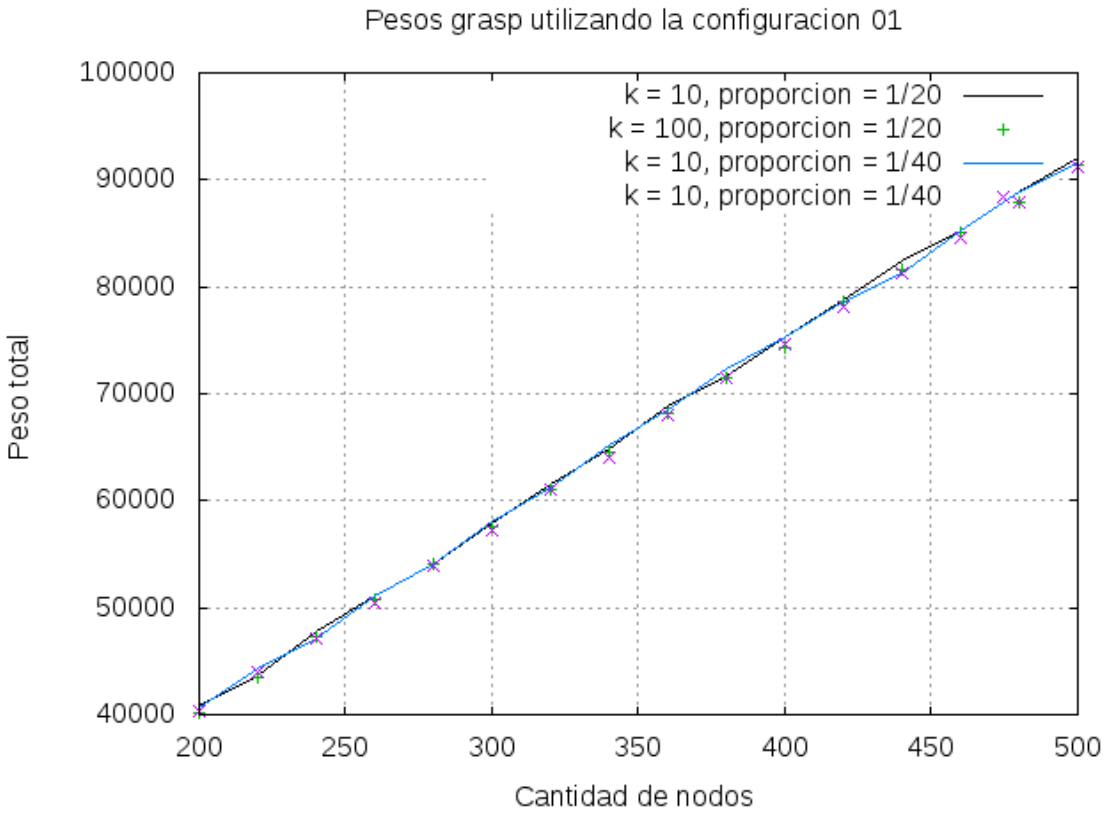
La idea del experimento es comparar cuán buena fue la solución (en términos nominales de peso) y cuánto tardó en encontrar dicha solución la metaheurística GRASP, dada una elección de los criterios. Por esto, dado que hay 16 resultados, dividiremos los resultados por peso y por tiempo de ejecución, y a su vez, por los criterios elegidos, es decir, el **00**, **01**, **10** y **11**.

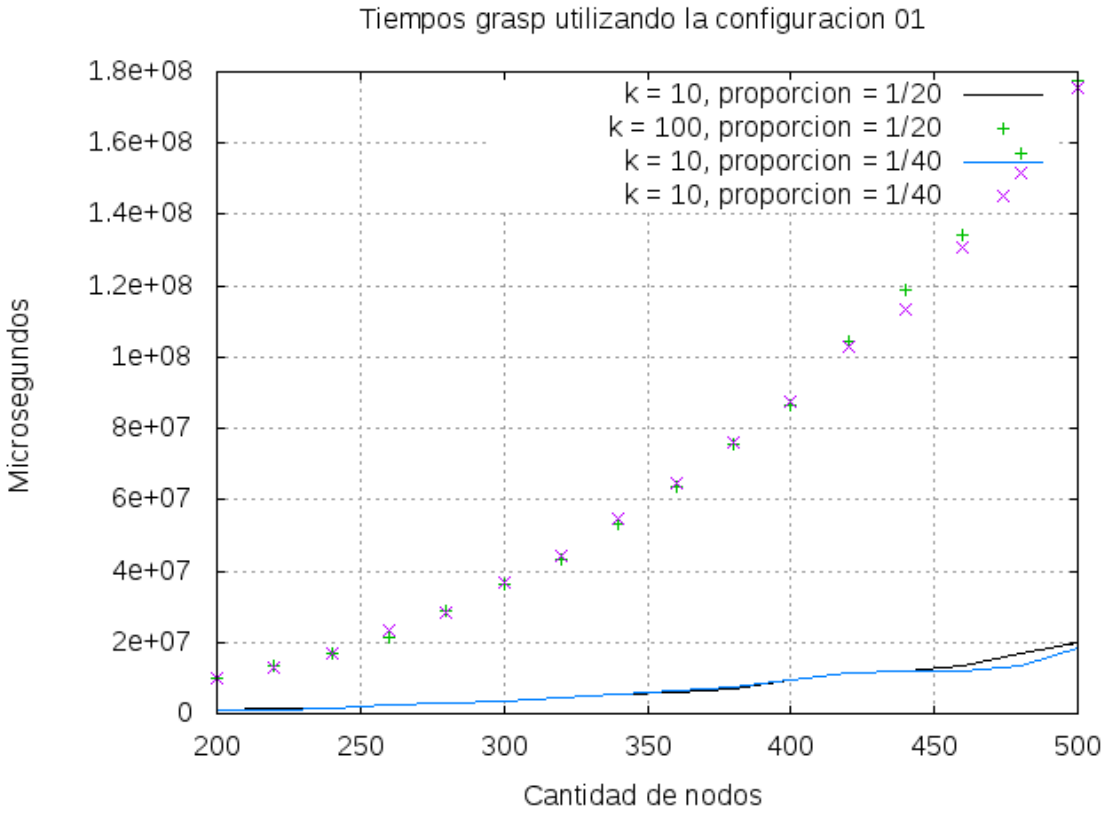
De cada combinación de criterios, elegiremos el que mejor nos pareció y los compararemos entre sí.

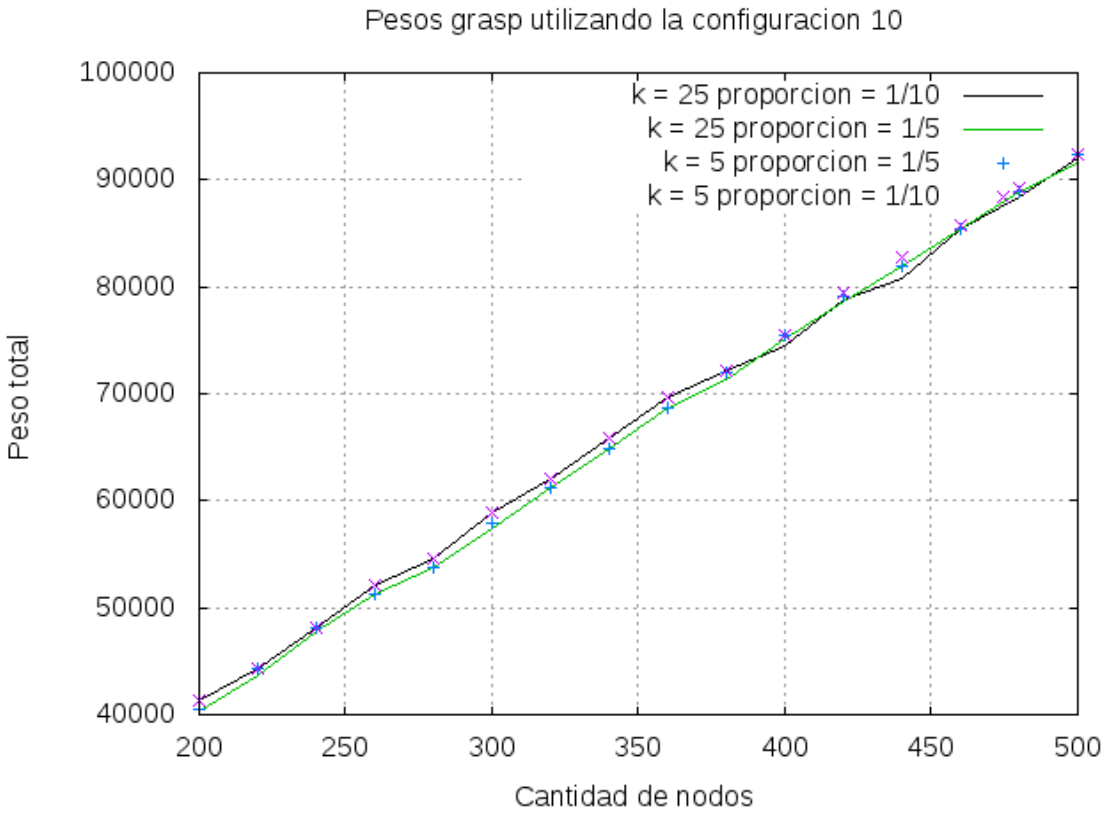
Los grafos de entrada son todos grafos completos, con n variando desde 200 hasta 500 y un k que proporcionalmente a n, $k = n/20$, la semilla utilizada para generar los pesos de las aristas es 13.

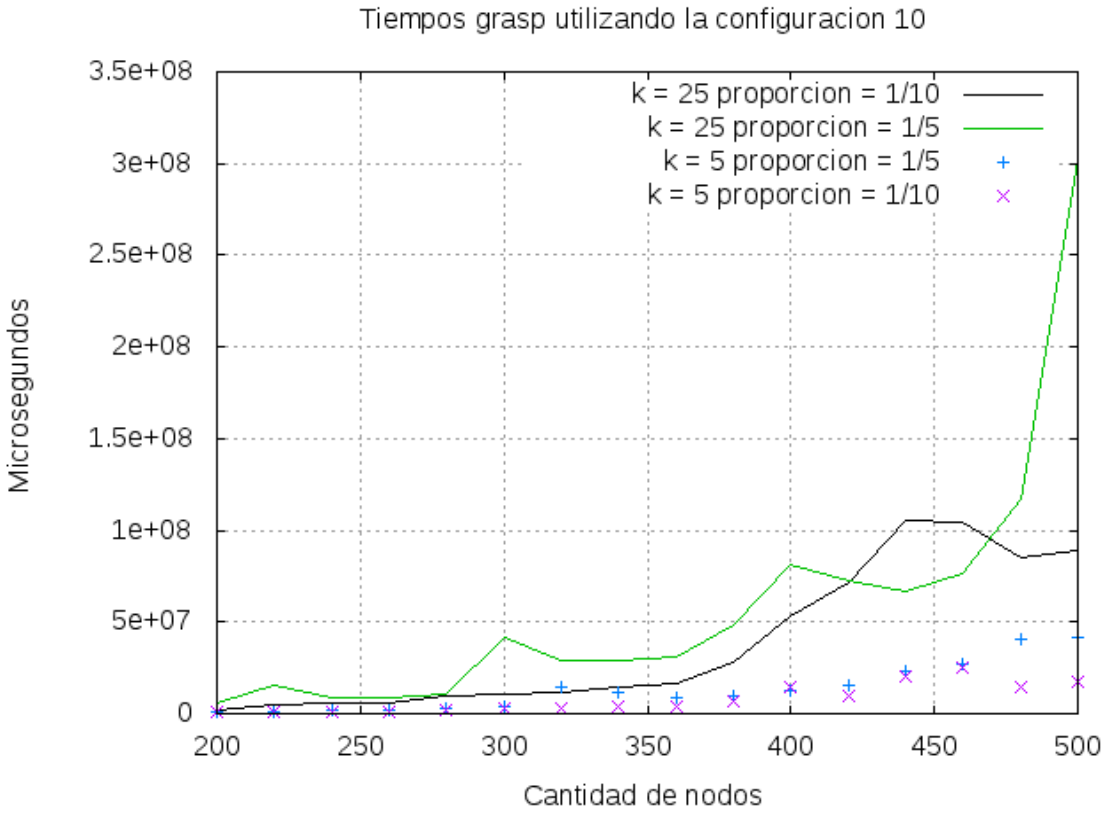

Pesos grasp utilizando la configuracion 00
Peso total
Cantidad de nodos
k = 10 proporcion = 1/5
k = 10 proporcion = 1/10
k = 100 proporcion = 1/5
k = 100 proporcion = 1/10
100000
90000
80000
70000
60000
50000
40000
200
250
300
350
400
450
500

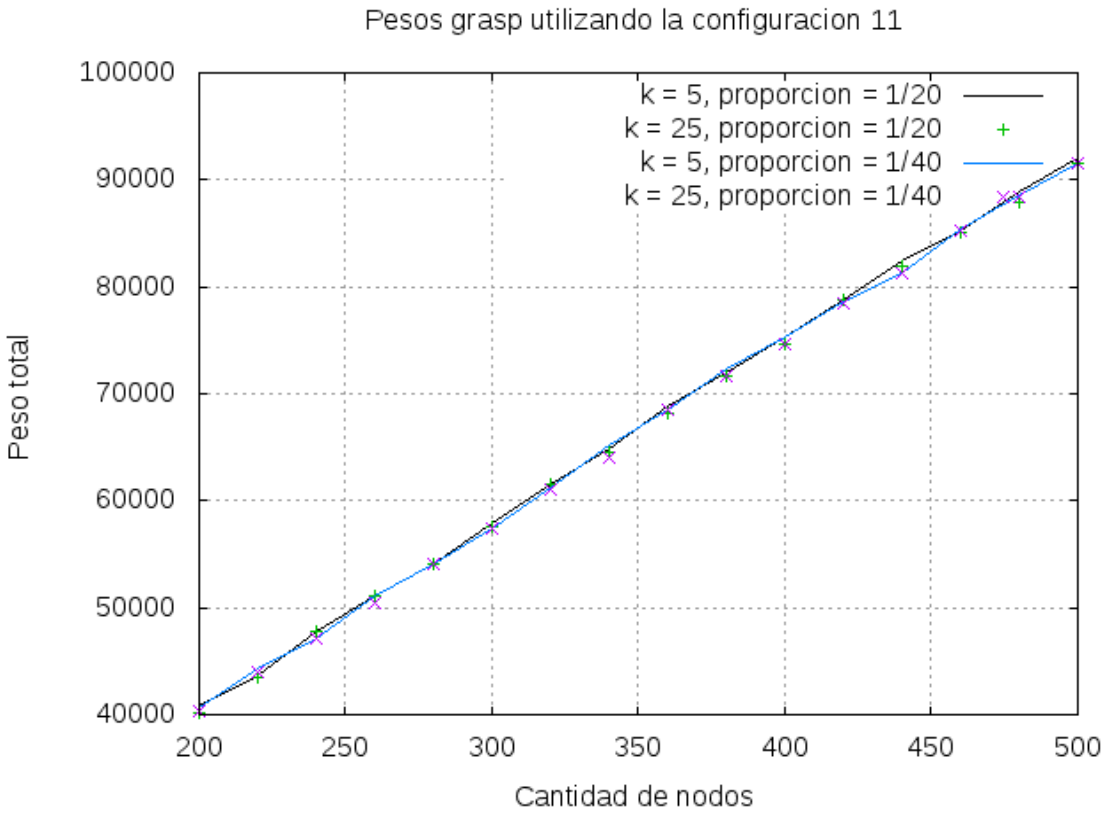

Tiempos grasp utilizando la configuracion 00
Microsegundos
Cantidad de nodos
k = 10 proporcion = 1/5
k = 10 proporcion = 1/10
k = 100 proporcion = 1/5
k = 100 proporcion = 1/10
3.5e+08
3e+08
2.5e+08
2e+08
1.5e+08
1e+08
5e+07
0
200
250
300
350
400
450
500

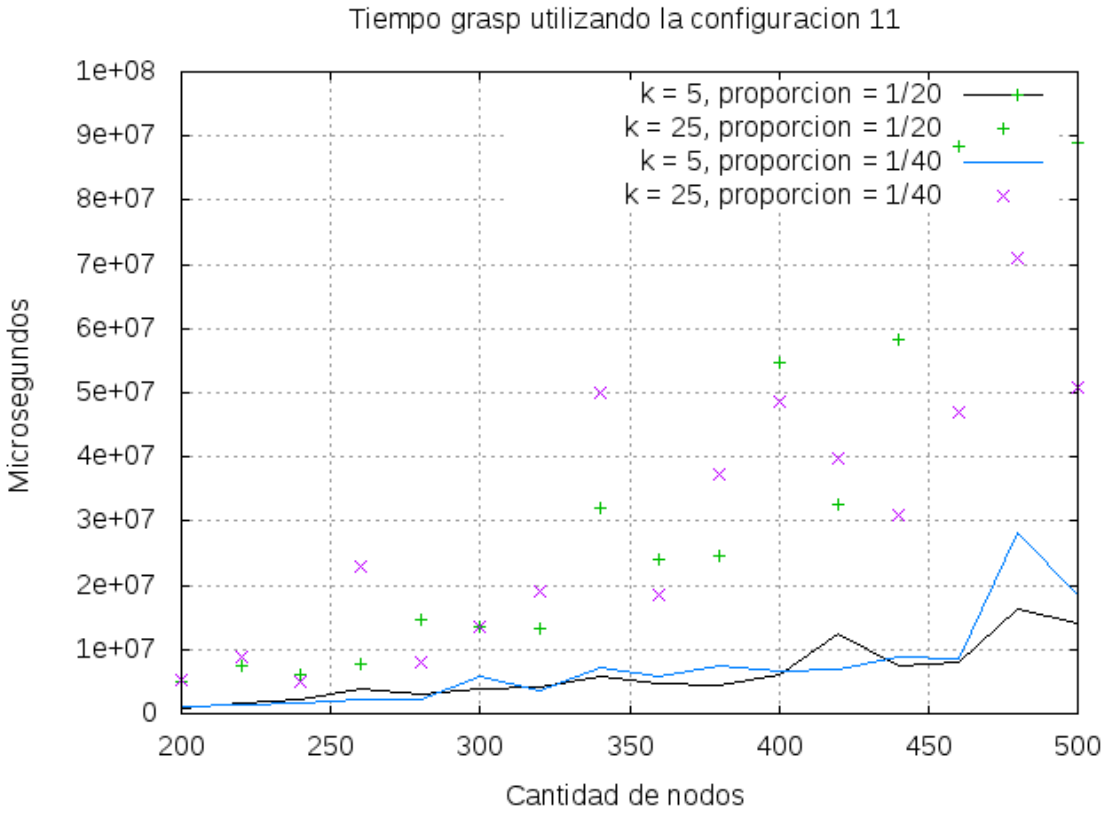

Pesos grasp utilizando la configuracion 01
Peso total
Cantidad de nodos
k = 10, proporcion = 1/20
k = 100, proporcion = 1/20
k = 10, proporcion = 1/40
k = 10, proporcion = 1/40
100000
90000
80000
70000
60000
50000
40000
200
250
300
350
400
450
500


Tiempos grasp utilizando la configuracion 01
Microsegundos
Cantidad de nodos
k = 10, proporcion = 1/20
k = 100, proporcion = 1/20
k = 10, proporcion = 1/40
k = 10, proporcion = 1/40
1.8e+08
1.6e+08
1.4e+08
1.2e+08
1e+08
8e+07
6e+07
4e+07
2e+07
0
200
250
300
350
400
450
500


Pesos grasp utilizando la configuracion 10
Peso total
Cantidad de nodos
k = 25 proporcion = 1/10
k = 25 proporcion = 1/5
k = 5 proporcion = 1/5
k = 5 proporcion = 1/10
100000
90000
80000
70000
60000
50000
40000
200
250
300
350
400
450
500


Tiempos grasp utilizando la configuracion 10
Microsegundos
Cantidad de nodos
k = 25 proporcion = 1/10
k = 25 proporcion = 1/5
k = 5 proporcion = 1/5
k = 5 proporcion = 1/10
3.5e+08
3e+08
2.5e+08
2e+08
1.5e+08
1e+08
5e+07
0
200
250
300
350
400
450
500



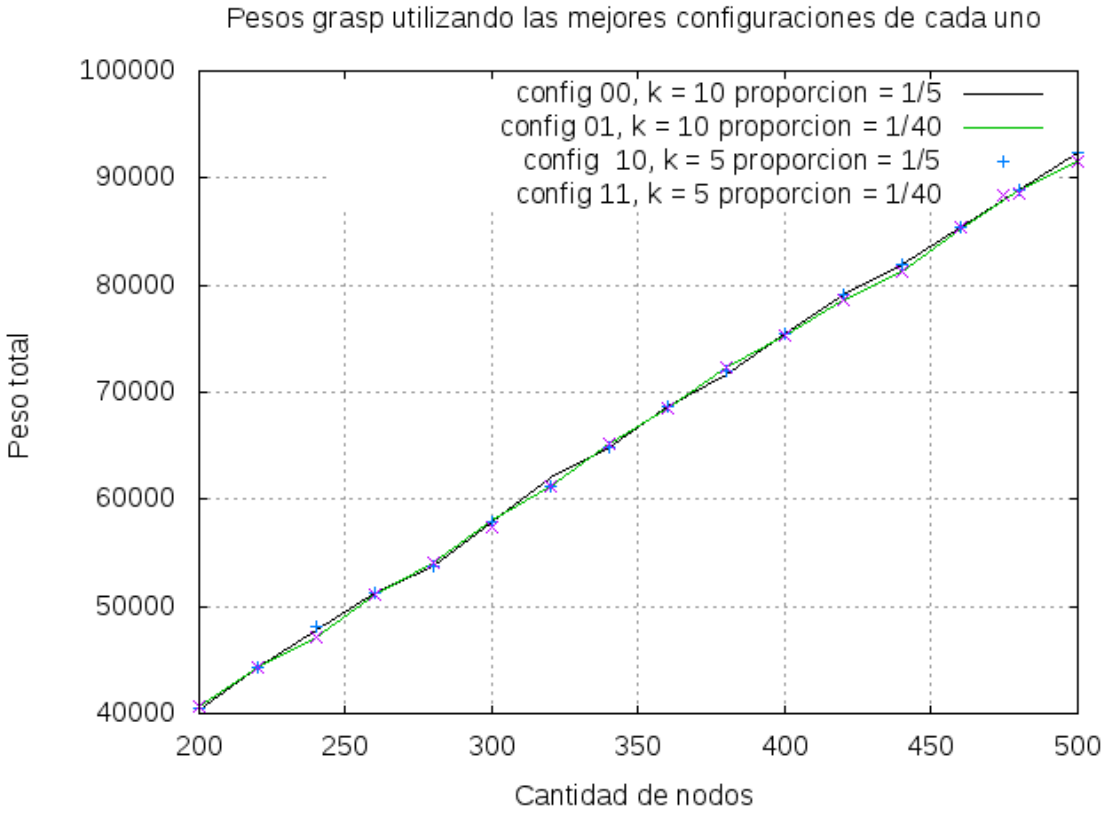


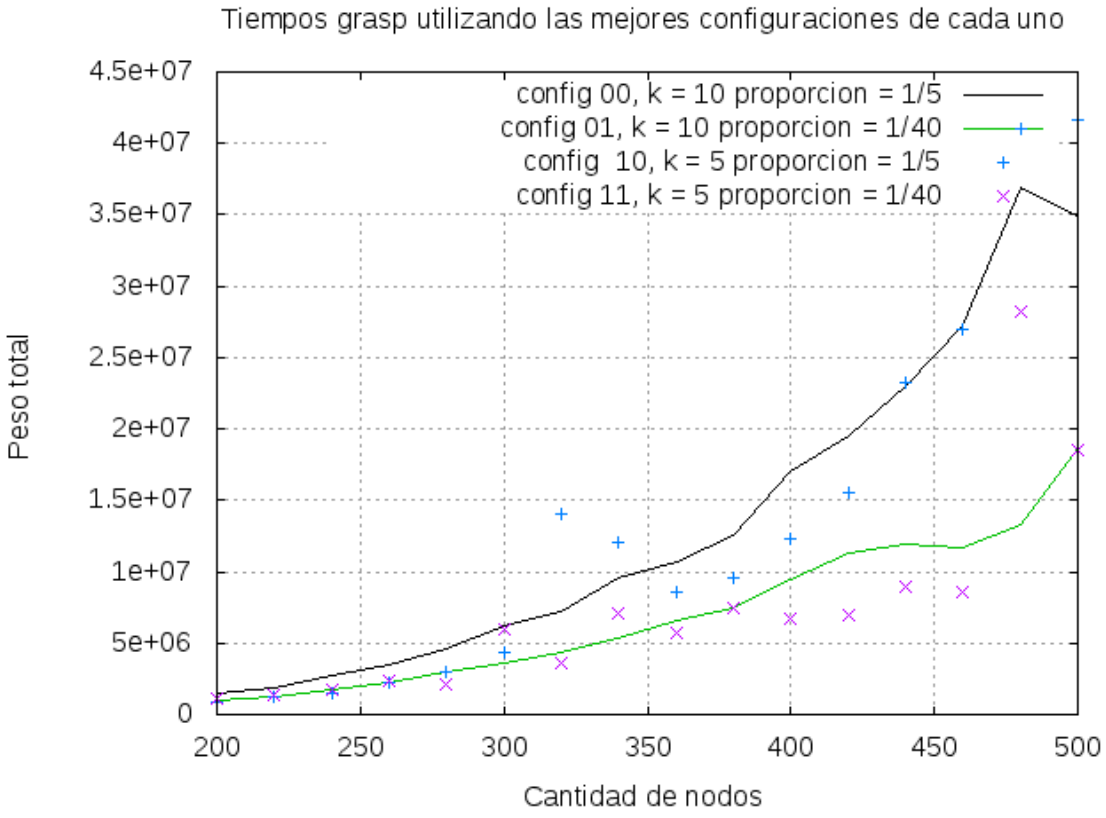
Cómo se ve los distintos gráficos, siempre que corramos a GRASP con muchas iteraciones, alcanzaremos resultados mejores que si lo corremos con menos iteraciones. Sin embargo, las mejoras que consigue no son sustanciales, las curvas no se alejan de forma significativa, pero sí lo hace el tiempo de ejecución, éste se aleja estrepitósamente a medida que la cantidad de nodos aumenta.

Elegir una *mejor* configuración no es una tarea sencilla, pues depende en qué condiciones se corra la metaheurística quizás puede valer la pena, es decir, si se tienen los recursos computacionales para correr vastas cantidades de cómputos puede que sea favorable aumentar el número de iteraciones, pues la solución obtenida nunca puede empeorar. Sin embargo, creemos que no vale la pena pagar el

costo computacional que trae aparejado correr a GRASP con tantas iteraciones, por eso optaremos por elegir las configuraciones que minimizen el tiempo de ejecución.

Como dijimos previamente, tomaremos una configuración de cada elección de criterios y las compararemos entre sí.





Al observar el gráfico de los pesos, vemos que las configuraciones elegidas encuentran soluciones muy similares, si no es que encuentran las mismas. Sin embargo, análogamente con lo mencionado anteriormente, los tiempos de ejecución sí presentan diferencias importantes.

La configuración que minimiza los tiempos de ejecución es la **11** con $k = 5$ y que elige entre los $n/40$ nodos más pesados. Es por esta razón, que en las comparivas generales usaremos una configuración similar a ésta.

7. Ejercicio 6

En esta sección compararemos resultados de todos los algoritmos, tanto en tiempos de ejecución, como en la calidad de la solución obtenida.

Primeramente, para comprar contra el algoritmo exacto, corrimos un experimento que usa grafos completos, con $k = 3$ y la semilla para asignar pesos en 13. El n varía de 5 hasta 22.

Las búsquedas locales utilizan como soluciones aleatorias como iniciales. La configuración de grasp elegida es la **11**, es decir, aleatorizar la elecicción de nodos en la heurística golosa, y parar las iteraciones de grasp cuando no se encuentre mejora después de k-iteraciones. El k elegido es 5, y el β utilizado para el algoritmo goloso es 3.

A continuación se presentan dos cuadros, uno que condensa los pesos obtenidos por cada algoritmo, y otro que refleja los tiempos de ejecución de los mismos.

Grafo Completo - Pesos					
	Exacto	Goloso	B. Local 1-opt	B. Local 2opt	Grasp
K = 3 N =5	25	25	25	49	25
K = 3 N =6	43	43	43	94	43
K = 3 N =7	59	59	59	78	59
K = 3 N =8	87	98	87	96	87
K = 3 N =9	77	228	77	129	77
K = 3 N =10	181	219	195	123	181
K = 3 N =11	280	312	316	396	280
K = 3 N =12	327	412	371	433	327
K = 3 N =13	416	473	419	539	416
K = 3 N =14	456	498	479	611	456
K = 3 N =15	581	686	606	616	590
K = 3 N =16	627	833	706	807	627
K = 3 N =17	650	838	739	747	650
K = 3 N =18	875	974	930	995	875
K = 3 N =19	990	1132	1178	1159	1022
K = 3 N =20	1071	1223	1187	1187	1071
K = 3 N =21	1321	1437	1374	1376	1344
K = 3 N =22	1485	1659	1648	1819	1500

Grafo Completo - Tiempos				
	Goloso	B. Local 1-opt	B. Local 2opt	Grasp
K = 3 N =5	31	20	33	124
K = 3 N =6	34	48	84	183
K = 3 N =7	37	56	149	158
K = 3 N =8	41	91	114	330
K = 3 N =9	35	228	249	419
K = 3 N =10	41	144	239	462
K = 3 N =11	45	211	298	501
K = 3 N =12	48	435	541	588
K = 3 N =13	43	543	459	855
K = 3 N =14	43	607	540	873
K = 3 N =15	49	212	920	1713
K = 3 N =16	53	343	519	1226
K = 3 N =17	58	991	891	1798
K = 3 N =18	62	412	713	1972
K = 3 N =19	89	839	1019	1151
K = 3 N =20	79	581	2709	1646
K = 3 N =21	103	568	2079	1178
K = 3 N =22	87	888	1551	3257

Como se observa en las tabla de pesos, en instancias muy chicas (hasta $n = 7$) todos los algoritmos menos 2-opt encuentran el resultado óptimo, pero a medida que el n aumenta, dicha tarea se dificulta.

Lo que es interesante ver es que casi en pocos casos la búsqueda local 2-opt supera a su hermana 1-opt. Y lo que era esperable, pero aún así debe ser marcado, es que GRASP siempre mejora las soluciones, y hasta en varios casos encuentra el resultado óptimo; incluso para $n = 21, 22$, la solución a la que llega es muy cercana a la óptima. Y lo que es notable es que es *confiable*, es decir, si observamos las demás heurísticas, su cercanía al resultado óptimo va fluctuando, en cambio la metahurística se mantiene estable, lo cual puede ser una cualidad muy valiosa.

Por otro lado, los tiempos de ejecución respetan lo analizado en los itemes anteriores. Para instancias chicas, dichos valores son parecidos entre todas las heurísticas. Pero a medida que el n aumenta, sólo la heurística golosa se mantiene baja. GRASP es quien tiene (en general) los tiempos de ejecición más altos, aunque a veces es superado por la búsqueda local 2-opt (ver $n = 20, 21$).

En segundo lugar, con la intención de ver como se comportan nuestros algoritmos frente al peor caso descripto de la heurística golosa, experimentamos generando grafos que pertenezcan a esta familia, fijando la cantidad de *centrales*[4] en k, para que de dicha forma la solución óptima nunca pueda ser obtenida por la heurística golosa.

La idea de ésto es poder ver si GRASP mejora o llega la solución óptima, partiendo de los resultados no óptimos provistos por la heurística golosa, que como dijimos antes, nunca llega al óptimo.

[4] Recordar que la descripción del peor caso se encuentra en el apartado de la heurística golosa.

Grafo Malo - Pesos					
	Exacto	Goloso	B. Local 1-opt	B. Local 2-opt	Grasp
K = 3 N =5	90	103	90	103	90
K = 3 N =6	90	132	90	132	90
K = 3 N =7	90	158	90	158	90
K = 3 N =8	90	161	161	161	90
K = 3 N =9	90	169	169	169	90
K = 3 N =10	90	183	183	183	90
K = 3 N =11	90	192	192	192	90
K = 3 N =12	90	217	217	217	90
K = 3 N =13	90	228	228	228	90
K = 3 N =14	90	271	271	271	90
K = 3 N =15	90	290	290	290	90
K = 3 N =16	90	290	290	290	90
K = 3 N =17	90	302	302	302	90
K = 3 N =18	90	317	317	317	90
K = 3 N =19	90	321	321	321	90
K = 3 N =20	90	339	339	339	90
K = 3 N =21	90	366	366	366	90
K = 3 N =22	90	370	370	370	90

Grafo Malo - Tiempos				
	Goloso	B. Local 1-opt	B. Local 2-opt	Grasp
K = 3 N =5	31	20	33	125
K = 3 N =6	43	48	84	138
K = 3 N =7	30	56	149	178
K = 3 N =8	40	91	114	160
K = 3 N =9	61	228	249	184
K = 3 N =10	50	144	239	196
K = 3 N =11	32	211	298	216
K = 3 N =12	59	435	541	227
K = 3 N =13	46	543	459	258
K = 3 N =14	44	607	540	267
K = 3 N =15	71	212	920	291
K = 3 N =16	67	343	519	309
K = 3 N =17	50	991	891	341
K = 3 N =18	77	412	713	358
K = 3 N =19	81	839	1019	381
K = 3 N =20	94	581	2709	410
K = 3 N =21	85	568	2079	449
K = 3 N =22	97	888	1551	457

En primera instancia, es importante aclarar que no es un error que la solución óptima sea siempre la que tiene de peso 90, si no que se debe a que es el mínimo peso que existe entre dos nodos *centrales*, y que como la cantidad de dichos vértices es igual al k, la solución óptima está dada por poner a los *satélites* en un conjunto, un *central* solo en un conjunto y los últimos dos *centrales* en otro conjunto. De esta forma solamente existe una arista en el conjunto que comparten los dos *centrales*, siendo ésta de peso 90.

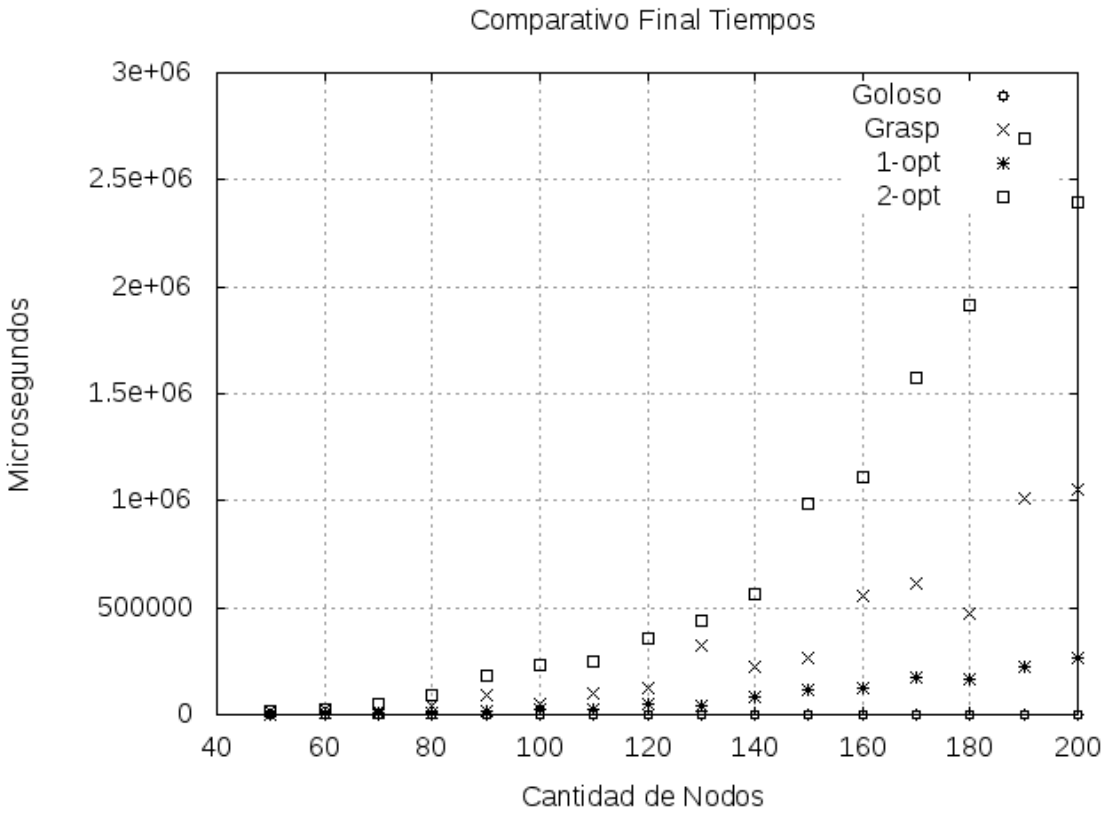
Dicho ésto, veamos cómo a partir de $n = 8$ solamente GRASP consigue llegar a la solución óptima, y siempre lo hace. No solo que llega a la óptima, si no que lo hace bastante rápido, pues el tiempo de ejecición de dicho algoritmo es menor a caulquiera de las dos búsquedas locales. Ésto significa que debe encontrar la solución óptima relativamente rápido, pues recordar que la configuración utilizada es *no se encontró mejora en 5-iteraciones*.

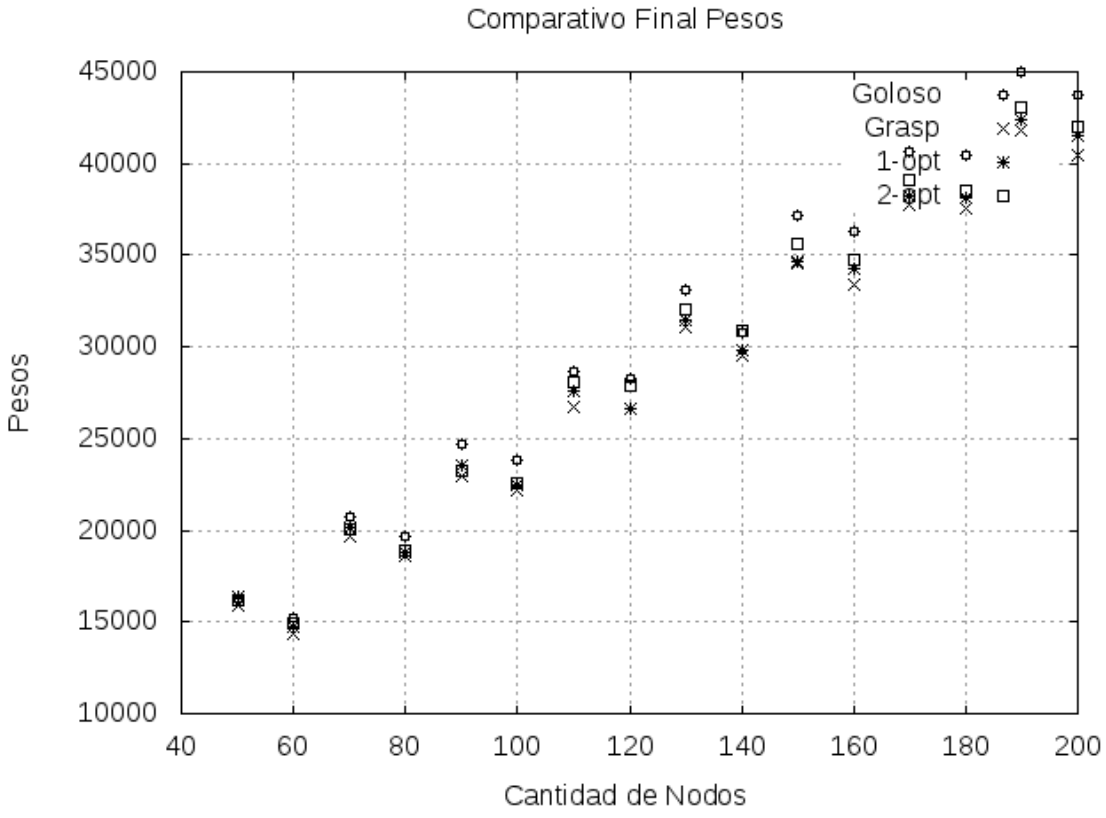
El último experimento realizado consiste en comparar los tiempos de ejecución y los pesos obtenidos por todos los algoritmos menos el exacto, pues en estas instancias no se lo puede correr (debido a su naturaleza exponencial).

Tomamos grafos completos, desde $n = 40$ hasta $n = 200$, incrementando de a 20, genrando así 16 resultados. Cada grafo fue creado con una semilla distinta.

GRASP fue corrido con la configuración **11** con la cantidad de iteraciones fijadas en 5, y siendo el β de la heurística golosa $n/25$, es decir, una proporción de n que varía en cada iteración.

Las herísticas de búsqueda local toman soluciones aleatorias como sus soluciones iniciales.





Como se puede observar en el gráfico de tiempos, a riesgo de ser repetitivo, GRASP se posiciona entre la 1-opt y 2-opt, siendo la segunda demasiado alta, y la segunda no siendo tanto más baja que GRASP.

En este experimento podemos ver lo que planteamos a lo largo de las experimentaciones comparativas. GRASP siempre encuentra una solución mejor que las de sus rivales (aunque no tanto más en la mayoría de los casos), y su costo computacional es lo suficientemente bajo como para considerarlo como una buena opción a elegir.

8. Conclusión

A partir de lo experimentado con GRASP y sus configuraciones y luego la comparativa final, podemos decir que por un lado, la heurística golosa desarrollada no es demasiado buena, porque limita mucho las acciones de la búsqueda local, ya que en general la mejora provista por la búsqueda local no es significativa.

Por otro lado, es interesante ver que aquí aparece una discución que es, por lo menos recurrente, en la computación, tiempo/recursos vs optimalidad. Si en la configuración de GRASP, incrementamos el número de iteraciones que éste corre, seguramente encontraremos mejoras en nuestros resultados. Pero, hasta qué punto está uno dsipuesto a resignar tiempo o utilizar recursos para llegar a un resultado que puede presentar mejoras, pero no significativas con respecto a resultados obtenidos utilizando menos tiempo.

Por último, nos gustaría decir que si bien el tiempo no fue un factor sobrante para nuestro grupo, este fue un trabajo práctico más que interesante, el cual tratamos de aprovechar y aprender todo lo que pudimos sobre qué experimentar, y cómo hacerlo.