

Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Reentrega Trabajo Práctico N° 3

Grupo N°11

Integrante	LU	Correo electrónico
Tomas Zulberti	908/04	tzulberti@gmail.com

Reservado para la ctedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

Cambios

General

- Se compara la calidad de la solución cuando se analiza el algoritmo.
- Se crearon mas archivos de entrada¹

Algoritmo Exacto

- Se agrego una nueva poda. Esto trajo como consecuencia el cambio de la complejidad y del peor caso.

Algoritmo Constructivo

- Se saco la opcion del Random
- Los algoritmos golosos se cambio la implementación y eso cambio en una mayor complejidad. No cambio el peor caso del algoritmo, pero cambio la forma en la que el mismo se comporta cuando se tiene un α y β de forma tal que hacen que el algoritmo sea random. Por esta razon, se calculo de nuevo la complejidad y se actualizaron los gráficos.

Heuristica de Busqueda local

- Se saco la opcion de agregar o sacar Random
- Se cambio el algoritmo para que la busqueda local se haga por cada componente conexa del grafo. Eso trajo un cambio de la complejidad. Ademas, se encontro un mal caso para el algoritmo.
- Ahora solo se mueve al vecino si tiene un mejor valor (antes era necesario de que tenga mejor o igual valor) Se saco la variable *mejoraPedida*
- La variable *cantIteraciones* paso a ser un porcentaje de la cantidad de vertices de la componente conexa actual.

GRASP

Mas o menos lo mismo que en la busqueda local.

¹Ver Heuristicas y Algoritmos, Archivos de entrada.

1. Casos de uso en la vida real

El único caso de uso en donde se nos ocurrió que se podía llegar a aplicar esto en la vida real es pensar en un divorcio. Supongamos que cada uno de los vértices del grafo son los objetos a dividir entre las dos personas, y hay un eje entre los dos objetos si solo se puede elegir uno de ellos. Además, cada objeto tiene un precio/valor. Por lo tanto, uno quiere elegir los objetos de forma tal que se maximice el valor.

2. Heurísticas Y Algoritmos

2.1. Estructuras Usadas

Hay dos objetos que fueron usados en todos los algoritmos por lo que se los detalla en esta sección. Los dos objetos usados son:

- El grafo. Para todos los algoritmos se uso la misma representación del grafo.
- La solución.

2.1.1. Grafo

El grafo se representa usando:

- Un arreglo de longitud de la cantidad de vértices del mismo. La posición k de ese arreglo representa el peso que tiene el vértice k .
- Un diccionario donde las claves son cada uno de los vértices del grafo, y el valor una lista que contiene todos los vértices adyacentes a la clave. Para implementar este diccionario, las claves se guardan sobre un arreglo, y los valores son listas.
- La cantidad de vértices del grafo.²
- El peso de todos los vértices del grafo, es decir, el peso de todos los vértices sumados.

Esta representación del grafo permite que:

- Saber la cantidad de vértices del grafo en $O(1)$. Esto es porque solo se tiene que verificar la longitud del arreglo
- Saber el peso de cualquier vértice en $O(1)$.
- Obtener los adyacentes al vértice k en $O(1)$. Esto es porque los valores del diccionario son los adyacentes a la claves, y por lo tanto solo se tiene que acceder a la posición k del arreglo.
- Saber si el peso de la solución actual es el máximo posible en $O(1)$, ya que tan solo se tiene que comparar con el peso del grafo.

2.1.2. Solución

La solución se representa usando:

- Un arreglo de booleanos, que tiene la longitud de la cantidad de nodos del grafo. Si la posición k del arreglo es verdadero, entonces eso indica que el vértice k del grafo se encuentra en la solución.
- El valor de la suma de los pesos vértices actuales.
- Un arreglo de enteros, cuya longitud es la cantidad de nodos del grafo. Si la posición k tiene un valor j , esto indica que hay j nodos en la solución actual que son adyacentes a k .

Cuando se crea una solución:

- El valor de la suma de los pesos se actualiza en 0.
- Todas las posiciones del arreglo de booleanos se ponen en falso.
- Todas las posiciones del arreglo de adyacentes se ponen en 0.

Ahora vemos las operaciones que se usan sobre esta estructura:

²En la implementación esto lo obtenemos pidiendo la longitud del arreglo de pesos.

- El hecho de verificar si es la mejor solución posible se hace en $O(1)$. Esto es porque el hecho de obtener el peso de todos los vértices del grafo se hace en $O(1)$, y lo mismo para obtener el peso de la solución actual. Esta función figura bajo el nombre *esValido* en los diferentes pseudocódigos. Que esa función *esValido(solución, vertice)* devuelva verdadero significa que el vértice puede ser agregado a la solución.
- Agregar un vértice a la solución tiene complejidad $O(n)$, siendo n la cantidad de vértices del grafo. Esto ocurre porque cuando se agrega un vértice a la misma se tiene que:
 - Actualizar el peso de la solución. Esto es $O(1)$ porque el grafo permite obtener el peso de cualquier vértice en $O(1)$.
 - Actualizar el arreglo de booleanos. Esto también es $O(1)$ porque se tiene que actualizar la posición del vértice agregado.
 - Actualizar el arreglo de cantidad de adyacentes. Para esto se tiene que obtener la lista de adyacentes del vértice agregado, y para cada uno de los vértices adyacentes sumarle 1 a la posición de ese vértice.

Todas las operaciones son $O(1)$, salvo el hecho de actualizar los adyacentes. Como un vértice cualquiera puede a lo sumo ser adyacente a $n-1$, entonces la complejidad de esta función es $O(n)$.

- Verificar si se puede agregar un vértice k a la solución se hace en $O(1)$. Esto es porque para esto se tienen que verificar dos cosas:
 - Que el arreglo de adyacentes en la posición k sea 0, y por lo tanto, que ningún vértice que se encuentre en la solución sea adyacente al mismo.
 - Que el vértice no se encuentre en la solución.

Como los dos casos son de acceder a una posición de arreglos, entonces esto es $O(1)$.

- Sacar un vértice de la solución tiene complejidad $O(n)$. Esto ocurre porque:
 - Se tiene que actualizar el peso de la solución. Esto es $O(1)$ porque el grafo permite obtener el peso de cualquier vértice en $O(1)$.
 - Se tiene que actualizar el arreglo de booleanos. Esto también es $O(1)$ porque se tiene que actualizar la posición del vértice sacado.
 - Se tiene que actualizar el arreglo de cantidad de adyacentes. Para esto se tiene que obtener la lista de adyacentes del vértice sacado, y para cada uno de los vértices adyacentes restarle 1 a la posición de ese vértice.

Por la misma razón que la función *agregar*, el orden de esta función es $O(n)$

- Copiar la solución. Como se tienen que copiar 2 arreglos y un valor entero, y los dos arreglos tienen longitud n , entonces la complejidad de esta función es $O(n)$.

2.2. Notas Aparte

Algunas notas importantes para no tener que repetirla en cada uno de los algoritmos:

- Todas las complejidad están calculadas en el modelo uniforme. Esto se debe a que el orden de los algoritmos va a depender más de la cantidad de nodos del grafo, que del valor de cada uno de los vértices del mismo.

2.3. Archivos de Entrada

Para poder comparar los diferentes algoritmos se nos ocurrieron los siguientes variantes:

- Variar Componentes Conexa: acá lo que se hace es variar la cantidad de componentes conexas del grafo mientras que los pesos y grado de los vértices se mantienen constantes en las variaciones. Se hicieron archivos de entrada cuando la componente conexa era de 10, 20, 50 y 70 vértices.
- Variar grado: acá lo que se hace es variar el grado de los vértices. Para esto se elige una cantidad al azar de vértices adyacentes. En este caso, no sabemos la cantidad de componentes conexas del grafo, ya como varía el grado y los vértices adyacentes son elegidos al azar, no podemos identificar cuantas componentes conexas se forman en cada grafo. Sin embargo, el peso de los vértices si se mantiene constante. Para este caso, los grados fueron 5, 10, 20, 50 y 70 de los vértices.³

³No todos los vértices tienen ese grado, pero es bastante cercano. Esto se debe a un error de implementación.

- Variar grado por componente: es similar al anterior, salvo que sabemos la cantidad de vértices tiene la componente conexa. En estos casos la cantidad de vértices se deja fija, al igual que los pesos de los vértices, pero cambia el grado de los vértices, por lo que la componente conexa se va volviendo más completa. Para esto, se fijo la componente conexa para que sean de 30 vértices, y se variaba el grado entre: 2, 5, 10, 27 y 25.
- Variar el peso cuando está separado por componentes conexas. En este caso, se dejó fijo la cantidad de vértices por componente conexa, y el grado de las mismas, y se variaba el peso de los vértices. Los variantes para el vértice v_i fueron las siguientes:
 - Que el peso sea $i/2$
 - Que el peso sea i
 - Que el peso sea $2*i$
 - Que el peso sea i^2
 - Que el peso sea un valor al azar entre 1 y la cantidad de vértices del grafo.
- Variar todo: grafos totalmente generados al azar. En este caso, tanto el grado de los vértices como el peso de los mismos era elegido de forma al azar. Al variar el grado al azar de cada uno de los vértices, entonces no se puede identificar de antemano la cantidad de componentes conexas del grafo. Estos casos están planteados para que el mejor caso de cada una de las heurísticas sea comparado con el algoritmo exacto.

Como algunos archivos involucran cosas que son elegidas al azar, todas las heurísticas pasaron los mismos archivos, haciendo así que la variante que es al azar tenga el mismo valor para las diferentes heurísticas.

3. Algoritmo Exacto

3.1. Introducción

La primera idea que tuvimos para este era la de hacer fuerza bruta. Es decir, formar todos las posibles combinaciones de vértices. Fijarnos para cada uno de ellos si era válido, y descartar los que no eran válidos. Luego, quedarnos con el máximo de los válidos.

Lo que se nos ocurrió después fue hacer una mejora sobre el caso anterior. A medida que vamos armando el conjunto de vértices actual, nos podemos fijar si el mismo es válido o no. En cuanto encontramos que no es válido, se lo descarta. Por lo tanto, luego quedaría una lista con todos los posibles conjuntos independientes del vértices del grafo que son validos, y solo se tendría que buscar el de mayor peso.

El principal problema con la idea anterior es que se usa demasiada memoria. Es decir, se tiene que guardar una lista con todas las posibles soluciones, que es mayor que $n!$ ya que como posible solución se pueden tener los vértices de distinta forma. Tiene que ser mayor que ese valor, porque $n!$ solo contempla el caso donde están todos los vértices en la solución. Falta tener en cuenta cuando no todos los vértices se encuentran en la solución.

Para evitar esto, se nos ocurrieron dos ideas:

- La primera es que la solución sea maximal en la cantidad de vértices, es decir, las soluciones a las cuales no se les pueda agregar mas vértices sin que dejen de ser validas. Por lo tanto, en la lista solo voy a agregar los conjuntos de vértices que sean maximales. Por ejemplo, si tengo el la solución $\{1, 2\}$, y la $\{1, 2, 3\}$, entonces se descarta la primera porque no es maximal.
- Buscar la primera solución válida, llamemosla *mejorSolución*. Una vez encontrada la misma, seguimos buscando otra solución, llamemosla *soluciónActual*. Si la *soluciónActual* es mejor que la *mejorSolución*, entonces se reemplaza la *mejorSolución*. En cualquiera de los dos casos, el algoritmo se repite hasta que no se puedan encontrar más soluciones.

La ventaja de esa idea con respecto a la segunda, es que esta solo tiene en memoria dos soluciones: la *soluciónActual* y la *mejorSolución*. Por lo tanto, no usa tanta memoria. El hecho de que sea maximal reduce la cantidad de soluciones a tener en cuenta.

Por último, nos dimos cuentas de 3 cosas importantes:

- Si la *mejorSolución* tiene todos los vértices del grafo, entonces sabemos que el algoritmo lo podemos terminar ahí porque no va a existir una mejor solución.
- Dada una *soluciónActual*, se pueden formar el conjunto de vértices que no se encuentran en la solución y que pueden ser agregados a la misma, llamemoslo *vérticesAAgregar*. Sin embargo, si el peso de la solución y el peso de los vértices del conjunto *vérticesAAgregar*, es menor que el peso de la *mejorSolución*, entonces no vale la pena seguir metiendo vértices a la *soluciónActual*, ya que por mas de que se puedan meter todos los vértices restantes no es posible que supere la *mejorSolución*.
- En el conjunto de vértices *vérticesAAgregar* solo van a figurar los vértices que son mayores al ultimo vértice agregado. Esto es para que se evite tener en cuenta soluciones en donde lo único que cambia es la posición de los vértices. Por ejemplo, las soluciones $\{1, 2, 3\}$ y la $\{3, 1, 2\}$ son iguales. Por eso, si en la lista de los vértices posibles solo tengo los mayores al ultimo vértice agregado, entonces no es posible que se formen soluciones repetidas. Sin embargo, esto no limita el hecho de que se revisen todas las soluciones, ya que si se llega a agregar un vértice v_2 , y el vértice v_1 es menor que el mismo, entonces esa solución se tuvo en cuenta anteriormente cuando la *soluciónActual* cuando el primer vértice que se tuvo en cuenta fue v_1 y luego v_2 .

Resumiendo, lo que se nos ocurrió hacer es un algoritmo de backtracking en donde en el mismo se vayan agregando vértices a la *soluciónActual*. Además, en otra variable, se va a guardar la *mejorSolución*. Además, el mismo va a tener 2 podas:

- Que la *mejorSolución* llegue a tener todos los vértices del grafo.
- Que la suma del peso de los vértices que es posible agregarle a la *soluciónActual* mas el peso del mismo sea mayor que el peso de la *mejorSolución*.

3.2. Pseudocódigo

Algorithm 1 iniciarBacktracking(grafo)

```

Sea soluciónActual ←  $\emptyset$ 
Sea mejorSolución ←  $\emptyset$ 
Sea vérticesAAgregar una lista con todos los vértices del grafo
devolver backtracking(grafo, mejorSolución, soluciónActual, vérticesAAgregar)

```

end

Algorithm 2 backtracking(grafo, mejorSolución, soluciónActual, vérticesAAgregar)

```

for vértice ∈ vérticesAAgregar do
    if esValido(soluciónActual, vértice) then
        Sea soluciónAux una copia de soluciónActual
        Agregar vértice a soluciónAux
        if peso(soluciónAux) == peso(grafo) then
            devolver soluciónAux
        else
            Sea vérticesAAgregarAux ← recalcular(grafo, soluciónAux, vérticesAAgregar, vértice)
            if sumaPesos(grafo, soluciónAux, vérticesAAgregarAux) > peso(mejorSolución) then
                Sea soluciónTmp ← backtracking(grafo, mejorSolución, soluciónAux, vérticesAAgregarAux)
                if peso(soluciónAux) < peso(soluciónTmp) then
                    soluciónAux ← soluciónTmp
                end if
                if peso(soluciónAux) == peso(grafo) then
                    devolver soluciónAux
                else
                    if peso(soluciónAux) > peso(mejorSolución) then
                        mejorSolución ← soluciónAux
                    end if
                    end if
                end if
            end if
        end if
    end for
    devolver mejorSolución

```

end

Algorithm 3 recalcular(*grafo*, *soluciónAux*, vérticesAAgregar, *ultimoVérticeAgregado*)

```

Sea listaResultado ←  $\emptyset$ 
for vértice ∈ vérticesAAgregar do
    if esValido(vértice, soluciónAux) y vértice > ultimoVérticeAgregado then
        Agregar vértice a listaResultado
    end if
end for
devolver listaResultado

```

end

Algorithm 4 sumaPesos(grafo, soluciónAux, vérticesAAgregarAux)

```

Sea sumaPeso  $\leftarrow$  peso(soluciónAux)
for vértice  $\in$  vérticesAAgregarAux do
    sumaPeso  $\leftarrow$  sumaPeso + peso(grafo, vértice)
end for
devolver sumaPeso
end
```

3.3. Complejidad

Veamos la complejidad de las funciones auxiliares para luego poder calcular la complejidad de la función principal.

En la función *sumaPesos*, todas las operaciones que se hacen se hacen en $O(1)$. Por lo que habría que ver cuantas veces se hace el ciclo de iterar sobre los vértices de *vérticesAAgregarAux*. Como esta lista, a lo sumo tiene todos los vértices del grafo, entonces la complejidad de la función es:

$$O(n)$$

donde n es la cantidad de vértices del grafo.

Ahora veamos la función *recalcular*. Al igual que en la función *sumaPesos*, todas las operaciones que se hacen en la misma son $O(1)$. Como se itera la misma lista, entonces el orden va a ser el mismo. Luego, esta función tiene orden $O(n)$.

Por último, nos queda el hecho de la función que inicializa los valores antes de llamar al backtracking. Como el mismo crea 2 soluciones vacías, las operaciones de hacer eso es $O(1)$, y el hecho de obtener una lista con todos los vértices del grafo se hace en $O(n)$.

Como terminamos de analizar todas las complejidad de las funciones auxiliares, ahora analizamos la complejidad de la función principal. Por cada uno de los ciclos que se hacen en la función *backtracking*, se tiene la siguiente complejidad:

- Copiar la solución: $O(n)$
- Agregar un vértice a la solución: $O(n)$
- Recacular los vértices posibles a meter: $O(n)$
- Verificar la poda *sumaPesos*: $O(n)$
- Todas las otras operaciones: $O(1)$

Como todas estas operaciones se hacen en la misma iteración, entonces por cada ciclo, la complejidad es:

$$O(n + n + n + n + 1) \in O(n)$$

Si pensamos todas las soluciones posibles del problema (sean validas o no), y los representamos como un grafo, obtenemos un árbol que tiene:

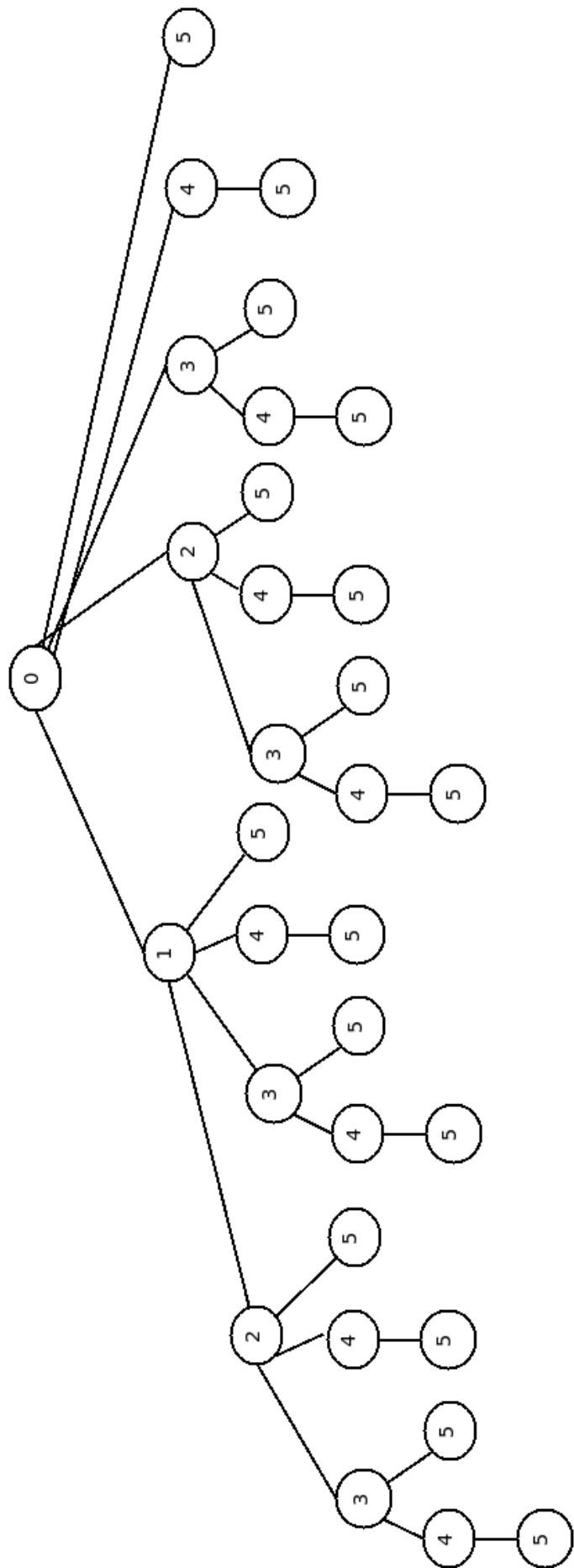
$$\sum_{i=1}^n \frac{n!}{(n-i)!}$$

ramas, y donde i es la cantidad de vértices que se tienen en la solución. Es decir, entre todas las soluciones de la fuerza bruta se encuentra la opción de tomar un único vértice, la de tomar dos vértices teniendo en cuenta el orden, etc... Como i es la cantidad de vértices que fueron agregados a la solución, entonces sabemos que cada rama va a ser de altura i . Sin embargo, eso es en la fuerza bruta.

Veamos ahora el caso del algoritmo. Supongamos que por alguna razón, las dos podas nunca llegan a acotar, es decir, nunca se llega a una solución que contiene todos los vértices, y siempre se puede seguir por la rama dado que se puede llegar a mejorar la solución. Si el primer vértice agregado fue i , entonces sabemos que todas las posibles ramas del mismo van a tener longitud a lo sumo $n - i$, ya que todos los vértices menores que i no van a ser agregados a la solución. Por lo que quedaría de ver, cuantas ramas existen. Sabemos que no van a poder existir mas de:

$$\sum_{j=1}^{n-i} \frac{(n-i)!}{(n-i-j)!}$$

ramas, que sería formar todas las soluciones posibles con esos vértices. Sin embargo, veamos de acotar un poco mejor. Por ejemplo, para 5 vértices, las posibles soluciones que se van a tener en cuenta son las siguientes:



En este gráfico, las ramas llegan hasta el peor caso, es decir, agregan todos los vértices posibles. Se puede notar, que la cantidad de ramas para un vértice dado depende de la cantidad de ramas que tienen los vértices que son mayores que el mismo. Por lo tanto, planteamos la siguiente ecuación para calcular la cantidad de ramas:

$$cantRamas(i) = \begin{cases} 1 & \text{si } i=n, \\ \sum_{k=i+1}^n cantRamas(k) & \text{si } i \neq n. \end{cases}$$

Por lo tanto, se tiene que la cantidad de veces que se hace el ciclo principal es 2^n ⁴. Sabemos que por cada eje del grafo se hacen operaciones que tienen una complejidad $O(n)$. Sabemos que si una rama llegase a tener altura n , entonces el algoritmo termina porque en ese caso todos los vértices del grafo fueron agregados a la solución. Sin embargo, esto no sucede si se tiene la altura llegase hasta $n-1$. Por lo tanto, la complejidad del algoritmo va a ser:

$$O(n + n * (n - 1) * 2^n) \in O(n^2 * 2^n)$$

Por último, nos queda calcular el tamaño de entrada de este problema. El mismo va a ser:

$$\begin{aligned} T &= \log(n) + \sum_{i=1}^n \log(peso(i)) + \sum_{i=1}^n \sum_{v \in ad(i)} \log(v) \\ T &\geq \sum_{i=1}^n \log(peso(i)) \\ T &\geq \sum_{i=1}^n 1 \\ T &\geq n \end{aligned}$$

Reemplazando en la complejidad calculada anteriormente nos queda que:

$$O(T^2 * 2^T)$$

y la misma es exponencial.

3.4. Gráficos de tiempo

Ahora veamos como se comporta el algoritmo si solo se tiene en cuenta el tiempo usado por el mismo y no la solución del mismo.

Antes de hacer los gráficos analicemos cual es el peor caso de este algoritmo.

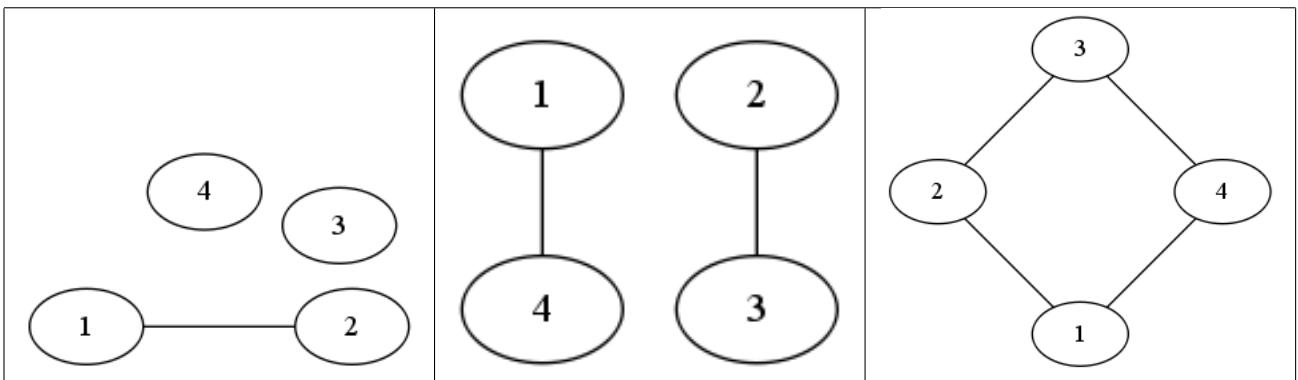
Para que el algoritmo tenga que volver para atrás cuando se intenta de agregar el último vértice del grafo, es decir, cuando se encuentra en la altura $n-1$, significa que hay un vértice que es adyacente a por lo menos un vértice de los que ya se encuentran en la solución.

Lo que nos dimos cuenta es que si todos los vértices tienen el mismo peso,

Para este algoritmo se pensaron 3 casos que pudiesen ser los peores:

- Cuando el grafo tiene un único eje.
- Cuando el grafo tiene varios ejes pero de forma tal que el vértice i es adyacente al vértice $n-i+1$.
- Cuando el grafo es un círculo, es decir, el vértice i es adyacente a los vértices $i+1$ y $i-1$

A continuación se presentan cada uno de los casos.



⁴Ver Apéndice, Cantidad de ramas en el algoritmo exacto

En el grafo que solo tiene un único eje, siempre que se intente de agregar el último vértice para que la solución contenga a todos los vértices del grafo, no va a ser posible ya que uno de los dos vértices incidentes del eje se encuentra en la solución.

Sin embargo, falta tener en cuenta la otra poda, es decir, la que tiene en cuenta el peso de los vértices. En caso de que todos los vértices tengan el mismo peso, entonces no importa cual de ellos sea adyacente entre sí, ya que en la altura $n-1$ las soluciones van a tener exactamente el mismo valor de la solución, y por lo tanto, esa poda no va a tener efecto.

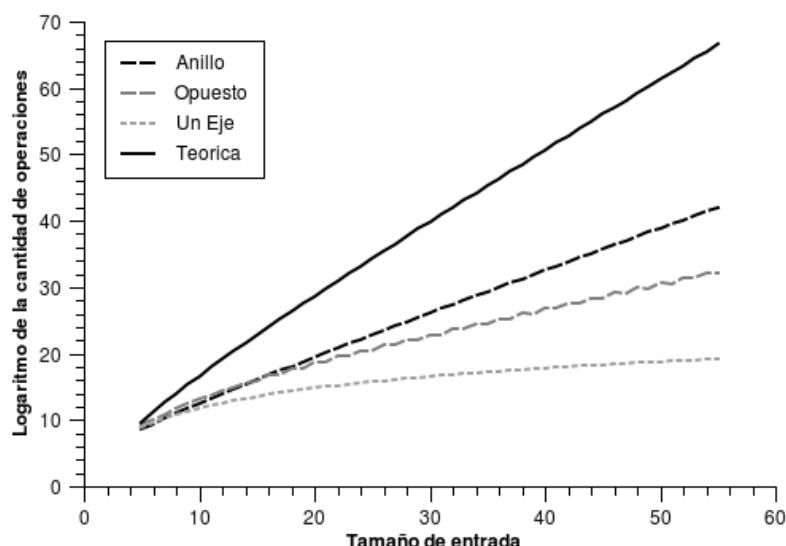
Luego, para el caso en donde el grafo tiene un único eje se planteo que todos los vértices tenga como su peso el el número del vértice por una constante⁵. Es decir, el vértice v_i va a tener peso $i*cte$.

Para el grafo en donde el vértice v_i es adyacente con el vértice v_{n-i+1} tambien fue necesario que todos los vértices tengan pesos diferentes. En caso de que no fuera así y todos los vértices tuvieran el mismo peso, entonces se encontraría rapido la solución ya que la misma permitiría tener la mitad de los vértices del grafo. Por lo tanto, ya cuando se revisa la primer rama del algoritmo se encuentra la solución óptima.

En cambio si el peso de los vértices fuese diferente, entonces eso podría no llegar a darse. Lo que se hizo es que todos los peso de los vértices sean su número multiplicado por una constante. De esta forma, no es cierto que la primer rama del grafo es la máxima, ya que la misma no tiene el vértice mas pesado del grafo.

Por último, para el grafo que es un anillo, también fue necesario hacer que todos los vértices tengan pesos diferentes por la misma razón que el grafo anterior.

A continuación se gráfica el logaritmo de la cantidad de operaciones hechas para estos tres tipos de grafos.



Ahora analizamos porque el mas cercano a la teórica es el grafo que es un anillo:

- Para el grafo que solo tiene un eje, se utilizo el eje (1,2). Esto hizo que la primer rama a ser tenida en cuenta sea la (1, 3, ...). Pero sabemos que la misma no era la solución máxima. Ya que el vértice 1 tiene peso 3, y el vértice v_2 tiene peso 6, entonces la solución máxima se alcanzaba cuando se analizaba la primer rama del segundo vértice. Sin embargo, para v_1 solo se verificaba la primer rama que es una solución valida, pero el resto de las ramas no son tenidas en cuenta ya que van a tener menos vértices van a pesar menos.
- Para el grafo cuando un vértice es adyacente al *opuesto*, hay dos cosas que cambian cuando se tiene en cuenta el algoritmo anterior:
 - Cuando se llega al segundo vértice el mismo no necesariamente es la mejor solución. En particular la mejor solución para este grafo esta compuesta por todos los vértices $(v_{n/2} \dots v_n)$. Por lo tanto, se tiene que hacer varias veces el algoritmo empezando en diferentes vértices.
 - Para cada vértice que se empieza la solución, no es cierto que la primer rama que es una solución valida es la mejor solución. Por lo tanto, se hacen también varias iteraciones sobre cada uno de los vértices.
- Por último, el caso donde el grafo es un anillo hace mas operaciones comparado con el anterior agrega muchos mas casos a tener en cuenta cuando se empieza por un vértice. Es cierto que la solución se encuentra cuando se empieza por v_2 o v_3 , dependiendo de si el grafo tiene una cantidad de vértices pares o impares. Sin embargo, el hecho de que haya mas vértices hace que existan mas posibilidades cuando se empieza por cada uno de los vértices.

⁵El valor de la constante fue 3

En la siguiente tabla figuran la cantidad de operaciones totales que hizo el algoritmo para cada uno de los archivos de entrada. Es importante notar que la cantidad de operaciones hechas para resolver los 55 grafos de tipo anillo son 568 veces la cantidad de operaciones hechas para resolver cuando el grafo es opuesto y 1.5 millones de veces la cantidad que cuando el grafo solo tiene un único eje.

Archivo Entrada	Anillo	Opuesto	Un Eje
Cantidad de operaciones	12820628510756	22544950700	8529019

Además, también se calcularon la cantidad de operaciones hechas para los grafos creados de forma azar. Al igual que el caso anterior se calculó la sumatoria para grafos de hasta 55 vértices ⁶.

Archivo Entrada	Random1	Random2	Random3
Cantidad de operaciones	5215994	5174449	5127932

Se nota claramente que la cantidad de operaciones son menores que la cantidad de operaciones hechas que cuando el grafo solo tenía un único eje.

3.4.1. Conclusiones

Por como fue creciendo la cantidad de operaciones hechas a medida que el grafo se le agregaban ejes pensábamos que si se le seguían agregando ejes la poda de los vértices que pueden llegar a ser agregados a la solución empezaría a acotar mas que los casos anteriores, y por lo tanto, no necesariamente se harían más casos que los planteados anteriormente.

Esto se ve claramente en un grafo que es completo, ya que al meter un vértice todo el resto de los vértices dejan de ser válidos.

Otro caso malo para el algoritmo hubiese sido un árbol, ya que tiene casi la misma cantidad de ejes que un anillo, y por lo tanto la poda de los vértices no afectaría tanto.

3.5. Análisis de la solución

El algoritmo exacto siempre da la solución óptima. Sin embargo, se puede notar que este algoritmo no trabaja por cada una de las componentes conexas del grafo. Esto es porque no es necesario separar el grafo en cada una de sus componentes conexas ya que el mismo analiza todas las soluciones posibles.

Por lo tanto, al calcular todas las posibles soluciones esto también calcula todas las posibles soluciones para cada componente conexa. Es posible que el algoritmo hubiese hecho menor cantidad de operaciones si se hubiese trabajado por cada componente conexa del mismo, ya que en cada iteración no tengo todos los vértices del grafo, sino que solo podría tener los vértices de esa componente conexa. Eso hace que el ciclo principal se repita una menor cantidad de veces y por lo tanto, se hagan menos operaciones.

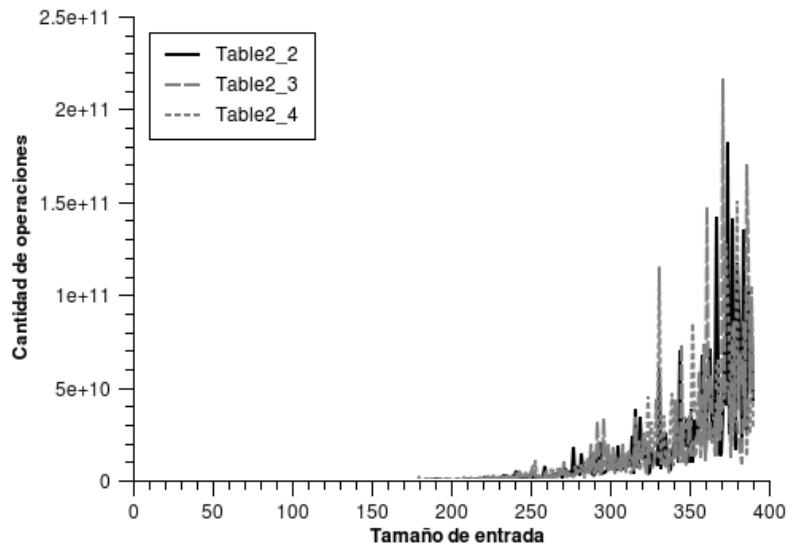
Sin embargo, no puede disminuir la complejidad ya que en el peor caso el grafo es conexo y se hacen la cantidad de operaciones planteadas anteriormente.

Por falta de tiempo, no se corrió este algoritmo para cada una de las variantes del archivo de entrada. Para el único caso donde este algoritmo fue ejecutado fue para los 3 archivos randoms⁷.

En el gráfico que figura a continuación se muestra la cantidad de operaciones hechas para resolver el algoritmo, para cada uno de estos archivos. A diferencia de los gráficos anteriores, en este se usaron grafos de hasta 390 vértices.

⁶Por más de que se habían calculado la solución exacta para grafos más grandes

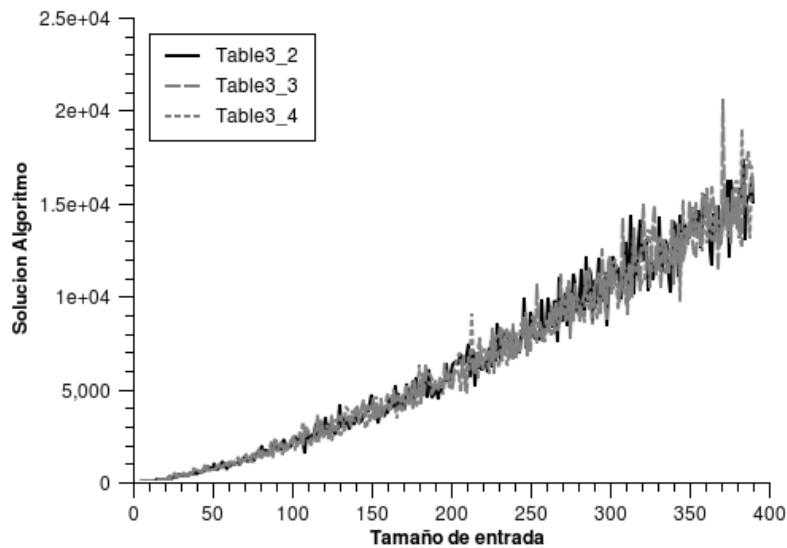
⁷Ver Heurísticas Y Algoritmos, Archivos de Entrada.



Aca es importante notar 2 cosas:

- Para que el gráfico fuese claro no fue necesario hacer el logaritmo de la cantidad de operaciones, como fue necesario en el gráfico del peor caso. Esto indica que la cantidad de operaciones es notoriamente menor.
- El crecimiento no es exponencial sino que es al azar. Esto se debe a que como el archivo de entrada fue totalmente generado al azar entonces es posible que en un grafo se tarde mucho en llegar a la solución óptima porque tiene pocos ejes. Sin embargo, en un grafo mas grande, es posible que se llegue a una solución mas rápido ya que el mismo tiene mas ejes, y por lo tanto, tiene que tener menos vértices a tener en cuenta.

También para estos tres archivos se graficó el valor de la solución.



De nuevo se puede notar que no es totalmente lineal ya que alterna un poco. Sin embargo, los picos no son tan sobresalientes como en el gráfico anterior.

4. Algoritmo Constructivo

4.1. Introducción

En esta familia de heurísticas, el cambio se debe a como se selecciona el nodo a agregar. Entre las diferentes formas de agregar vértices se nos ocurrieron las siguientes:

- Agregar el de mayor peso. En teoría esto debería de funcionar porque cuando el algoritmo termina, se tienen todos los vértices de mayor peso.
- Agregar los de menor grado. Esta opción salió de pensar que mientras menor sea el grado del vértice, entonces mayor sería la cantidad de vértices que tiene la solución, y por lo tanto, mayor sería el peso de la misma.
- Agregar los de mayor beneficio. Para esto nosotros definimos el beneficio como el peso del vértice sobre la cantidad de vértices adyacentes del mismo. Como el grafo no tiene porque ser conexo, entonces pueden existir vértices aislados. Por esto es que agregamos la condición de que si el vértice no tiene adyacentes, entonces el beneficio del mismo es el peso del mismo.

Para no tener que escribir el pseudocódigo 3 veces, lo que hacemos es definir una función llamada *valor* la cual devuelve el valor según el tipo de heurística que se está usando:

- Para el de mayor peso, el valor de un vértice devuelve el peso del mismo
- Para el de menor grado, el valor es el grado del vértice.
- Para los de mayor *beneficio*, el valor peso del vértice sobre el grado del mismo.

4.2. Pseudocódigo

NOTA: El pseudocódigo que figura a continuación es genérico para los 3 casos. La función *valor* que toma como parámetro un vértice es la que devuelve:

- El grado del vértice
- El peso del vértice
- El peso sobre el grado del vértice.

Además, el α y β que figuran en el pseudocódigo son las variables del GRASP que indican lo siguiente:

- El α indica cual es la diferencia que puede haber entre el valor del vértice que va a ser agregado y el vértice actual.
- El β indica la cantidad de vértices que van a ser tenidos en cuenta.

Algorithm 5 heuristicaConstructiva(grafo, alfa, beta)

```

Sea solución una solución vacía
Sea puedoSeguir ← True
while puedoSeguir do
    Sea nodoAAgregar ← obtenerNodoAAgregar(grafo, solución)
    if nodoAAgregar ∈ nodos(grafo) then
        Agregar nodoAAgregar a solución
    else
        puedoSeguir ← False
    end if
end while
end
```

Algorithm 6 obtenerNodoAAgregar(grafo, solución, alpha, beta)

```

Sea nodoAAgregar  $\leftarrow$  un vértice que no pertenece al grafo
Sea listaPosibles  $\leftarrow \emptyset$ 
for vértice  $\in$  vértices(grafo) do
    if esValido(solución, vértice) then
        Agregar (vértice, valor(vértice)) a listaPosibles
    end if
end for
if listaPosibles ==  $\emptyset$  then
    devolver nodoAAgregar
end if
Ordenar listaPosibles en forma decreciente o creciente dependiendo usando el valor del vértice.
if alpha == 100 then
    devolver el vértice de primero(listaPosibles)
else
    Sea listaRandom  $\leftarrow \emptyset$ 
    Sea mejorVértice  $\leftarrow$  primero(listaPosibles)
    while noPaseBeta(listaRandom, vértices(grafo), beta) y quedan elementos en listaPosibles do
        Sea vérticeValor el proximo valor de listaPosibles
        if cumpleElAlpha(valor(vérticeValor), valor(mejorVértice), alpha) then
            Agregar el vértice de vérticeValor a listaRandom
        end if
    end while
    devolver un valor cualquiera de listaRandom
end if
devolver nodoMayorPeso
end
```

Algorithm 7 noPaseBeta(*listaActual*, cantVértices, beta)

```

Sea maxCantNodos  $\leftarrow$  cantVértices * beta / 100
devolver len(listaActual)  $\leq$  maxCantNodos
end
```

Algorithm 8 cumpleElAlpha(*valorActual*, *mejorValor*, alpha)

```

Sea minimoPosible  $\leftarrow$  mejorValor * alpha / 100
devolver valorActual  $\geq$  minimoPosible
end
```

4.3. Detalles de implementación

En la implementación del código, hay dos grandes cambios:

- Al ordenar la lista: dado que el primer vértice de la lista es el más importante (ya sea porque es el que se va a devolver cuando *alpha* es 100 o porque es contra el que se compara para tener en cuenta el valor del *alpha*), entonces la lista *listaPosibles* se tiene que ordenar dependiendo del valor. Por ejemplo, si se tiene que seleccionar el vértice de menor grado entonces la lista tiene que estar ordenada de forma creciente, pero si se tiene que seleccionar el de mayor grado entonces la lista se tiene que ordenar de forma decreciente. Por lo tanto, la lista siempre se ordena de forma creciente, y si es necesario ordenarla de forma decreciente entonces simplemente se la da vuelta. Esto no cambia el orden final, ya que el hecho de ordenar la lista tiene mayor complejidad que el hecho de darla vuelta. No se tienen en cuenta las operaciones adicionales de darla vuelta, ya que se podría hacer un algoritmo que ordene de forma creciente o decreciente dependiendo de un booleano, y que ambos tengan la misma complejidad.
- Al verificar si un valor cumple con el α . Aquí, al igual que el caso anterior, hay que tener en cuenta si *mejorValor* representa un máximo o un mínimo. Va a ser un máximo cuando el algoritmo usado es el de mayor peso o beneficio, pero va a ser un mínimo cuando se usa el de menor grado. Por lo tanto, si *mejorValor* es un máximo se usa el algoritmo planteado anteriormente. Pero si es un mínimo, se usa el siguiente algoritmo que tiene la misma complejidad que la original:

```
Algorithm 9 cumpleElAlpha(valorActual, mejorValor, alpha)
    Sea  $maximoPosible \leftarrow mejorValor * (alpha+100) / 100$ 
    devolver  $valorActual \leq maximoPosible$ 
end
```

4.4. Complejidad

Primero veamos el orden de la función *obtenerNodoAAgregar*. En esta función lo que se hace es revisar todos los vértices del grafo y verificar si pueden ser agregados a la solución. Sabemos que el hecho de verificar si un vértice puede ser agregado a la solución es $O(1)$ ⁸, al igual que agregar un vértice a la solución. Por lo tanto, el primer ciclo tiene complejidad $O(n)$ donde n es la cantidad de vértices del grafo.

Cuando la solución se encuentra vacía, entonces cualquier vértice puede ser agregado a la solución. Luego, la lista *listaPosibles* va a tener todos los vértices del grafo. Por lo tanto, la complejidad de ordenar la misma es $O(n * log(n))$.

Por último queda analizar que es lo que sucede cuando el *alpha* no es 100. En ese caso, se tiene que seleccionar un vértice al azar. El orden de las funciones *cumpleElAlpha* y *noPaseBeta* es $O(1)$. Como se itera por todos los vértices que se encuentran en *listaPosibles*, entonces el ciclo tiene orden $O(n)$. Al momento de seleccionar una vértice al azar de *listaRamdon*, la misma puede llegar a tener todos los vértices del grafo, y por lo tanto eso tiene una complejidad de $O(n)$.

Por lo tanto, la complejidad final de *obtenerNodoAAgregar* es:

$$O(n) + O(n * log(n)) + O(n) + O(n) \in O(n * log(n))$$

Ahora vemos la complejidad del algoritmo principal. Dentro del ciclo, se hacen dos operaciones:

- Buscar el vértice a agregar. Como se demostró anteriormente, esto tiene complejidad $O(n * log(n))$.
- Verificar si el vértice a agregar pertenece al grafo, o si en verdad ya no se pueden agregar mas vértices a la solución. Esto se puede hacer en $O(1)$ ya que el vértice no es mas que un número entero, y en el grafo, se guardan la cantidad de vértices del mismo.
- Agregar un vértice a la solución. Como se comentó anteriormente la complejidad de esto es $O(n)$.

El ciclo principal, en caso de que puedan ser agregado todos los vértices a la solución se ejecuta n veces, donde n es la cantidad de vértices que tiene el grafo. Luego, la complejidad del algoritmo es:

$$n * (O(1) + O(n * log(n)) + O(n)) \in O(n^2 * log(n))$$

Ahora vemos cual es el tamaño de entrada para este algoritmo:

$$\begin{aligned} T &= \log(n) + \sum_{i=1}^n \log(peso(i)) + \sum_{i=1}^n \sum_{v \in ad(i)} \log(v) \\ T &\geq 1 + 1 + \sum_{i=1}^n \log(peso(i)) + \sum_{i=1}^n \sum_{v \in ad(i)} \log(v) \\ T &\geq \sum_{i=1}^n \log(peso(i)) \\ T &\geq \sum_{i=1}^n 1 \\ T &\geq n \end{aligned}$$

Por lo tanto, la complejidad en función del tamaño de entrada nos queda:

$$O(T^2 * log(T))$$

y la misma es polinómica

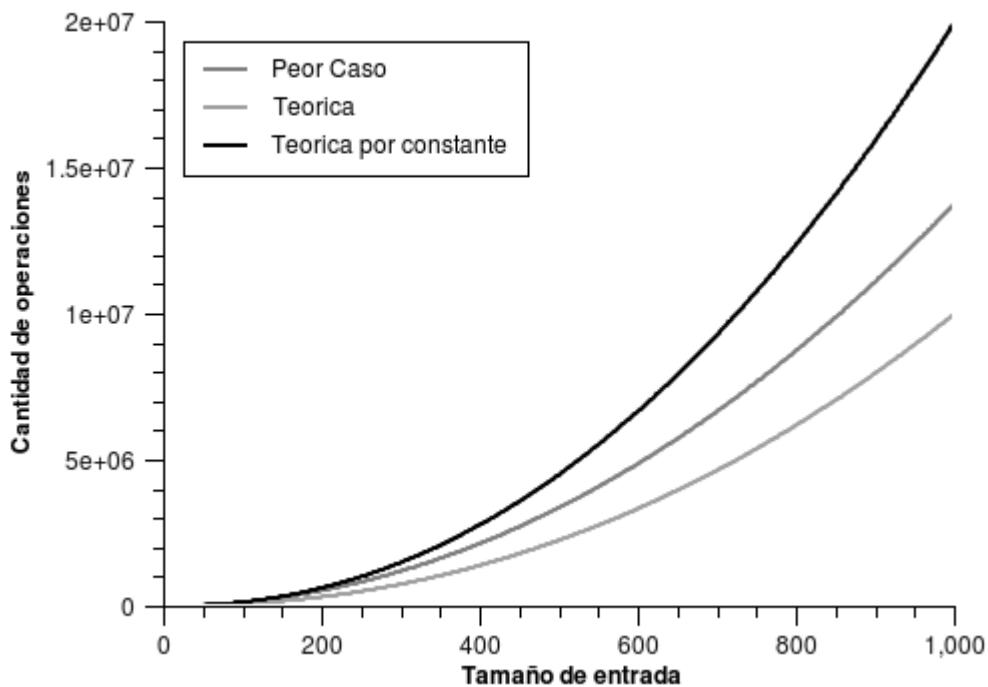
⁸Ver Heurísticas y algoritmos, Estructuras Usadas, Solución.

4.5. Gráficos de tiempos

Acá vamos a ver como se comporta el algoritmo en función del tiempo sin tener en cuenta la calidad de la solución. Antes de hacer los gráficos de la cantidad de operaciones, vemos cual es el peor caso para este tipo de algoritmo.

Si el grafo es un conjunto de vértices aislados, es decir, no tiene ejes, no importa cual de las tres opciones se elija para agregar un vértice a la solución, la cantidad de veces que se hace el ciclo principal es el máximo posible. Además, siempre que se llama a *obtenerNodoAAgregar*, también se va a hacer la máxima cantidad de operaciones ya que todos los vértices que no fueron agregados anteriormente a la solución tienen que ser tenidos en cuenta.

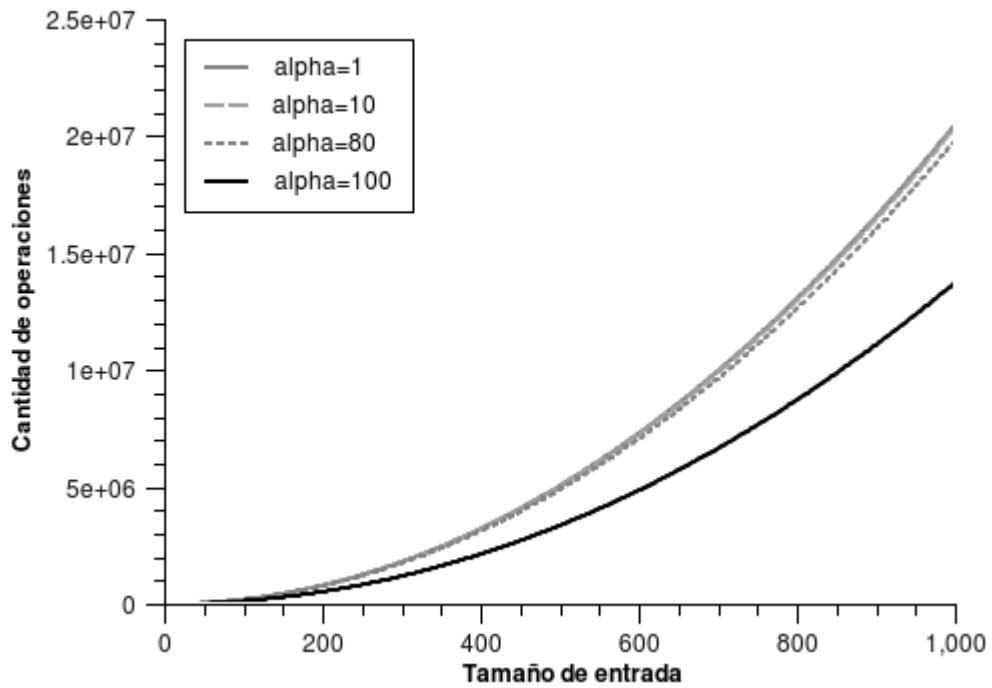
En el gráfico que figura a continuación, se grafica la cantidad de operaciones que hace el algoritmo cuando el grafo no tiene ningún eje. Como todas las variantes de los algoritmos tienen el mismo orden se puede seleccionar cualquiera.



En el grafo anterior se usaron grafos de hasta 1000 vértices. Las operaciones teóricas calculadas quedan por debajo de la cantidad de operaciones, pero la constante por la que se multiplico la cantidad de operaciones teóricas es 2.

El gráfico anterior fue hecho sin tener en cuenta el valor de α y β . Si se tienen en cuenta estos dos valores cuando se ejecuta el algoritmo, la complejidad final del algoritmo no cambia, pero se suman una cantidad de operaciones. Ahora veremos como es que se comporta el algoritmo cuando se tienen en cuenta los valores de α y β .

En el gráfico que figura a continuación se dejó fijo el valor de β pero se varió el α . El archivo de entrada en este caso fue el peor caso planteado anteriormente. El β se lo dejó fijo en 100 de forma tal que todos los valores que cumplan con el alpha comentado



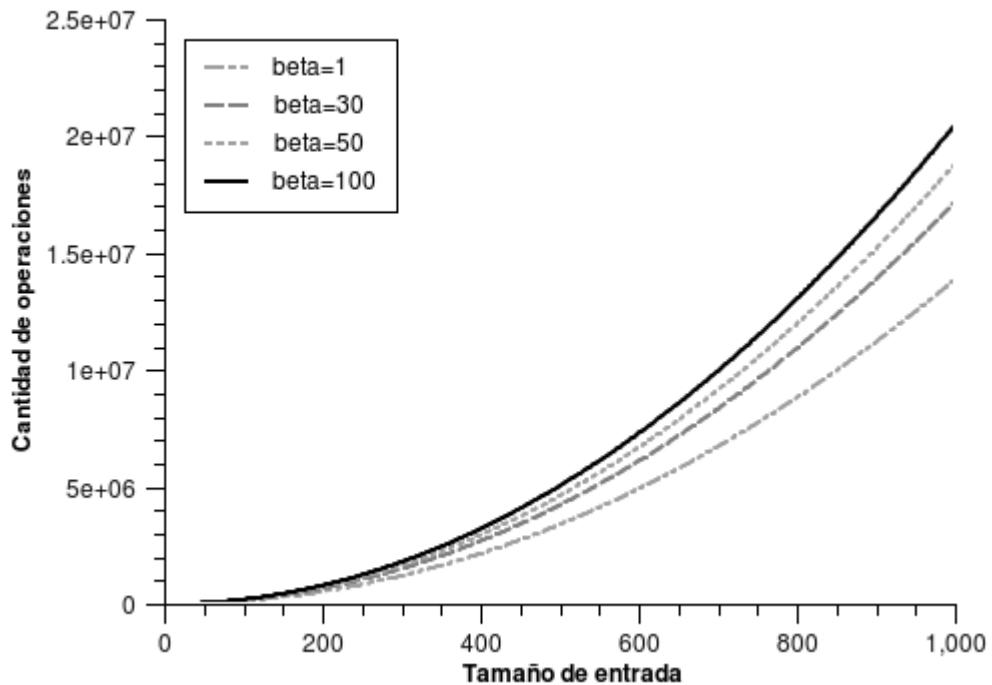
Es interesante notar que la cantidad de operaciones varía cuando el α deja de ser 100, es decir, cuando se tiene que seleccionar un valor al azar. Sin embargo, entre los diferentes valores menores que 100 no hay mucha diferencia entre la cantidad de operaciones.

Para notar más claramente la cantidad de operaciones hechas, lo que se hizo es sumar la cantidad de operaciones hechas por cada variante para obtener todas las soluciones, es decir, sumar la cantidad de operaciones de todos los grafos por cada variante.

Valor de α	1	10	30	50	80	90	100
Cant. operaciones	6766912103	66726006197	6653700453	6600734313	6546843791	6534257951	4517642079

Como se puede notar, la cantidad de operaciones se reduce a medida que el valor de α crece. Esto tiene sentido ya que a medida que el valor de α crece menos valores tienen la posibilidad de ser insertados en la lista *listaRandom*, y por lo tanto, se tiene una lista con menor cantidad de elementos al elegir el vértice al azar.

Ahora cambiamos el valor de β , pero dejamos el α fijo, también con el mismo archivo de entrada. El valor de α fue fijado en 1, para que el límite sea β y no el α al momento de agregar vértices a la lista *listaRandom*.



A diferencia de cuando se cambio el α , en este caso mientras mayor sea el valor de β mayor va a ser la cantidad de operaciones. Esto tiene sentido ya que mientras mayor sea el beta, mayor va a ser la cantidad de vértices que van a poder se agregados a la lista *listaRandom*, y también es mayor la cantidad de veces que se repite el ciclo principal.

De nuevo, se calculo la cantidad de operaciones sumadas para cuando se variaba el β obteniendo la siguiente tabla:

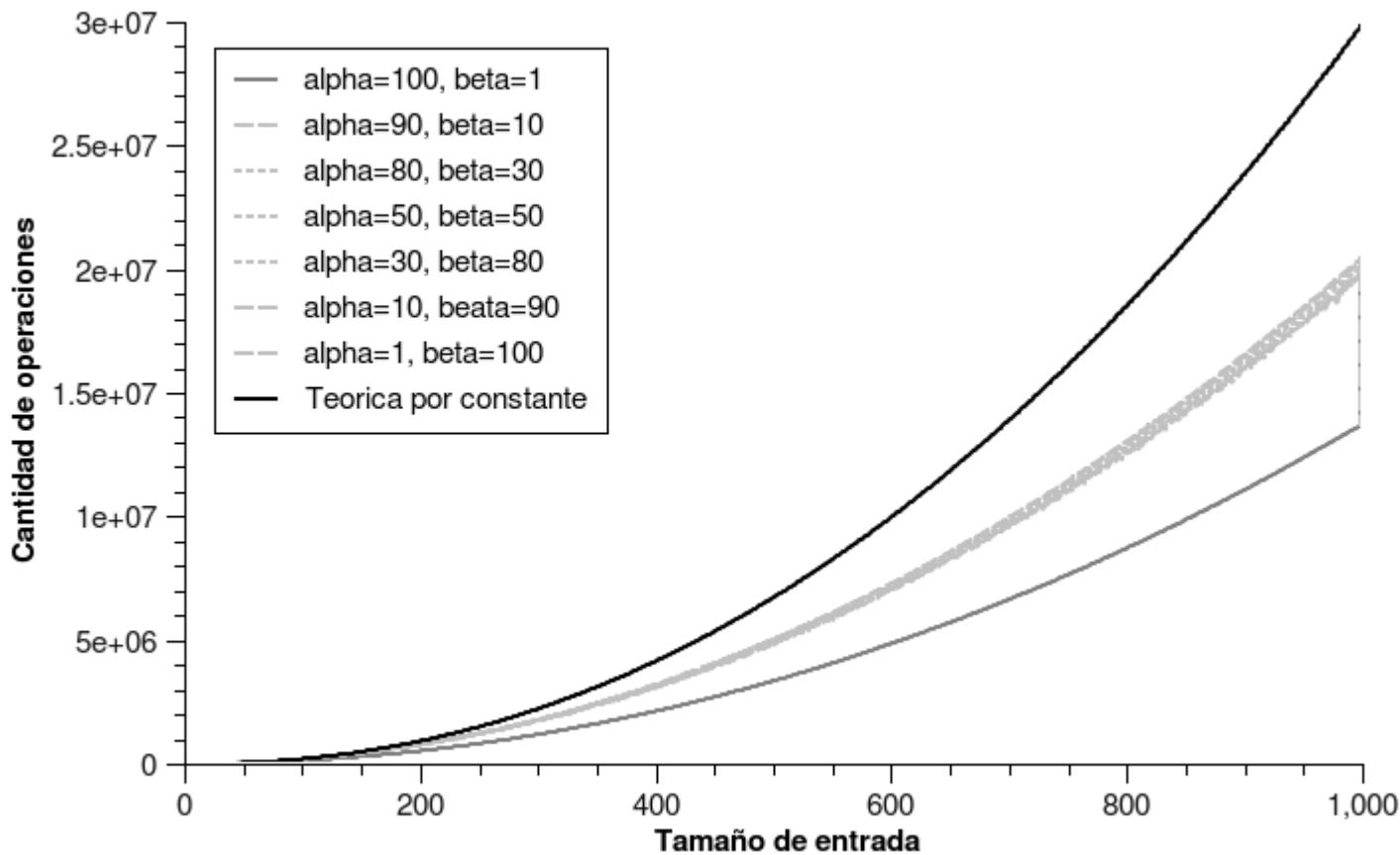
β	1	10	30	50	80	90	100
Operaciones	4569392024	4951780062	5669671152	6208427774	6678179362	6744949289	6766984702

Por lo tanto, se puede llegar a la conclusión de que mientras menor es α y mayor es β mayor es la cantidad de operaciones que se hacen. A continuación se grafican casos donde se vario tanto el α con el β .

Para esto, se eligieron los siguientes valores:

α	1	10	30	50	80	90	100
α	100	90	80	50	30	10	1

A continuación se gráfica la cantidad de operaciones hechas por cada uno de los algoritmos, y se la compara contra la complejidad teórica por una constante.



En este caso la constante que multiplico a la función teórica es mayor⁹ que en el gráfico donde no se tuvo en cuenta los valores de α y β . Esto se debe a que el hecho de buscar el valor al azar suma en la cantidad de operaciones aunque no varia en la complejidad final del algoritmo.

Como en el gráfico no se pueden distinguir muy claramente, la tabla a continuación muestra la cantidad de operaciones totales hechas para cada caso:

Valor $\alpha - \beta$	100-1	90-10	80-30	50-50	30-80	10-90	1-100
Cant Operaciones	4537377040	6531152023	6566751597	6603968100	6674000483	6744450567	6780575901

Como se puede notar a medida que crece el α y decrece β mayor es la cantidad de operaciones. Sin embargo, hay un gran salto de cuando α pasa de 100 a 90. Esto se debe a que en el caso cuando α es 100 no se busca el valor random, y por lo tanto no se hace la iteración sobre la lista de *listaPosibles*.

4.5.1. Conclusiones

Una optimización que pudo ser agregada al algoritmo teniendo en cuenta que los valores de *listaPosibles* están ordenados es que si el *vérticeValor* actual no cumple con el α esperado entonces sabemos que ningún otro elemento de la lista lo va a cumplir.

Eso evita operaciones cuando el β es grande pero el α es chico, ya que en esos casos pocos vértices son agregados a la lista *listaRandoms*, y por lo tanto se están haciendo iteraciones sobre los elementos de la lista *listaPosibles* que podrían haberse evitado.

4.6. Análisis de la solución

En todas las variantes de la heurística golosa, si el grafo no es conexo entonces es lo mismo aplicar la heurística para cada una de las componentes conexas del mismo que para todo el grafo entero. La razón de esto se que la heurística busca los que cumplen cierto valor, y no le importa en que componente conexa se encuentra la misma. Por lo tanto, a continuación se va a analizar el grafo como si tuviese una única componente conexa.

⁹En este caso la constante es 3 mientras que el caso anterior fue 2.

Uno pensaría que hacer el algoritmo constructivo goloso por cada una de las componentes conexas puede llegar a una mejor solución, pero esto no es cierto. Por ejemplo, si cuando se agrega un vértice cuando el algoritmo trabaja por cada una de las componentes conexas del grafo, entonces el mismo vértice va a ser agregado cuando se trabaje con el grafo entero porque:

- En ambos casos el vértice cumplió el criterio para ser seleccionado y agregado.
- En ambos casos el vértice puede formar parte de la solución. Si el mismo podía ser agregado teniendo en cuenta la componente conexa, entonces puede ser agregado cuando se tiene en cuenta todo el grafo ya que no puede haber un vértice en la solución que se adyacente al mismo. Esto sucede porque en caso contrario, el vértice no puede ser agregado cuando se tiene en cuenta la componente conexa.

4.6.1. Soluciones correctas y erróneas

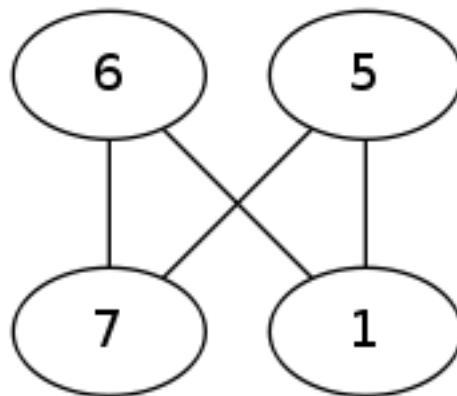
En la siguiente sección vamos a analizar cuando es que falla cada uno de los casos del algoritmo goloso, y cuando es que da la solución exacta.

Mayor Peso Sabemos que esta heurística no va a fallar en los siguientes casos:

- Grafos completos: en este caso, la solución ideal va a ser un único vértice, y el mismo va a ser el de mayor peso. Esto se deduce de que todos los vértices son adyacentes entre sí, y por lo tanto solo puede haber uno en la solución. Por lo tanto, el algoritmo de mayor peso va a meter el vértice de mayor peso, y al intentar de insertar el segundo vértice el algoritmo va a terminar.
- Bipartitos completos. Sea un $K_{p,q}$ un grafo bipartito completo. Si la sumatoria de los pesos de los vértices que pertenecen al conjunto p es mayor que la sumatoria de los pesos de los vértices que pertenecen al conjunto q , y el vértice mas pesado se encuentra en p , entonces el algoritmo da la solución exacta. Sucede lo mismo si se da vuelta p y q . Esto sucede porque al ser un grafo bipartito, si se selecciona un vértice del conjunto p , entonces no se va a poder seleccionar ningún vértice del conjunto q . Pero como los vértices del conjunto p pesan mas, entonces ellos son la solución óptima.

Ahora veamos casos en los que este algoritmo falla:

- Árboles. El algoritmo puede llegar a fallar en árboles donde dado un vértice padre y sus hijos, el vértice padre pesa mas que todos sus hijos, pero al mismo tiempo, la suma de los pesos de todos los hijos es mayor a la suma del vértice. Por lo tanto, de ser posible, sería mejor poner todos los hijos que los vértices padres de mayor peso. Por ejemplo, estos casos se pueden ver muy fácilmente si los hijos son hojas.
- Bipartito completos: va a fallar cuando el vértice mas pesado pertenece al conjunto q , pero la suma de los vértices de p es mayor que la suma de los vértices de q . Esto es porque al estar en q el vértice mas pesado, entonces va a devolver como la solución todos los vértices del conjunto q ya que en ese conjunto va a estar el primer elemento que va a haber seleccionado.



En este caso, la solución dada por el algoritmo es (7,1) cuando la solución ideal es (6,5).

Mayor Beneficio Sabemos que este algoritmo no va a fallar en los siguientes casos:

- En los grafos completos, al igual que el de mayor peso, van a dar la solución exacta. Esto sucede porque como todos los vértices tienen el mismo grado, el de mayor peso también va a ser el de mayor beneficio.
- Bipartitos completos en el caso de Mayor Grado.

Sabemos que este algoritmo va a fallar cuando:

- Bipartitos completos: Si se tiene un grafo bipartito completo $K_{p,q}$ y el peso de los vértices de p es menor que los vértices de los pesos de q , y pero el vértice de mayor beneficio se encuentra en p .
- Árboles: es similar al caso de mayor peso, pero la diferencia se encuentra en que el beneficio del padre es mayor al beneficio de todos los hijos pero la suma del beneficio de todos los hijos es mayor que el beneficio del padre.

Menor Grado La idea de este criterio de inserción para poder meter una gran cantidad de vértices en la solución, y por lo tanto, como tengo bastantes vértices se supone que la solución tiene que ser buena.

Sabemos que el mismo va a fallar cuando:

- Grafos completos: en estos casos solo se puede meter un único vértice. Por lo tanto, todos los vértices tienen el mismo grado, y entonces no esta definido cual es que se mete (se selecciona uno random). Por lo que puede ser que el algoritmo de la solución exacta, pero tiene mas probabilidades de que no sea así.
- Bipartitos Completos. Si usamos un grafo bipartito como el caso en el cual el de mayor grado no falla, y suponemos que $p > q$, entonces el algoritmo va a seleccionar un vértice del conjunto p , ya que estos van a tener menor grado. Pero si la solución era los vértices que estaban en q , entonces sabemos que el algoritmo falla.

Aun asi, sabemos que no va a fallar cuando:

- Si el grafo con al menos 3 vértices y sea v_1 un vértice adyacente a todo el resto de los vértices del grafo, pero ninguno es adyacente entre sí. Si la suma de los vértices que no son v_1 no es menor, entonces el algoritmo da la solución correcta porque selecciona a todos los vértices que no son v_1 porque:
 - No son adyacentes entre sí.
 - Tienen todos el mismo grado 1

4.6.2. Ejecuciones

En este caso si pudimos correr el algoritmo para cada uno de los casos de archivos planteados.

Empecemos por los archivos donde solo se varia la cantidad de vértices por cada componente conexa, dejando el peso y el grado de los vértices fijos. En la tabla que figura a continuación figura la cantidad de veces que el algoritmo dio la mejor solución comparado contra los otros dos casos.

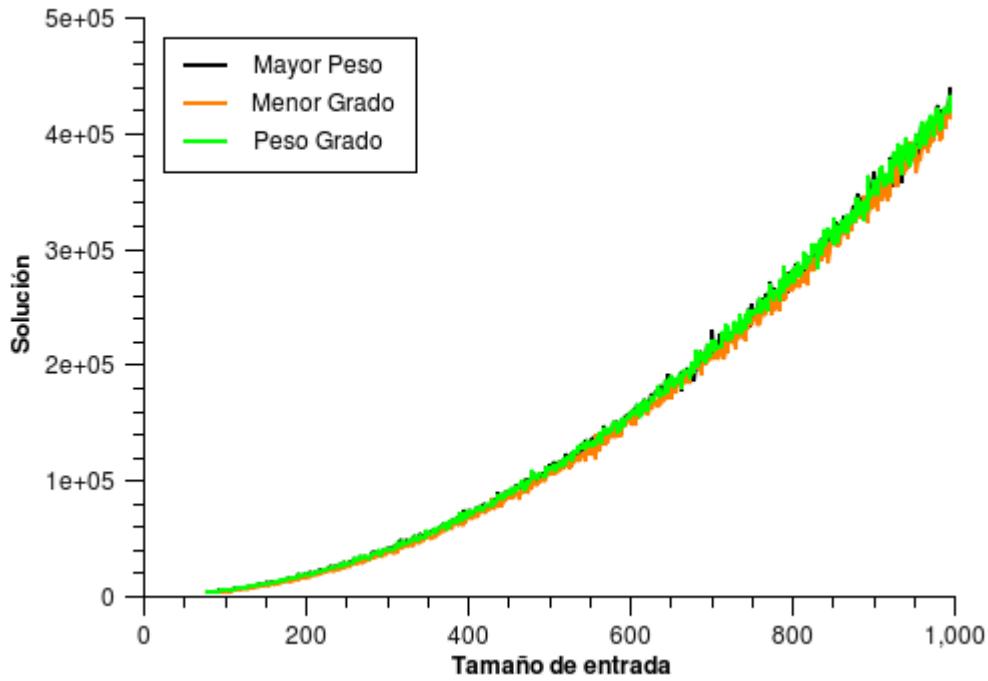
Algoritmo \ Cant Vértices	10	20	50	70
Mayor Peso	128	289	446	495
Peso Grado	444	434	442	489
Menor Grado	348	203	68	23

Aunque resulte raro, la cantidad de veces en que la variante de *Mayor Peso* dio la mejor solución o una solución igual a la de los otros dos variantes crece, mientras que la de *Peso Grado* se mantiene constante. Por último, la variante de *Menor Grado* decrece la cantidad de veces que dio la mejor solución.

En un principio uno podría llegar a pensar que *Mayor Grado* da mejor solución que *Peso Grado*. Sin embargo esto es un error, ya que ambas soluciones son parecidas. En los grafos analizados anteriormente, todos los vértices tienen el mismo peso, y prácticamente el mismo grado. Por lo tanto, la solución creada por *Mayor Grado* son soluciones que eligen vértices que pueden ser agregados, y este puede ser cualquiera del grafo dado que todos tienen el mismo peso.

Como los archivos de entrada fueron creados con un problema de que es posible que los últimos vértices del grafo no tengan el mismo grado que los vértices anteriores, entonces lo normal es que *Peso Grado* seleccione uno de esos vértices, y luego agregue vértices. Sin embargo, la variación entre $\frac{\text{peso}}{\text{grado}}$ de cada uno de los vértices es realmente poca.

En el gráfico que se encuentra a continuación graficamos como es que dio la solución para cada uno de las variantes.

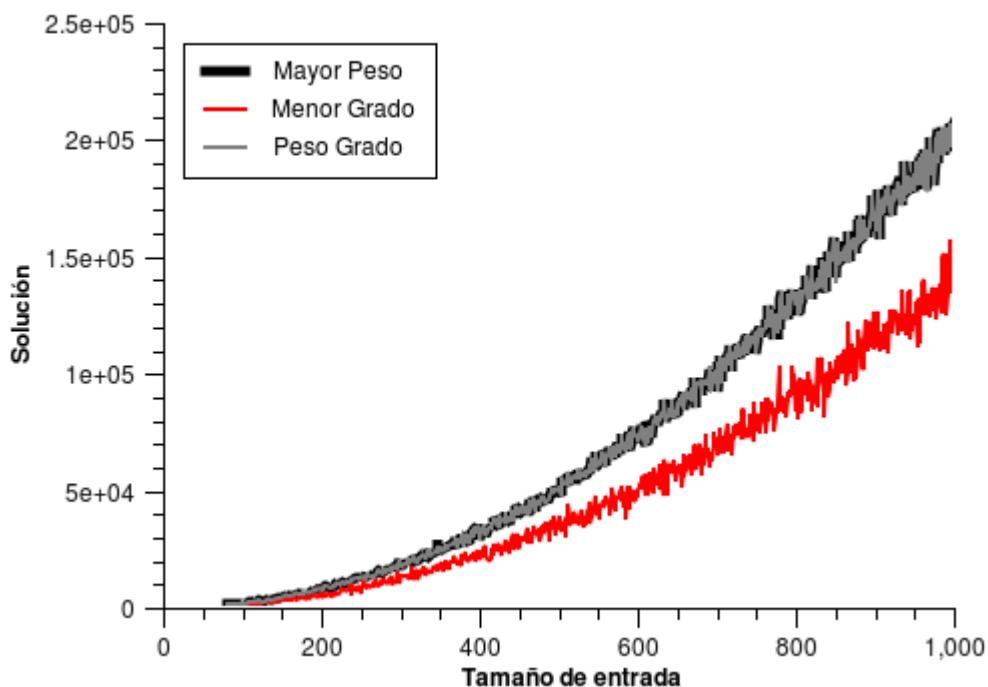


Como era de esperarse todos los algoritmos dieron soluciones realmente similares.

Esto tambien va a suceder con las diferentes variantes de los archivos de entrada porque en todos ellos, para un mismo archivo, todos los vértices tienen el mismo peso o el mismo grado. Sin embargo, hay dos clases de archivos para los cuales esto no es cierto:

- Variar Peso Grado, ya que en este archivo, el peso de los vértices es diferente dentro de un mismo grafo, por mas de que el grado es constante.
- Variar Peso Conexo, es similar al anterior.
- Variar todo

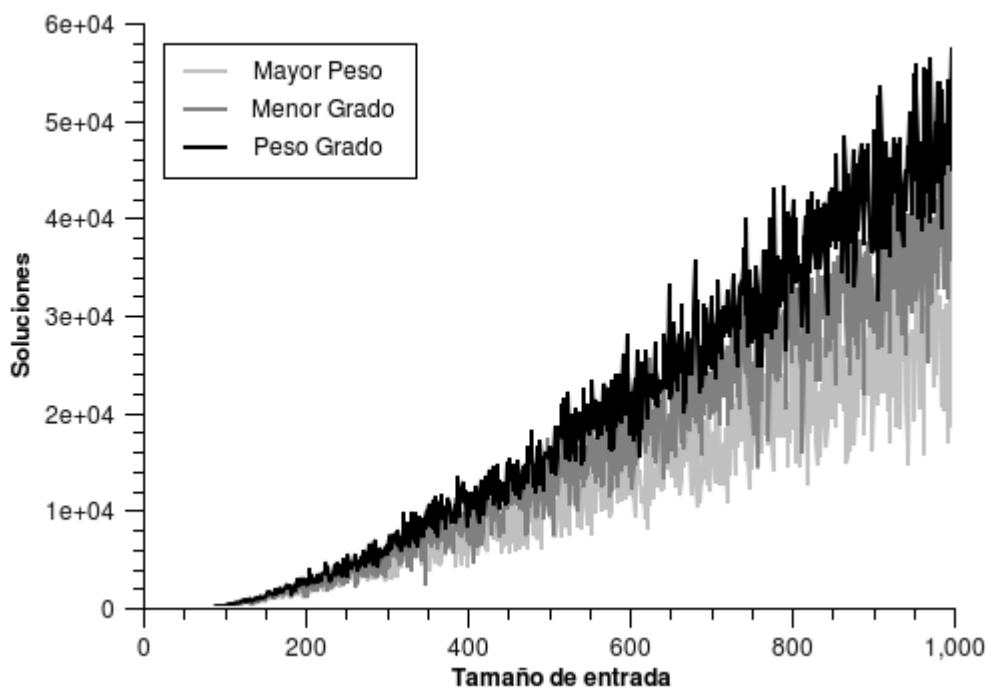
Veamos como queda la tabla en el caso cuando se varia el peso de los vértices, pero el grado se deja constante. Si graficamos la solución en función del tamaño de entrada, obtenemos lo siguiente:



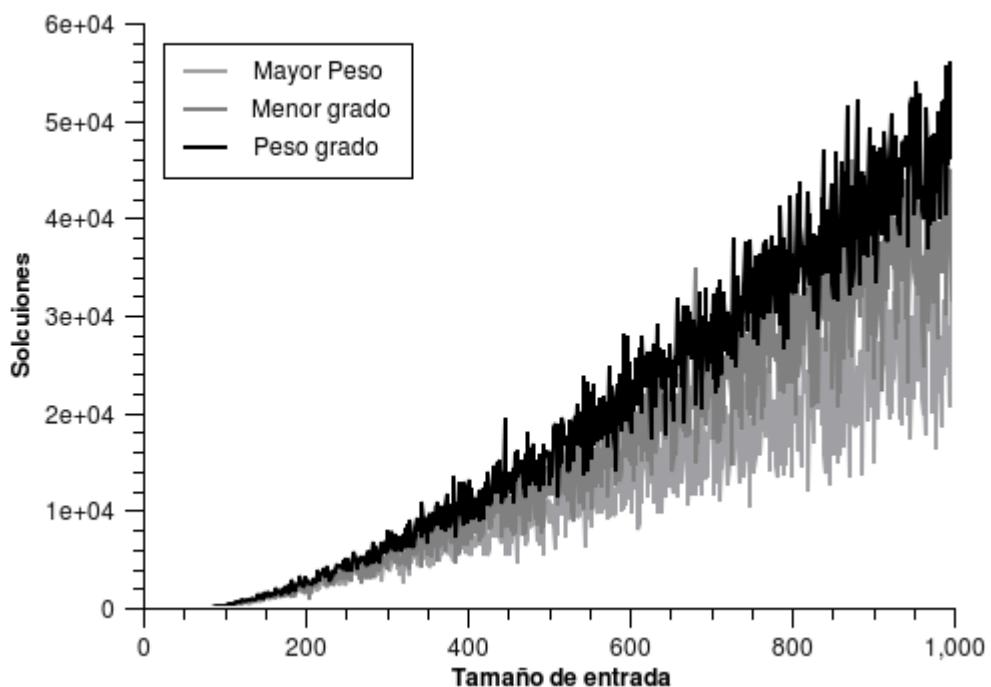
En este caso, tanto *Peso Grado* como *Mayor Peso*, tienen soluciones similares, pero *Menor Grado* tiene soluciones peores que los casos planteados anteriormente. El gráfico anterior fue hecho cuando el peso del vértice v_i es i^2 . Sin embargo, en estos casos es constante el grado. Al ser constante el grado para cada uno de los vértices del grafo, entonces las dos mejores variantes se van a comportar de forma similar, ya que el vértice de *Mayor Peso*, sea también el que tiene mayor $\frac{\text{peso}}{\text{grado}}$.

Por lo tanto, descartamos estas variante del archivo en donde para un mismo grado, todos los vértices tienen el mismo grado, ya que esto hace que los dos algoritmos se comporten de forma similar. Luego, nos fijamos como es que se comportan los algoritmos en los archivos generados al azar, ya que en estos, tanto el peso como el grado de los vértices cambia.

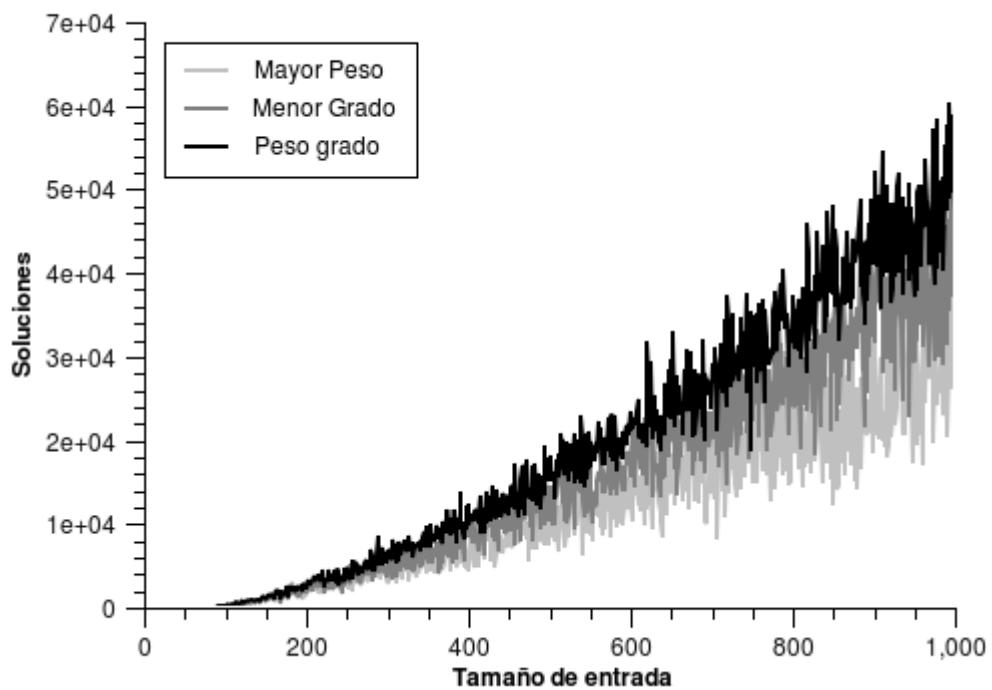
Para el primer archivo generado al azar se obtiene el siguiente gráfico:



Para el segundo archivo, se obtiene el siguiente:



Por ultimo, para el tercero, se obtiene:

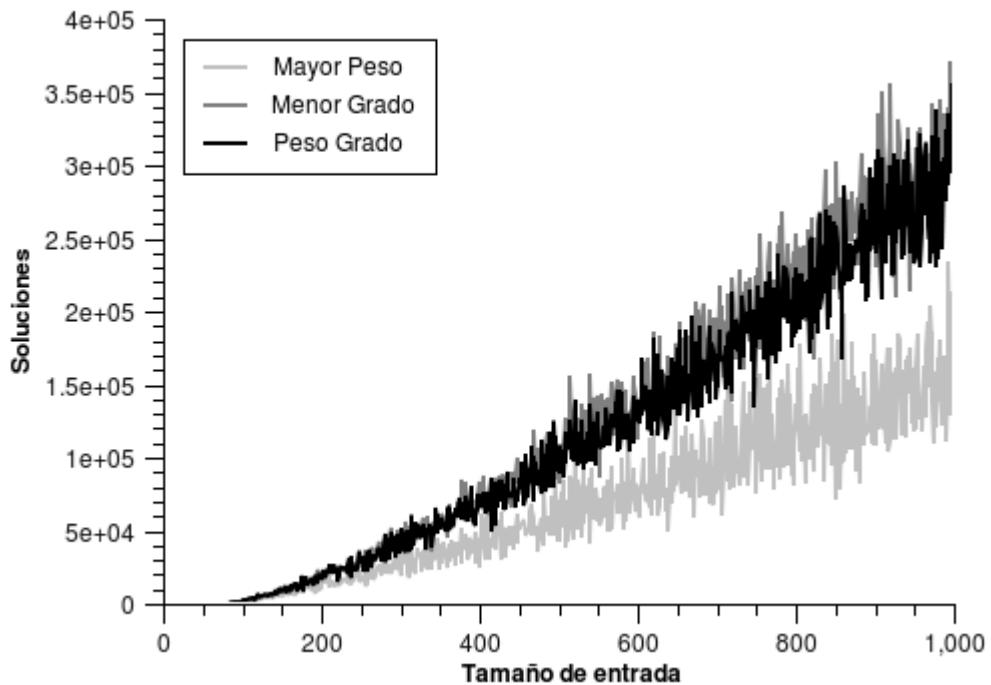


Como se puede ver, en general la variante *Peso Grado* da la mejor solución, y sin embargo, la segunda variante es la de *Menor Grado*. Sin embargo, si contamos cuantas veces dio la mejor solución cada uno de los algoritmos, obtenemos la siguiente tabla:

Algoritmo \ Archivo Random	1	2	3
Mayor Peso	48	43	51
Peso Grado	844	855	842
Menor Grado	47	48	51

4.6.3. Conclusiones

Por clara mayoría el criterio de agregar el de mayor beneficio demostró ser el mejor en los tres casos. Sin embargo, vemos la cantidad de operaciones que realiza este algoritmo contra el que realizan los otros dos:



Eso es la cantidad de operaciones que se realizaron para el tercer archivo. Como se puede ver, la cantidad de operaciones de *Peso Grado* es prácticamente el doble que *Mayor Grado*, pero sin embargo, es menor que *Menor Grado*.

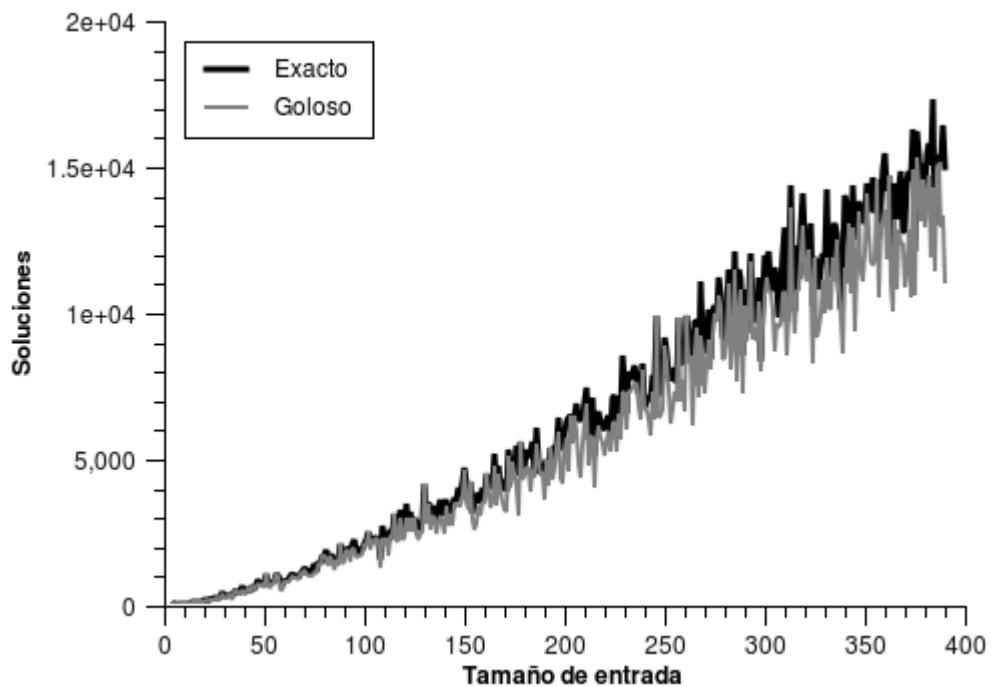
Luego, teniendo en cuenta que la cantidad de operaciones comparadas contra los casos anteriores no es demasiado diferente, pero si lo es la *calidad* de la solución entonces considero que realmente es la mejor de las 3 variantes.

Tiene sentido que la variante *Peso Grado* sea el que de la mejor solución ya este tipo de heurística tiene en cuenta el *costo* de meter un vértice en la solución. Costo refiriéndose al hecho de impedir que una cierta cantidad de vértices no puedan formar parte de la solución; y no en el sentido de la cantidad de operaciones

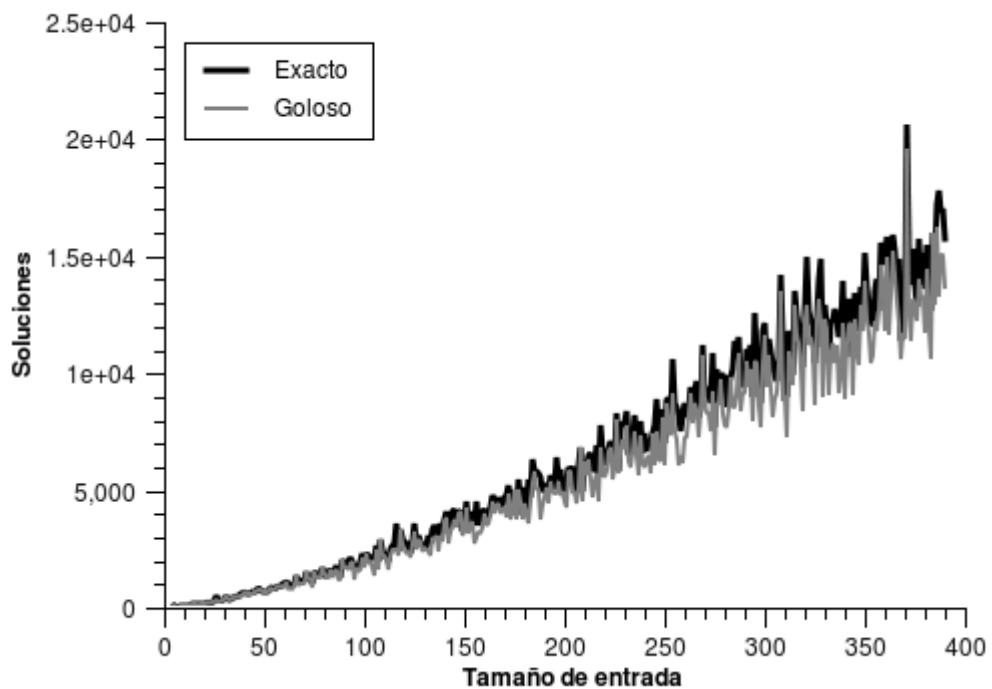
4.7. Comparaciones

El único algoritmo ejecutado antes que este para poder comparar es el algoritmo exacto. Por lo tanto, compramos las soluciones de la variante *Peso Grado* del algoritmo goloso contra la solución exacta. Estas comparaciones están hechas sobre los 3 archivos randoms, ya que esos fueron los mayores casos para los cuales se corrió el algoritmo exacto.

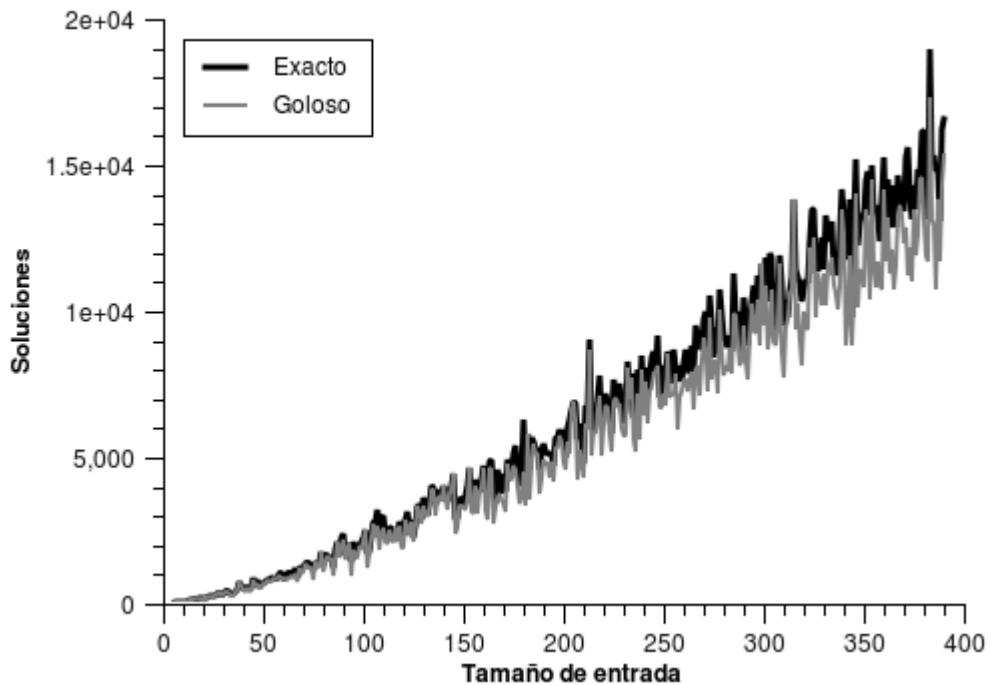
Para el primer archivo:



Para el segundo archivo:



Para el tercer archivo de entrada random:



Como se puede notar en los tres casos la solución del algoritmo exacto fue similar a la solución del algoritmo goloso. Si para cada archivo, sumamos el peso de todas las soluciones gráficas para cada uno de los algoritmos, obtenemos la siguiente tabla:

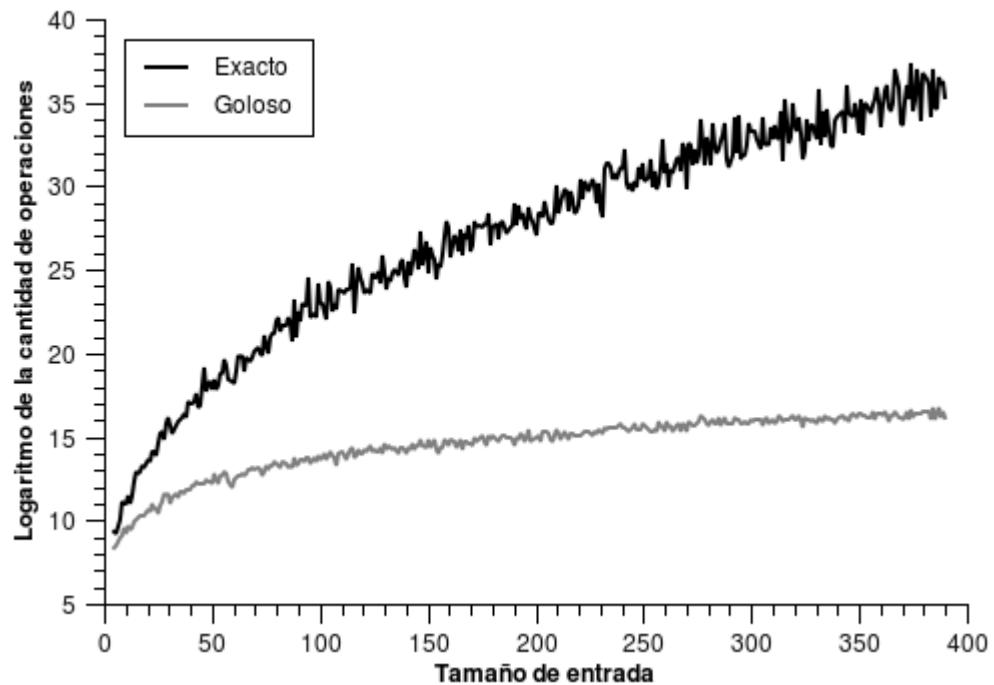
Algoritmo \ Archivo Random	1	2	3
Exacto	2484885	2478760	2453158
Goloso	2235806	2225758	2197002

Notar que aunque el constructivo se había corrido para grafos de hasta 1000 vértices, solo se hizo la cuenta para grafos de hasta 390 vértices, dado que ese fue el máximo tamaño de entrada usado por el algoritmo exacto.

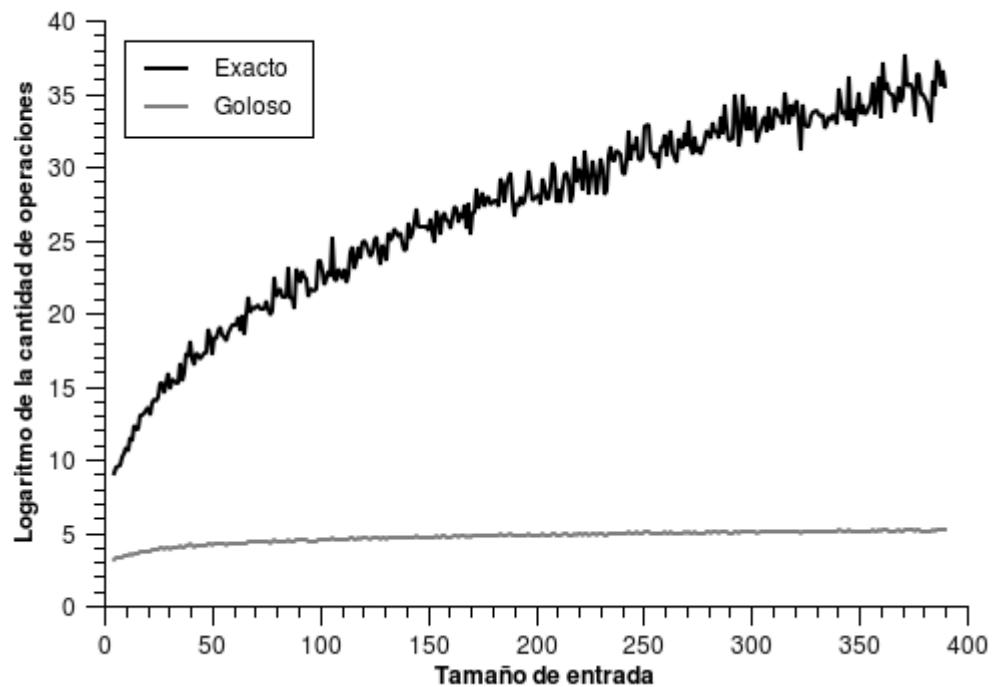
Como se puede notar en los tres casos, la diferencia entre los algoritmos es el 11% de la solución exacta. En un principio, uno podría llegar a considerar que esa diferencia es grande, mas si se tiene en cuenta que solo se usaron grafos de hasta 390 vértices.

Ahora comparamos la cantidad de operaciones hechas entre el algoritmo goloso y exacto en cada uno de los casos. Es importante que en todos los casos se hizo el logaritmo de la cantidad de operaciones ya que el algoritmo exacto es exponencial.

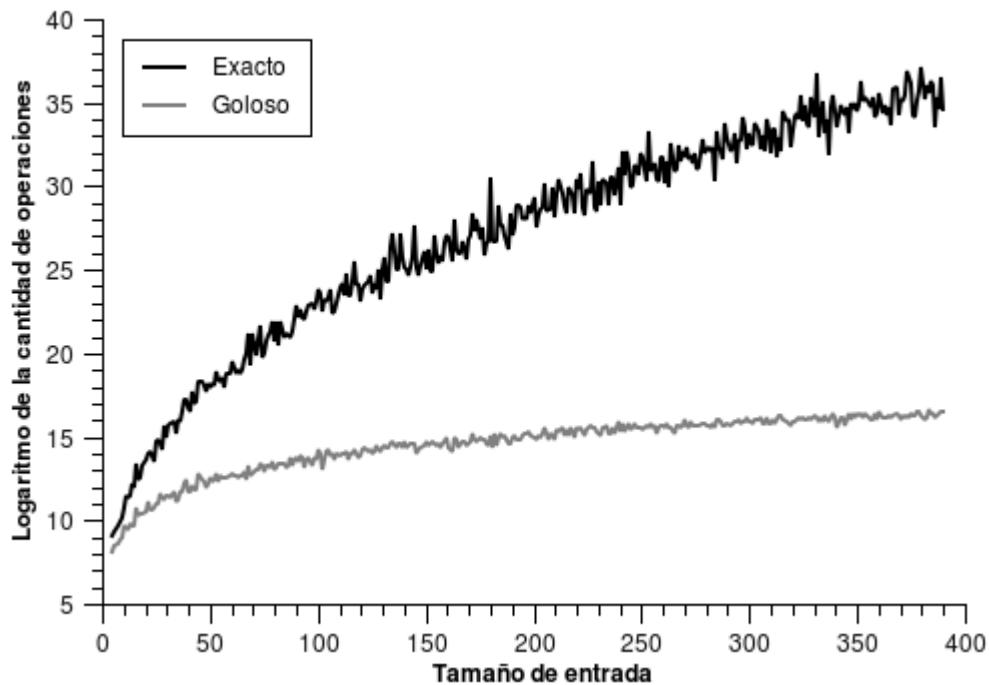
Para el primer archivo:



Para el segundo archivo:



Para el tercer archivo:



Como se puede notar la cantidad de operaciones en el algoritmo exacto es mucho mayor que la cantidad de operaciones del goloso. Si, sumamos la cantidad de operaciones de cada uno de los gráficos para cada uno de los algoritmos obtenemos la siguiente tabla:

Algoritmo \ Archivo Random	1	2	3
Exacto	3318116640309	3118748085286	3248197502833
Goloso	14393755	14382416	14343020

Por lo tanto, el algoritmo exacto hizo en caso promedio 216844 la cantidad de veces de operaciones de lo que hizo el exacto. Si se tiene en cuenta que la diferencia entre las soluciones es de un 11 %, entonces es muy importante tener en cuenta la diferencia en la cantidad de operaciones hechas.

5. Heurística de Búsqueda local

5.1. Introducción

Básicamente lo que hacemos en esta heurística es lo siguiente: Dada una solución inicial,

- Sacamos una cantidad especificada de vértices de la misma. Los vértices sacados siguen un criterio específico.
- Agregamos vértices siguiendo otro criterio específico.
- Nos fijamos si terminamos.

Como solución inicial hay 3 variantes del algoritmo goloso.

Para poder seleccionar los vértices que se sacan se eligieron 3 opciones:

- Sacar el que tiene menor peso. La idea detrás de esto, es que este vértice es el que menos suma a la solución actual. Por lo tanto, lo que se busca es reemplazarlo por alguno que agregue mayor valor a la misma.
- Sacar el de mayor grado. Como los vértices en la solución entre sí no pueden ser adyacentes, entonces el hecho de tener un vértice de grado alto impide de que se puedan agregar bastantes vértices a la misma.
- Sacar el de menor beneficio, es decir, cuando el peso sobre el grado sea mayor. La idea de esto es la misma que el hecho de sacar el de menor peso.

Una variante de este algoritmo que se nos ocurrió fue el hecho de sacar mas de un vértice. Lo que pensamos fue que si la solución tiene dos vértices que bloquean muchos otros vértices potenciales, y solo sacamos uno de ellos es posible que el otro nunca se saque. Por lo tanto, la idea fue que se saca un número configurable de vértices. En caso de que ese número supere la cantidad de vértices que la solución actual, entonces se sacan todos los vértices de la solución actual.

Para poder hacer eso, lo que hacemos es lo siguiente:

- Creamos una lista con todos los vértices que van a ser sacados.
- Ordenamos esa lista según el criterio bajo el cual los vértices van a ser sacados.
- Iteramos la lista.
- Ahora, cuando se tiene que sacar el/los vértice/s, hacemos lo siguiente:
 - Si se tiene que sacar solo uno, entonces sacamos el actual.
 - Si se tiene que sacar k , entonces se saca el actual y los siguientes $k-1$ vértices que figuran en la lista. Si los elementos pasan la longitud de la lista, por ejemplo, si estoy en la posición 4, se tienen que sacar 4 y la lista tiene una longitud de 6, entonces se trata a la lista como si fuese circular.

Es importante tener en cuenta que cuando se intentan de sacar mas de un vértice no intentamos de formar todos las posibilidades de como sacar esa cantidad ya que esto tiene orden exponencial. Sino que tan solo nos quedamos con una forma bastante simple, si el vértice actual es el v_i y se tienen que sacar k vértices, entonces se van a sacar los vértices v_i a v_{i+k} .

Al momento de agregar los vértices tenemos las siguientes opciones (básicamente, son lo contrario de las opciones de sacar):

- Agregar el de mayor peso
- Agregar el de menor grado.
- Agregar el de mayor beneficio

Es importante notar que por cada vértice sacado es posible que se pueda meter mas de un vértice en la solución. Por ejemplo, si se saca el vértice v_1 que es adyacente a v_2 y a v_3 y los mismos no son adyacentes entre sí y a ningún otro vértice de la solución entonces nada me impide ponerlos a los dos en la solución actual. Esto se da por ejemplo en los grafos de tipo árbol cuando se saca el padre de varias hojas. Cuando se vayan a agregar los vértices sabemos que se van a poder agregar a la solución.

Como a la solución actual se le puede sacar mas de un vértice, entonces suponemos que un vecino es valido, si al tener en cuenta los vértices agregados, hubo un vértice sacado que no se encuentra en la solución formada. Por ejemplo, si se sacan los vértices v_i y v_j un vecino valido puede contener a v_i pero no a v_j y viceversa. También existe la posibilidad de que no contenga a ninguna de los dos.

Para agregar vértices lo que hacemos es lo siguiente:

- Crear una lista con todos los vértices que pueden ser agregados a la solución.
- Se ordena esa lista según el criterio de inserción.
- Se itera sobre la lista. Si se puede insertar el vértice actual, entonces lo metemos en la solución. Ademas, hacemos lo siguiente
 - Con otro iterador, iteramos la lista desde el principio, y nos fijamos si podemos agregar el vértice a la solución
 - Si se puede lo agregamos. Sino pasamos al siguiente.
 - Si cuando termina ese ciclo tenemos una mejor o igual solución que la mejor actual, entonces se termina el ciclo de agregar y sacar.
 - Sino, no se hace nada
- Una vez que se termino de agregar los vértices nos fijamos que al menos un vértice que fue sacado no haya sido agregado a la solución. Si todos volvieron a ser agregados entonces se repite el ciclo. Sino se pasa al siguiente paso
- Si cuando se termina este ciclo, no tenemos una mejor solución, probamos de sacar de otra forma los vértices y se vuelve a hacer este ciclo. En caso contrario se termina el ciclo.

Para ver si el algoritmo termina se nos ocurrieron las siguientes ideas:

- Se llego a la solución que contiene todos los vértices del grafo. En este caso sabemos que no se va a poder llegar a una solución mejor.
- Todos los vecinos de la solución actual son peores. Es importante notar que si un vecino tiene el mismo peso, entonces nos movemos para ese vecino.
- Se hicieron la cantidad de operaciones especificadas para esa componente conexa.

Cuando se revisa la vecindad de la solución actual hay dos opciones:

- Revisar todos los vecinos posibles y quedarnos con el mejor de ellos.
- Ir revisando los vecinos, y al encontrar uno que lo mejore, nos movemos a ese vecino.

Entre las dos opciones solo implementamos la segunda. No por quedarse en el primero de los vecinos es peor, ya que en este caso el vecino al que me muevo puede que no sea el mejor de todos los vecinos. Sin embargo, creemos que ninguna de las dos opciones puede asegurar que se va a caer en un mínimo local.

Un detalle importante del algoritmo es que lo comentado anteriormente se hace por cada componente conexa del grafo. Por ejemplo, cuando se tienen que sacar k vértices, los mismos se sacan de la misma componente conexa. Lo mismo para cuando se agregan los vértices.

Esto permite que cuando el grafo tiene mas de una componente conexa, entonces busque la mejor solución para cada una de ellas. De esta forma se mejora la solución final. En caso contrario, si no se trabaja por componente conexa entonces se pueden sacar vértices de la solución que en el grafo están aislados, y por lo tanto no tiene sentido sacarlos de la componente conexa.

Por lo tanto, cuando se identifica una componente conexa, lo que se hace es lo siguiente:

- Si tiene un único vértice, entonces el vértice es aislado y por lo tanto sabemos que tiene que ir en la solución final.
- Si tiene dos vértices, entonces sabemos que la mejor solución va a estar dado cuando se agrega a la solución el vértice de mayor peso.
- Si se tiene 3 o mas vértices entonces se hacen los pasos comentados anteriormente, pero solo se tienen en cuenta los vértices de la componente conexa actual cuando se agregan o sacan vértices.

5.2. Pseudocódigo

El algoritmo que figura a continuación es genérico, en función de como elegir los vértices para agregar y sacar vértices. En ambas funciones se usa la función *valor*, la cual dado un vértice devuelve:

- El grado del vértice.
- El peso del vértice

- El peso sobre el grado del vértice, o sea, el beneficio.

Algorithm 10 heuristicaBusquedaLocal(grafo, soluciónInicial, cantVérticesSacar, maxIteraciones, mejoraMinima)

```

Sea verticesRevisados ←  $\emptyset$ 
Sea soluciónFinal ← soluciónInicial
while ( $\exists$  vértice  $\notin$  verticesRevisados) do
    Sea primerVerticeNoRevisado ← el primer vértice no revisado del grafo.
    Hacer BFS o DFS empezando en primerVerticeNoRevisado
    Sea vérticesComponenteConexa ← todos los vértices de la componente conexa
    Marcar todos los vértices de la lista vérticesComponenteConexa
    soluciónFinal ← busquedaPorComponente(grafo, soluciónFinal, cantVérticesSacar, maxIteraciones, mejoraMi-
        nima, vérticesComponenteConexa)
end while
end
```

Algorithm 11 busquedaPorComponente(grafo, soluciónLocal, cantVérticesSacar, maxIteraciones, mejoraMinima, *vérticesComponenteConexa*)

```

if (vérticesComponenteConexa tiene solo un vértice) then
    Agregar el vértice si el mismo no pertenece a la soluciónLocal
    devolver soluciónLocal
else if (vérticesComponenteConexa tiene dos vértices) then
    Agregar el vértice mas pesado a la soluciónLocal
    Sacar el vértice de menor peso a la soluciónLocal
    devolver soluciónLocal
else
    Sea mejorSolución ← soluciónInicial
    Sea soluciónAnterior ← soluciónInicial
    Sea cantIteraciones ← 0
    while no termine(cantIteraciones, maxIteraciones, vérticesComponenteConexa) do
        Sea listaSacar ← obtenerListaSacar(grafo, mejorSolución, vérticesComponenteConexa)
        Ordenar listaSacar.
        for vértice = 0 hasta len(listaSacar) do
            Sea verticesSacados un arreglo
            Sea soluciónActual una copia de mejorSolución
            for i = 0 hasta cantVérticesSacar do
                Sea vérticeAux ← el que esta a i posiciones de vértice en listaSacar
                Sacar vérticeAux de soluciónActual
                Marcar vérticeAux en verticesSacados
            end for
            Sea listaAregar ← obtenerListaAregar(grafo, soluciónActual, vérticesComponenteConexa)
            for vérticeAregar  $\in$  listaAregar do
                Sea soluciónTemp ← una copia de soluciónActual
                if esValido(vérticeAregar, soluciónTemp) then
                    Agregar vérticeAregar a soluciónTemp
                    for vérticeAux  $\in$  listaAregar do
                        if esValido(vérticeAux, soluciónTemp) then
                            Agregar vérticeAux a soluciónTemp
                        end if
                    end for
                    if peso(soluciónTemp) > peso(mejorSolución) then
                        mejorSolución ← soluciónTemp
                        Salir del for de agregar y sacar vértices
                    end if
                end if
            end for
        end for
        if no cambie la mejorSolución o peso(mejorSolución) == peso(grafo) then
            devolver mejorSolución.
        end if
        cantIteraciones ← cantIteraciones +1
    end while
```

```
end while
devolver mejorSolución
end if
end
```

Algorithm 12 termine(cantIteraciones, maxIteraciones, vérticesComponenteConexa)

```

Sea total  $\leftarrow \frac{\max\text{Iteraciones} * \text{len}(\text{verticesComponenteConexa})}{100}$ 
if cantIteraciones  $\leq$  total then
    devolver false
else
    devolver true
end if
end

```

Algorithm 13 obtenerListaSacar(grafo, soluciónActual, vérticesComponenteConexa)

```

Sea resultado  $\leftarrow \emptyset$ 
for vértice  $\in$  vérticesComponenteConexa do
    if vértice  $\in$  soluciónActual then
        Agregar (vértice, valor(vértice)) a resultado
    end if
end for
devolver resultado
end

```

Algorithm 14 obtenerListaAgregar(grafo, soluciónActual, vérticesComponenteConexa)

```

Sea resultado  $\leftarrow \emptyset$ 
for vértice  $\in$  vérticesComponenteConexa do
    if esValido(soluciónActual, vértice) then
        Agregar (vértice, valor(vértice)) a resultado
    end if
end for
devolver resultado
end

```

5.3. Detalle de implementación

Ahora cuando se ordena tanto la lista de los vértices a sacar como la lista de los vértices a agregar, la misma se ordena se ordena según el valor. También, teniendo en cuenta el criterio, esa lista se puede ordenar de mayor a menor o de menor a mayor. Por ejemplo, cuando se saca el vértice de mayor grado, esa lista se ordena de mayor a menor, pero cuando se saca el de menor peso se ordena de menor a mayor.

El algoritmo esta predeterminado para ordenar las listas de menor a mayor. Si es necesario que se ordene de forma inversa lo que se hace, es ordenarla y después darla vuelta. El hecho de darla vuelta tiene una complejidad de $O(l)$ donde l es la cantidad de elementos de la lista. Como veremos mas adelante, la complejidad de ordenar la lista es $O(l * \log(l))$ y por lo tanto, el hecho de darla vuelta no suma a la complejidad final, sino que solo cambia la cantidad de operaciones.

Otro hecho a tener en cuenta es el hecho de cuando se saca mas de un vértice de la *soluciónActual*. Para poder sacar los vértices v_i hasta v_{i+k} , lo que hacemos es pasar la lista *listaSacar* a un arreglo. Por lo tanto, solo se tienen que acceder a las posiciones v_i hasta v_{i+k} . Si la posición v_{i+k} es mayor que la longitud del arreglo, entonces se usa la posición v_{i+k-j} donde j es la longitud del arreglo.

5.4. Complejidad

Veamos ahora el análisis de complejidad del algoritmo. Empecemos por la complejidad de las funciones auxiliares.

Veamos la función *termine*. Por como se implemento la solución¹⁰, el hecho de obtener el peso de la misma se hace en $O(1)$. Como todas las operaciones que también se hacen en esta función son $O(1)$ porque estamos en el modelo uniforme¹¹, entonces la complejidad de esta función es $O(1)$.

Ahora veamos la función *obtenerListaSacar*. De nuevo, por la implementación de la solución, podemos saber si un vértice cualquiera pertenece a la misma o no en $O(1)$, ya que los vértices de la misma están sobre un arreglo. Luego, verificar si un vértice cualquiera pertenece a la solución es tan solo verificar el valor de esa posición del arreglo. Por lo tanto, como esta función se hace para todos los vértices de la componente conexa, y el resto de las operaciones son tienen complejidad constante, la complejidad de esta función es $O(k)$ donde k es la cantidad de vértices de la componente conexa. Como la componente conexa puede a lo sumo tener todos los vértices del grafo, entonces sabemos que la complejidad de esta función es: $O(n)$ donde n es la cantidad de vértices del grafo.

¹⁰Ver Heurísticas y Algoritmo, Estructuras Usadas, Solución

¹¹Ver Heurísticas y Algoritmos, Notas Aparte

Ahora veamos la función *obtenerListaAregar*. Esta es similar a la función *obtenerListaSacar*, teniendo en cuenta que el vértice no tiene que estar en la solución. Como las operaciones tienen complejidad O(1), al igual que el resto de las operaciones de la misma, y el algoritmo se repite para todos los vértices de la componente conexa, entonces la complejidad es: O(k) donde k es la cantidad de vértices de la componente conexa. Por la misma razón que la función *obtenerListaSacar*, la complejidad de esta función es O(n)

Por ultimo nos queda verificar el orden de la función principal. Sabemos que el hecho de obtener la lista *listaSacar* es O(n). Para ordenar una lista, se la puede pasar a un arreglo y luego ordenar el arreglo, y luego volver a pasarl a una lista. Esto se puede hacer en:

$$O(n + n * \log(n) + n) \in O(n * \log(n))$$

El hecho de copiar la *mejorSolución* a la *soluciónActual* tiene complejidad O(n). El hecho de pasar la lista de los vértices a sacar a una arreglo tiene complejidad O(n). Además, el hecho de obtener el vértice a sacar es O(1) ya que *listaSacar* se la copio a un arreglo. Por último, el sacar un vértice tiene complejidad O(n) ¹², y el marcarlo tiene complejidad O(1) ya que solo se tiene que acceder a esa posición del arreglo. Como se sacan *cantVérticesSacar*, el hecho de sacar todos los vértices para luego formar los vecinos tiene complejidad:

$$O(\text{cantVerticesSacar} * n)$$

Ahora veamos la parte de cuando se agregan vértices. Como dijimos anteriormente, el hecho de copiar *soluciónActual* a *soluciónTemp* tiene complejidad O(n). El hecho de verificar si *vérticeAregar* se puede agregar a *soluciónTemp* se hace en O(1), y el hecho de agregarlo se hace en O(n). Lo mismo para todos los vértices de *vérticeAux*. Como la lista puede a lo sumo tener todos los vértices del grafo, la complejidad del mismo es O(n). Por lo tanto, el orden de completar la *soluciónTemp* para que sea maximal, es:

$$O(n^2)$$

El hecho de verificar si la solución formada después de agregar los vértices es diferente que la anterior, se hace revisando que alguno de los vértices agregados no se encuentre en la lista de vértices sacados. Como los vértices sacados se lo tiene sobre un arreglo, entonces esto se hace en O(n).

Luego, como el ciclo de probar por cual vértice se empieza sacando se hace a lo sumo n veces, entonces la complejidad de una iteración del ciclo es:

$$O(n * (n^2 + n + n * \text{cantVerticesSacar}))$$

Como $n * \text{cantVerticesSacar}$ no puede ser mayor que O(n^2), entonces la complejidad queda:

$$O(n^3)$$

Por último nos queda ver la cantidad de veces que se repite el ciclo principal. La cantidad de veces que se ejecute el ciclo principal va a depender de la cantidad de iteraciones que fueron dadas y el tamaño de cada una de las componentes conexas ¹³. Por lo tanto, a la complejidad la vamos a dejar expresada como:

$$O(\text{cantIteraciones} * n^3)$$

donde *cantIteraciones* es la cantidad de veces que se hace el ciclo principal.

A lo sumo, el grafo es un conjunto de vértices aislados por lo que el ciclo anterior no se puede repetir mas de n veces. Por lo tanto, la complejidad del algoritmo sin tener en cuenta el orden del algoritmo constructivo es:

$$O(\text{cantIteraciones} * n^4)$$

Ahora el hecho de calcular la solución inicial tiene la complejidad O($n^2 * \log(n)$), en el pero de los casos. Por lo tanto, el orden final de la heurística es:

$$O(n^2 + \text{cantIteraciones} * n^4) \in O(\text{cantIteraciones} * n^4)$$

Por último vemos el tamaño de entrada para este problema.

$$\begin{aligned} T &= \log(n) + \sum_{i=1}^n \log(\text{peso}(i)) + \sum_{i=1}^n \sum_{v \in \text{ad}(i)} \log(v) \\ T &\geq \sum_{i=1}^n \log(\text{peso}(i)) \\ T &\geq \sum_{i=1}^n 1 \\ T &\geq n \end{aligned}$$

¹²Ver Heurísticas y Algoritmos, Estructuras Usadas, Solución

¹³Ya que *maxCantIteraciones* es un porcentaje en función de la cantidad de vértices de la componente conexa que se está mejorando.

Como no sabemos cual es el valor de p , no sabemos cual va a ser el valor del mismo en función del tamaño de entrada. Llamamos q al valor que es representar p sobre el tamaño de entrada. Por lo tanto, la complejidad de este algoritmo es:

$$O(\text{cantIteraciones} \cdot T^4)$$

y la misma es polinómica.

5.5. Gráficos

De nuevo acá, se va a analizar como se comporta el algoritmo en función del tiempo sin tener en cuenta la calidad de la solución. Se va a intentar de buscar el peor caso, en función del tiempo.

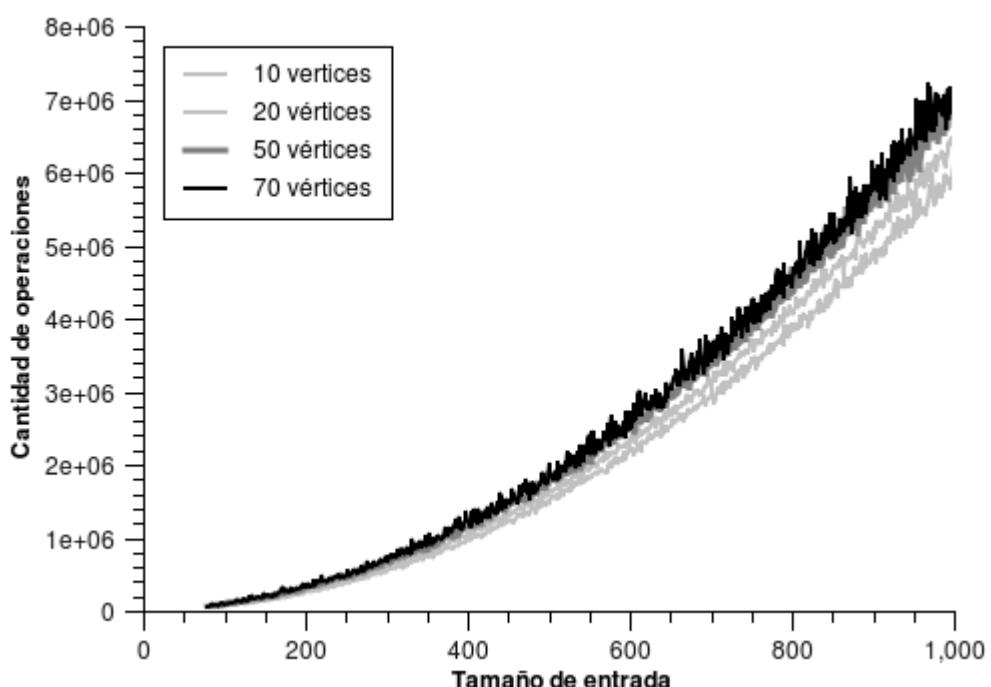
Lo primero que planteamos es cual es el peor caso para cada una de las *variables* del sistema:

- Para que la función *busquedaPorComponente* se ejecute muchas veces, entonces todos los vértices del grafo tienen que ser aislados. Sin embargo, en ese caso, la función se hace muy rápido ya que cada una de las componentes conexas tiene un único vértice.
- Para que el ciclo la función *busquedaPorComponente* se repita la mayor cantidad de veces posibles, influye el hecho de la cantidad de iteraciones que se tienen que hacer. Si la componente conexa tiene pocos vértices entonces *maxIteraciones* tiene que ser muy grande, y realmente mejorar esa cantidad de veces, porque sino la función termina porque no encontro un vecino que sea mejor que la solución actual.
- La cantidad de vértices sacados de la solución actual. Esto influye ya que mientras mayor se la cantidad de vértices sacados, mayor es la cantidad de veces que se repite el ciclo de buscar una mejor solución.

Claramente la cantidad de iteraciones que haga el algoritmo esta relacionado de la solución inicial. Por ejemplo, si la misma es demasiado buena, y por lo tanto, muy cercana a la solución ideal, entonces es muy difícil que se mejore el porcentaje requerido. Sin embargo, si se empieza con una solución mala, entonces sería razonable que la búsqueda local la mejore, por lo tanto la se debería de pasar varias veces por el mínimo requerido.

Además, hay una relación entre el la cantidad de iteraciones y el tamaño de la componente conexa. Por mas de que *maxIteraciones* se mantenga constante si crece el tamaño de la componente conexa, entonces crece la cantidad de iteraciones hechas, dado que *maxIteraciones* es un porcentaje en función de la cantidad de vértices de la componente conexa que se esta analizando.

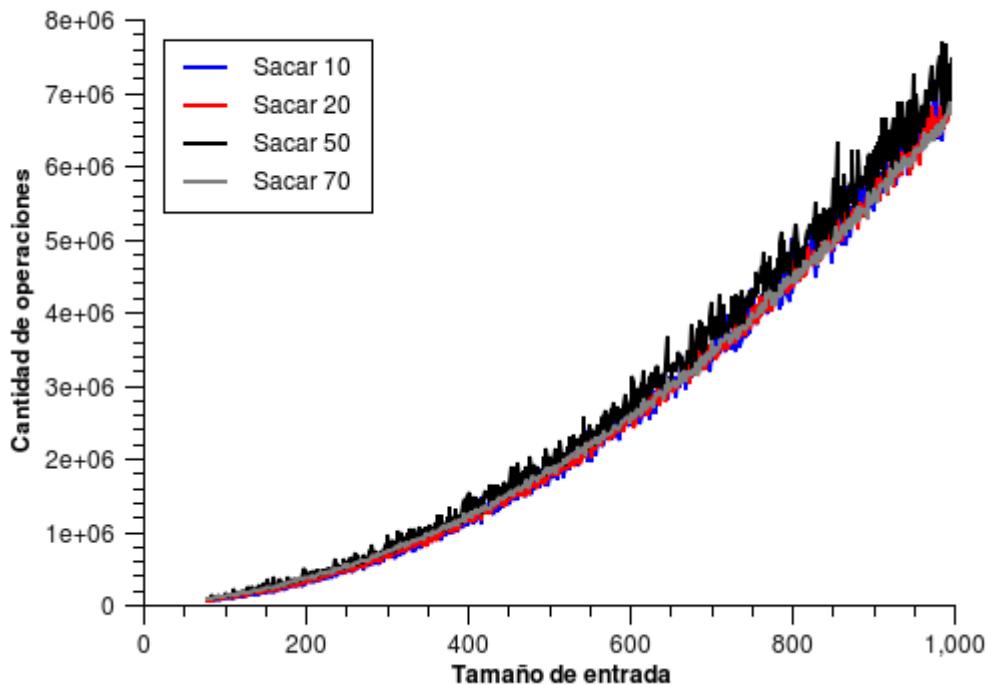
En el gráfico que figura a continuación son casos donde el grado de los vértices es siempre el mismo, al igual que el peso, pero el tamaño de cada una de las componentes conexas del grafo es diferente. Es importante notar, que en todos los casos se uso la misma configuración. Es decir, se uso el mismo algoritmo para sacar vértices, para agregar, y el mismo porcentaje de *maxIteraciones*.



Aunque se nota poco, la cantidad de operaciones crecen a mediada que la cantidad de vértices de la componente conexa crece. Esto resulta ya que la cantidad de iteraciones del ciclo de la función de buscar el vecino hace mayor cantidad de operaciones ya que tienen mas vértices. Esto influye cuando se va a seleccionar el vértice que se va a sacar o vértices a agregar.

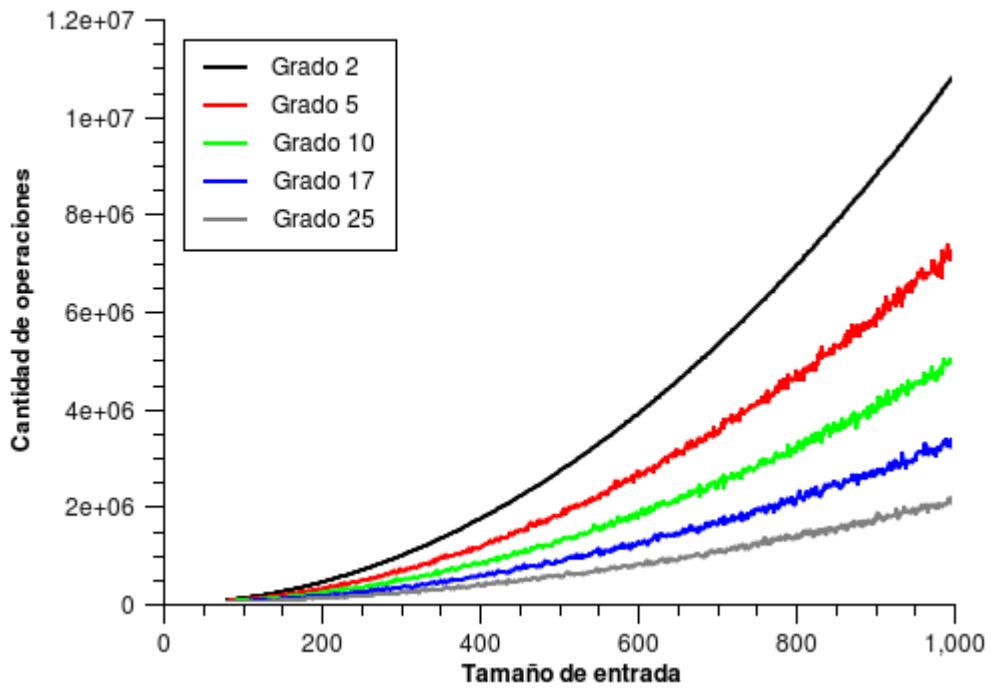
Luego, es normal que se hagan mas operaciones a medida que la cantidad de vértices por componente crece ya que *maxIteraciones* es el mismo porcentaje para todos, pero a medida que el algoritmo crece, también crece la cantidad de iteraciones que se puede hacer.

En el gráfico que figura a continuación se muestra la cantidad de iteraciones cuando se tiene un grafo donde todos los vértices tienen el mismo grado y peso, y todos los grafos tienen la misma cantidad de componentes conexas, pero en el algoritmo lo que se varía es el porcentaje de los vértices que se sacan.



Como se puede notar en el grafo, la cantidad de operaciones prácticamente no cambia por más de que crezca el porcentaje de vértices a sacar del algoritmo. Aunque crece la cantidad de operaciones, no es tan influyente como la cantidad de componentes conexas del grafo.

Por último, se gráfica un caso donde la cantidad de vértices por componente conexa es fija, al igual que el peso de los vértices, pero donde cambia el grado de los vértices. En este caso, todas las variables del algoritmo de búsqueda local tienen todos el mismo valor.

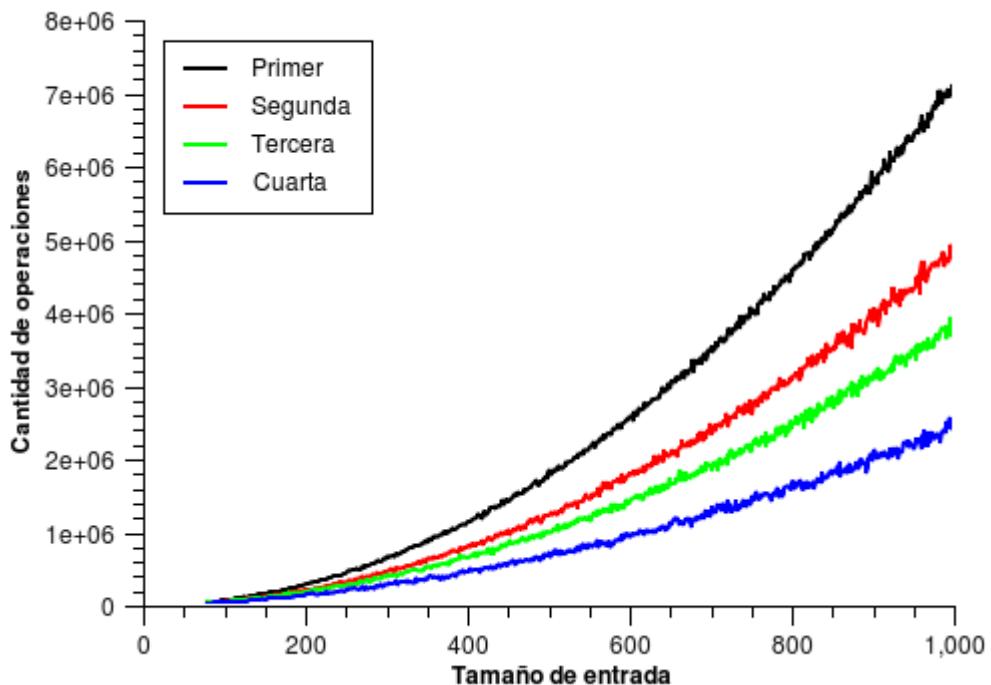


Como era de esperarse, la cantidad de operaciones del algoritmo baja a medida que el grado del vértices se hace mayor. Esto sucede porque al ser mayor el grado, menos son los vértices que puedo llegar a agregar a la solución.

Por ultimo, variamos las 4 cosas:

- Cantidad de vértices por componentes conexas.
- Grado de los vértices.
- Cantidad de vértices a sacar.
- Cantidad de iteraciones a hacer.

En cada paso, las 4 variables se hacen mas grandes, resultando en el siguiente gráfico:



Es importante notar, que a medida que crecía la cantidad de vértices por componente conexa, también crecía el grado.

5.5.1. Conclusiones

Aunque no se pudo encontrar un peor caso para el algoritmo, ya que el mismo depende de demasiadas variables, hay casos en donde se nota que son peores que otros.

Del último gráfico, se puede llegar a deducir que lo mas importante es el grado del vértice, ya que esto permite que se tengan en cuenta mas vecinos. Sin embargo, también hay que tener en cuenta la solución que haya dado el algoritmo goloso ya que si la solución fue muy mala, entonces el ciclo se puede repetir una gran cantidad de veces ya que siempre encuentra un vecino mejor.

5.6. Análisis de la solución

5.6.1. Solución correcta y erróneas

Algo importante, es que para poder analizar cuando falla y cuando da una solución exacta, depende en gran parte de la solución hecha por el algoritmo constructivo.

Veamos entonces los casos de los algoritmos constructivos y busquemos una combinación de agregar y sacar de forma tal que se llegue a la solución exacta.

Constructivo MayorPeso Sabemos que este algoritmo falla:

- Árboles: para solucionar el problema, una posible solución seria sacar los vértices de mayor grado, y poner los de menor grado. En este caso, las hojas tienen menor grado que el vértice padre, y por lo tanto, si se saca al padre y se pone una hoja, la solución se puede completar con todas las hojas del padre. Obteniendo así, una mejor solución.
- Bipartito completo: acá el problema se encuentra en que tenemos que sacar todos los vértices de la solución usando el contador *cantVerticesSacar*. No sirve sacar solo 1, ya que al ser un bipartito, no se va a poder poner ningún otro vértice salvo el que se saco. Por lo tanto, considero en este caso que la búsqueda local no mejora esto.

Constructivo Mayor Beneficio Los dos casos en los que falla este algoritmo son similares a los del constructivo del mayor peso. Por lo tanto, se podría decir que se va a solucionar el problema de los árboles, pero el de bipartito completo.

Constructivo menor grado Por último, veamos si se pueden mejorar los casos donde este algoritmo falla:

- Completos: en este caso, se puede seleccionar cualquier algoritmo de sacar, ya que la solución solo va a tener un único vértice. Por lo tanto, si luego de sacarlo se agrega el de mayor peso o el de mayor beneficio, se llega a una solución óptima.
- Bipartitos Completos: como se comento anteriormente, para poder modificar la solución inicial para este tipo de grafos, es necesario sacar primero todos los vértices que tiene la solución inicial, voy a decir que para este caso, la búsqueda local no mejora el caso.

5.6.2. Ejecuciones

El búsqueda local se ejecuto siempre luego de obtener el resultado del algoritmo goloso cuando se usa la variante de *Peso Grado*. Fuera de eso se ejecutaron las siguientes variaciones:

- Algoritmo Agregar: se uso tanto el de Mayor Grado, Menor Peso, y Peso Grado.
- Algoritmo Sacar: se uso tanto el Menor Grado, Mayor Peso y Peso Grado.
- Porcentaje de nodos a sacar: se usaron los valores 10, 20, 50 y 70
- Porcentaje de iteraciones: se usaron los valores 10, 20, 30, 50, 70, 100, 300, 500, 1000.

Cada uno de los archivos se calculo para todas las posibles combinaciones de esos valores¹⁴

Como el algoritmo tiene varios casos en los cuales vale la pena analizarlos por separado, entonces vemos como es que se comporta el mismo para cada uno de los archivos de entrada.¹⁵

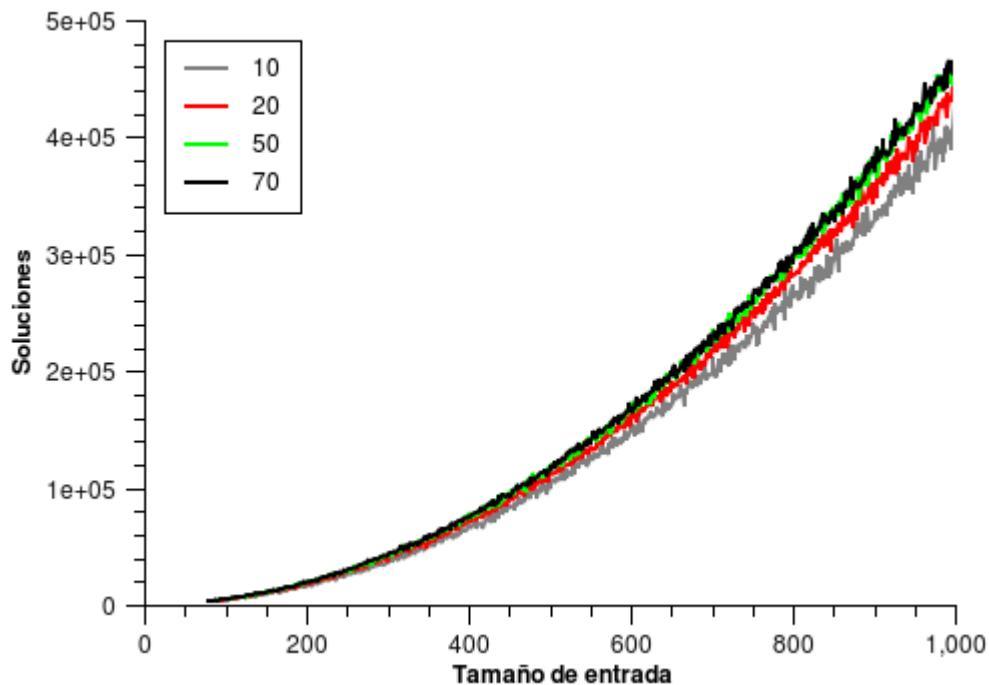
Es importante notar que en todos los gráficos que figuran a continuación siempre se uso la misma variante de la búsqueda local, es decir, siempre se uso el mismo algoritmo para agregar vértices, el mismo para sacar, y el porcentaje de vértices a sacar y el de la cantidad de iteraciones a hacer. Las combinaciones elegidas corresponden al mejor caso de la búsqueda local, sobre el cual se comenta luego.

¹⁴Si, se ejecutaba la búsqueda local 324 veces por cada archivo de entrada

¹⁵Ver Heurísticas y Algoritmos, Archivos de entrada

5.6.3. Variar Componentes conexas

Para el gráfico que figura a continuación, se dejó fijas todas las variables del algoritmo, salvo que el grafo varía la cantidad de vértices por cada componente conexa del mismo. Es decir, todos los vértices del grafo tienen el mismo grado, pero cada linea graficada tiene diferente cantidad de vértices por componentes conexas.



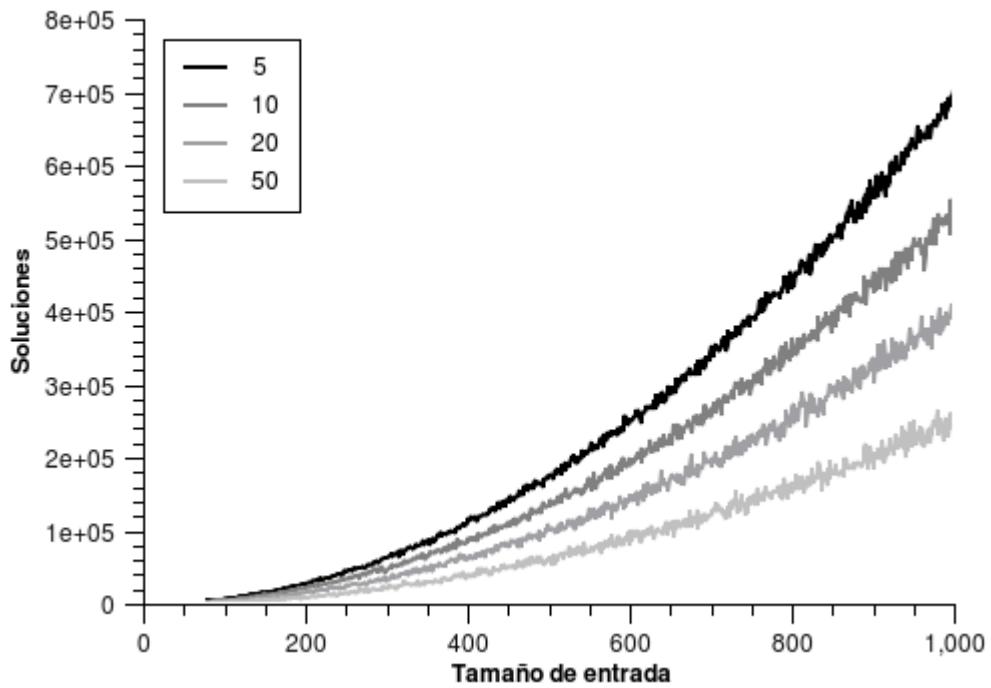
Aunque realmente la mejora no es mucho, mientras mas grande es la componente conexa mejor es la solución. Sin embargo, como vimos anteriormente esto esta relacionado con el grado de los vértices. Como en el gráfico anterior, la diferencia entre cada uno de los casos es realmente poca, a continuación se muestra la tabla de todas las soluciones sumadas para cada opción:

Cant Vertices por componente	10	20	50	70
Soluciones sumadas	136348530	146650728	153507666	155081904

Tal como muestra el gráfico, la soluciones en promedio mejoran, pero la diferencia no es muy grande.

5.6.4. Variar Grado

En el gráfico que figura a continuación también se dejó fija las variables del algoritmo, pero se cambio el grado en los vértices y no se sabe la cantidad de componentes conexas del mismo.

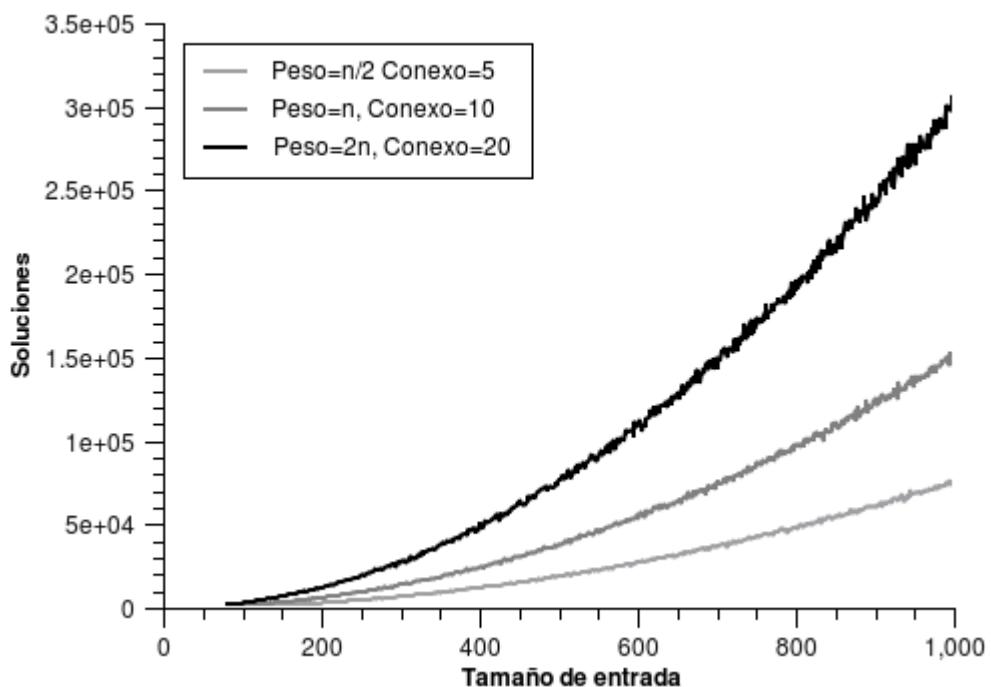


Como se puede ver en el grafo, el valor de la solución baja a medida que el grado sube. Esto no es algo particular de la búsqueda local sino que sucede para todos los algoritmos. Es decir, a medida que el grado de los vértices se hace mayor, entonces esto implica que menos vértices pueden llegar a ir a la solución.

Lo mismo sucede en el para los archivos dentro de **Variar Grado Conexo**, ya que los mismos tienen la misma característica. En esos archivos se mantiene constante la cantidad de vértices por componente conexa, y también el peso de los vértices, pero cambia el grado de los vértices. Por lo tanto, al haber menos vértices en la solución esto va a hacer que el valor de la misma sea menor.

5.6.5. Variar Peso Conexo

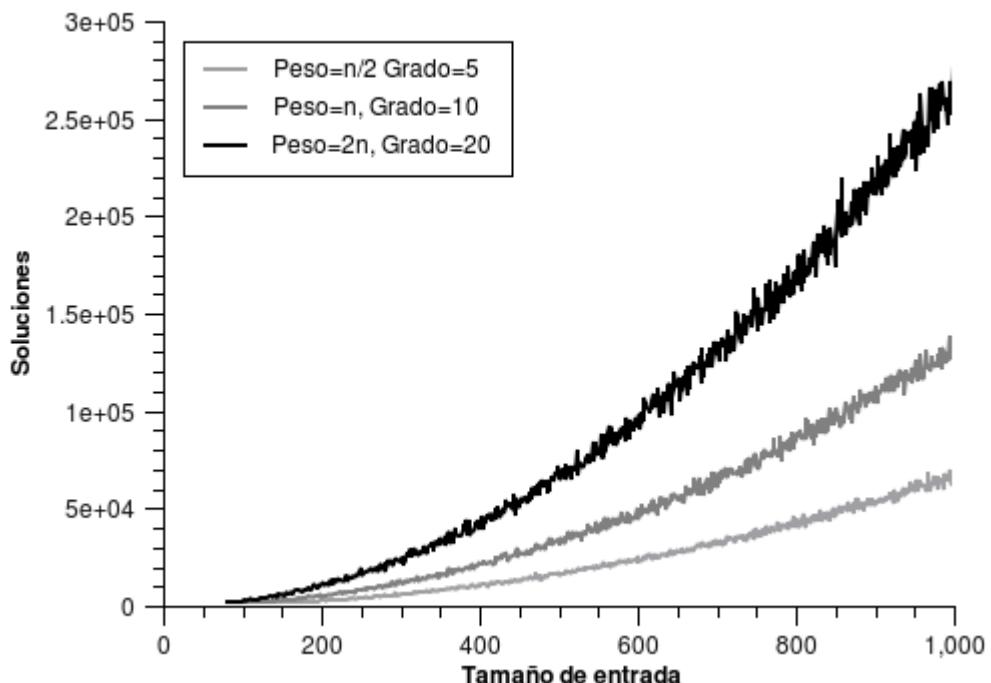
En el gráfico que figura a continuación se dejó fijo el grado de los vértices, pero tanto la cantidad de vértices por componente conexa como el peso de los vértices crece.



Tal como era de esperarse, crece la solución ya que los pesos de los vértices crece, y el grado de los vértices se mantiene constante. Además, al ser mayor la cantidad de vértices por componente conexa y que las mismas mantengan el mismo grado, permite que se seleccionen mas vértices por cada componente conexa.

5.6.6. Variar Peso Grado

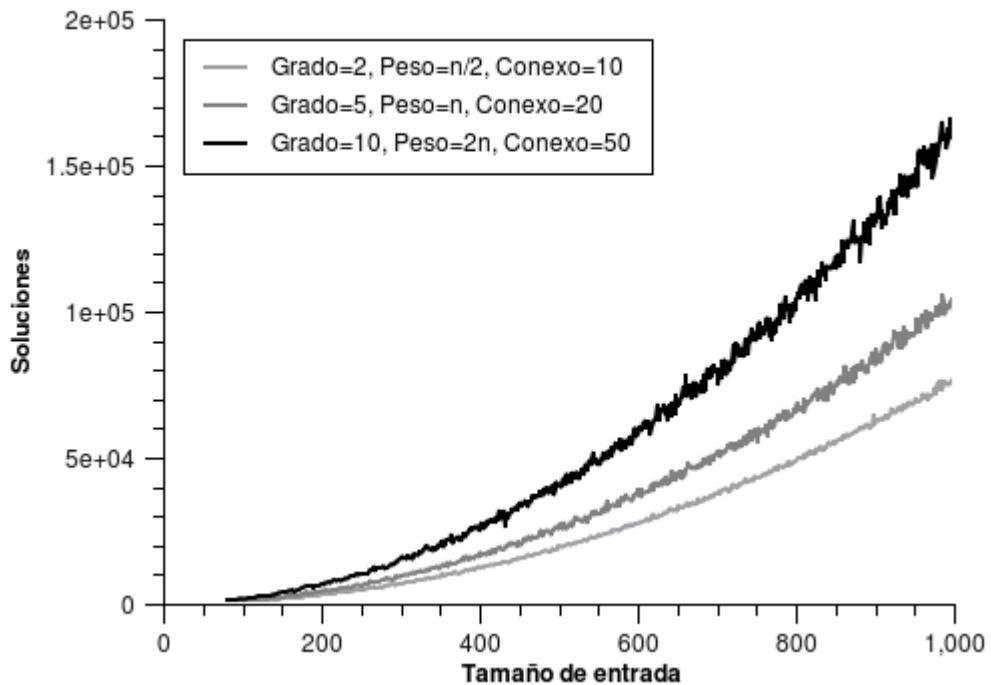
En el siguiente gráfico se cambia tanto el grado como el peso de los vértices, pero no se sabe la cantidad de componentes conexas que tiene el mismo. Es similar al anterior, salvo que en este caso, cambia la cantidad de vértices por componentes



Un pensamiento común sería pensar que mientras mayor es el grado, menos vértices van a ir en la solución. Eso es normal, pero lo que no es cierto es que al doble del grado de los vértices, haya en la solución la mitad de los vértices. Es por esto, que por mas de que el grado se duplique, también lo hace el peso, y aunque seguramente haya menos vértices en la solución la cantidad no necesariamente se reduce por la mitad.

5.6.7. Variar todo

Por último, en el gráfico que figura a continuación se vario tanto el grado, como la cantidad de vértices del grafo, como el peso de los vértices.



Acá, al igual que el caso anterior, sucede que por mas de que el grado de los vértices se hace mayor, no es cierto que la solución también disminuye en esa proporción.

5.6.8. Mejor Combinación

Como se comentó anteriormente se corrieron 324 variantes de la búsqueda local cada una con al menos un valor diferente de configuración. Esas 324 se corrieron para cada uno de los archivos de entrada. El problema queda ahora en identificar cual es la mejor solución.

Si para cada archivo se ordenan todas las 324 variantes en función del valor de la sumatoria de todas las soluciones y de la sumatoria de la cantidad de operaciones. Además, si a eso se lo ordena de forma decreciente en función de la suma de las soluciones, y en forma creciente en función de la cantidad de operaciones, se puede identificar para cada archivo en general cual fue la variante que dio la mejor solución en la menor cantidad de operaciones.

Haciendo eso sobre todos los archivos de entrada, y luego buscando un patrón en común llegamos a la conclusión de que el mejor caso para este algoritmo es:

- Se tiene que seleccionar el vértice de *Menor Grado* cuando se agregan vértices para moverse al vecino.
- Cuando se sacan vértices para moverse al vecino, la mejor solución va a estar dada por cuando se saca el vértice de *Mayor Grado*.
- En cada iteración de ir a buscar el vecino, se tiene que sacar el 20 % de los vértices que se encuentran en la solución y que también están en la componente conexa.
- La cantidad de iteraciones resulto ser el 20 % de la cantidad de vértices que tiene la componente conexa que se esta optimizando.

Es importante tener en cuenta, que estas variables son las mejores cuando se elije como variante de goloso la de *Peso Grado*.

Otra idea fue la de calcular cual fueron los valores de configuración que dieron la mejor solución. Es decir, que variante del algoritmo para agregar vértices dio la mejor solución si se tienen en cuenta todos los archivos de entrada. Para esto, lo que se hizo es sumar las soluciones de todos los casos de todos los archivos, y sumarle el valor eso a las opciones de configuración que correspondan.

Luego de hacer eso, se obtuvieron las siguientes tablas:

Opciones de Agregar	Soluciones sumadas	Opciones de Sacar	Soluciones sumadas
MayorPeso	6847731441991	MenorPeso	6882655348264
MenorGrado	6952054618331	MayorGrado	6882206805785
PesoGrado	6847731441991	PesoGrado	6882655348264

Opciones de Iteraciones	Soluciones sumadas
10	2294155221131
20	2294170277448
30	2294170286222
50	2294170286252
70	2294170286252
100	2294170286252
300	2294170286252
500	2294170286252
1000	2294170286252

Opciones de Cant Vertices Sacar	Soluciones sumadas
10	5161507081298
20	5168131183199
50	5162774524461
70	5155104713355

De esta tabla se puede deducir que en **general**:

- El algoritmo que se elige para agregar un vértice es importante, ya que hay una gran diferencia entre las diferentes opciones.
- No tiene sentido que la cantidad de iteraciones que se hagan sean mayores al 50 %, ya que en esos casos siempre se llega a la misma solución. Tal vez esto sucede porque en esa cantidad de iteraciones se llega a un mínimo local y por lo tanto el algoritmo termina.
- Si se sacan demasiados vértices de la solución la misma no mejora, sino que realmente empeora.

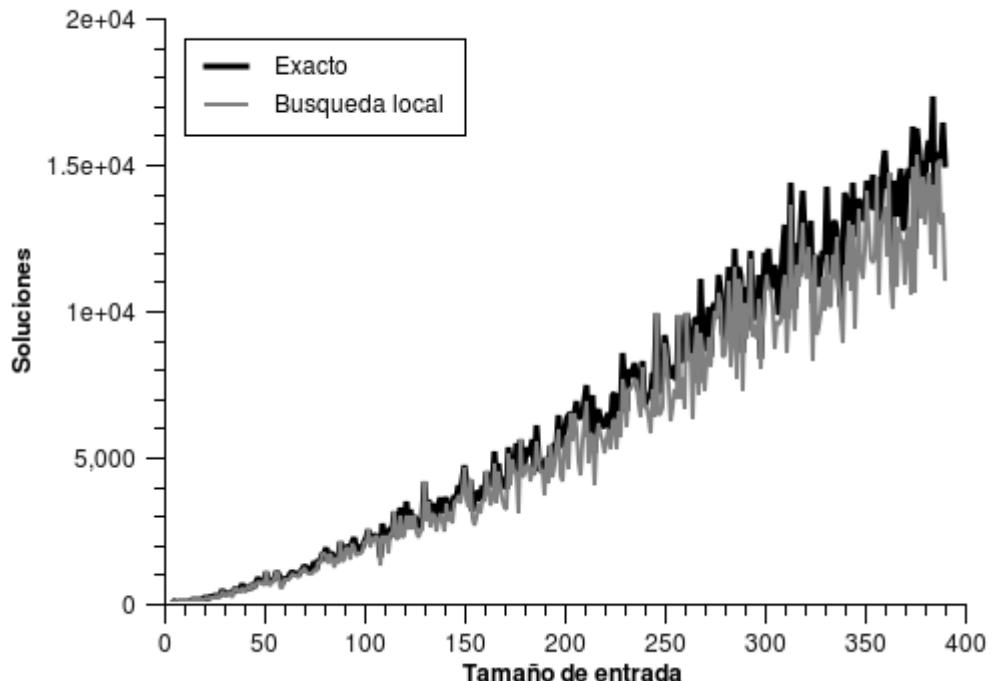
Como el caso que fue seleccionado como mejor caso, tiene los mejores valores, o esta cerca de los mismos, entonces se usa esa opción para luego ejecutar el GRASP y para comparar contra el algoritmo exacto y el goloso.

5.7. Comparaciones

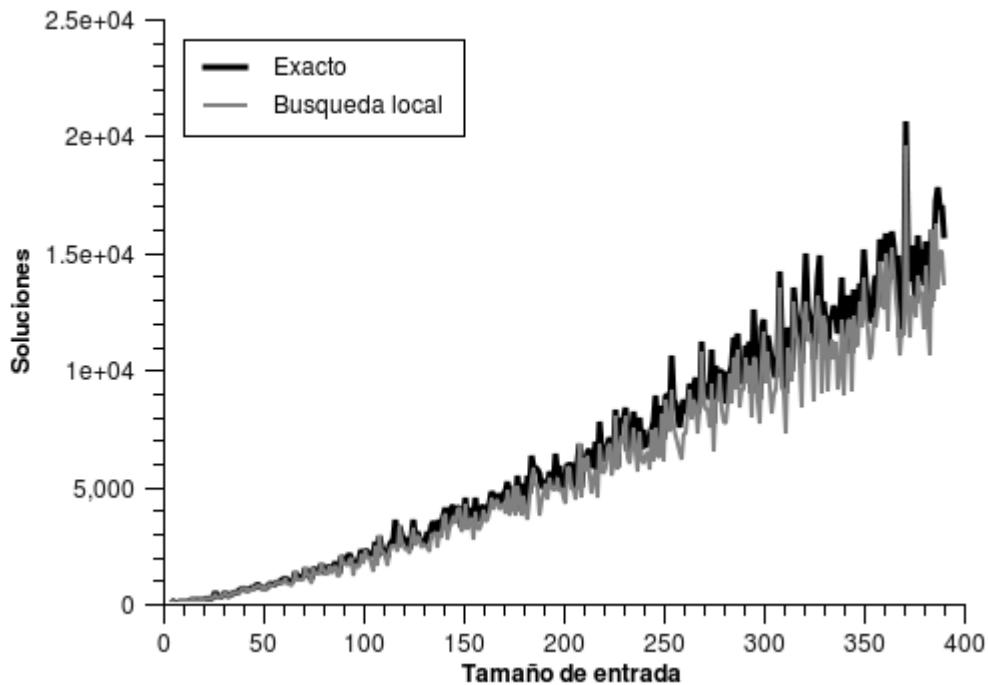
Ahora que tenemos definido cual es el mejor caso para la búsqueda local, tenemos que comprarlo contra el algoritmo exacto y el goloso. Para estas comparaciones también vamos a usar los archivos randoms.

5.7.1. Algoritmo exacto

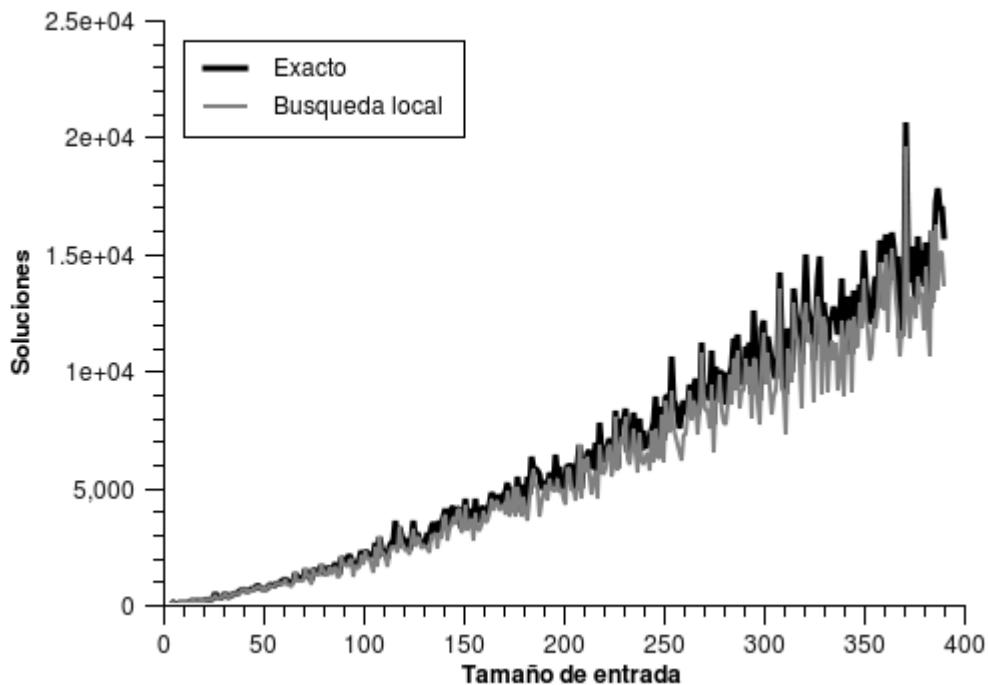
Para el primer archivo:



Para el segundo archivo:



Para el tercer archivo de entrada random:

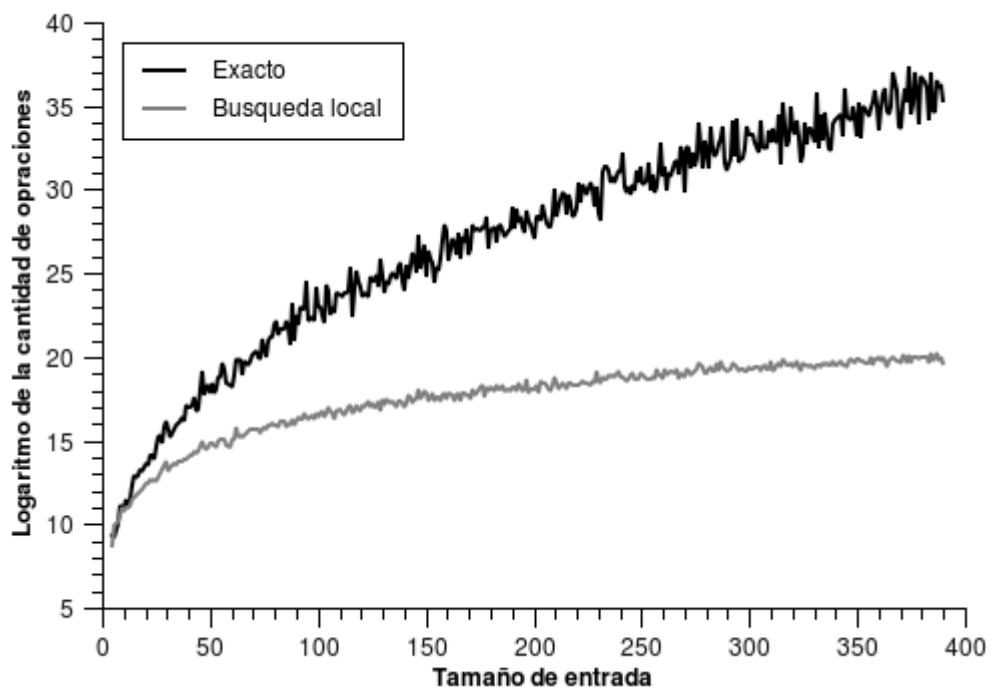


Si se comparan estos gráficos contra los gráficos del algoritmo goloso, es decir, cuando al goloso se lo compara contra el algoritmo exacto, es difícil notar una diferencia. Si hacemos la tabla que hicimos en su momento con el goloso obtenemos los siguientes resultados:

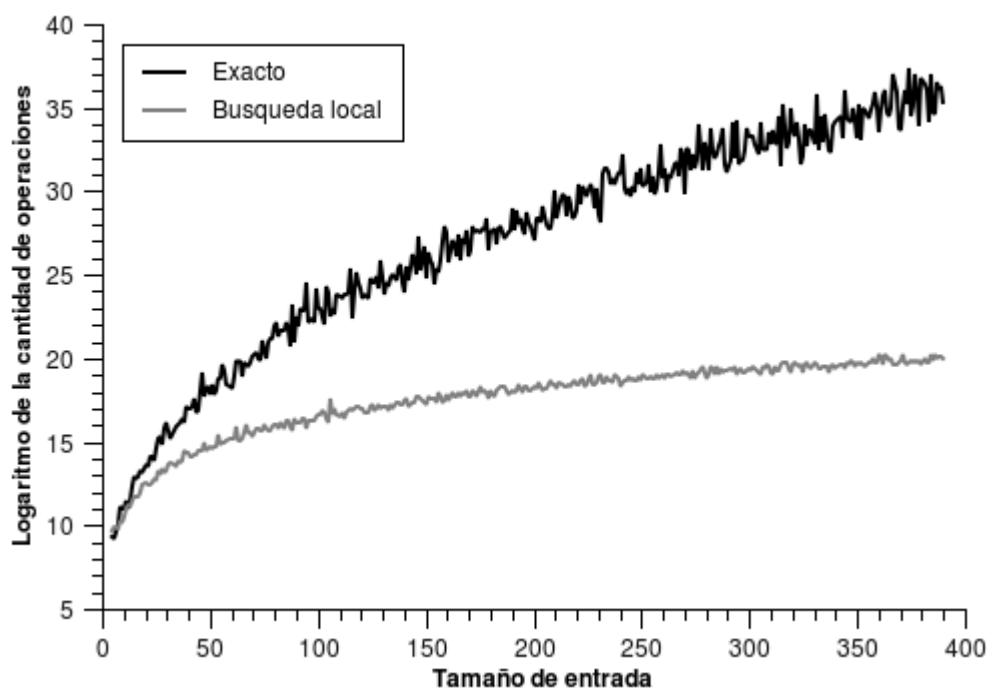
Algoritmo \ Archivo Random	1	2	3
Exacto	2484885	2478760	2453158
Busqueda local	2295597	2289605	2267010

A diferencia que el algoritmo goloso, ahora la diferencia es cerca del 8 % de la solución exacta. Por lo tanto, la diferencia se redujo en un 3 % comparado contra la solución que había dado el algoritmo goloso. Ahora vemos la cantidad de operaciones. De nuevo, se gráfico el logaritmo de la cantidad de operaciones.

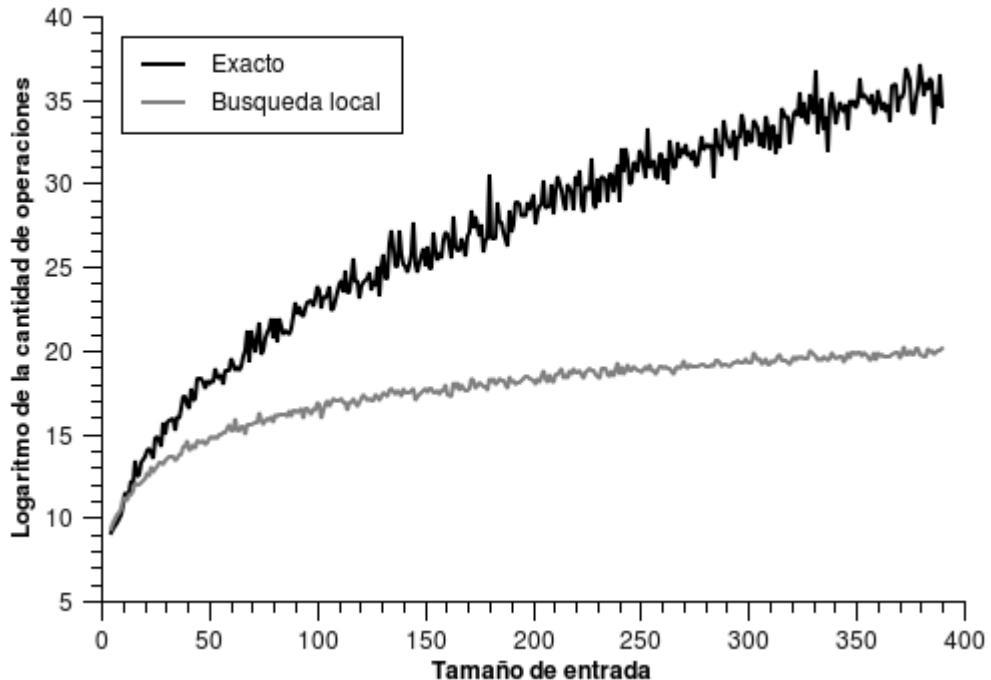
Para el primer archivo:



Para el segundo archivo:



Para el tercer archivo de entrada random:



Al igual que el algoritmo goloso, la búsqueda local también se había corrido para grafos de hasta 1000 vértices, pero solo se hizo la cuenta para grafos de hasta 390 vértices, por que hasta ese tamaño de entrada fue calculado por el exacto.

Si calculamos la tabla anterior, pero esta vez para la cantidad de operaciones obtenemos los siguientes valores:

Como se puede notar la cantidad de operaciones en el algoritmo exacto es mucho mayor que la cantidad de operaciones del goloso. Si, sumamos la cantidad de operaciones de cada uno de los gráficos para cada uno de los algoritmos obtenemos la siguiente tabla:

Algoritmo \ Archivo Random	1	2	3
Exacto	3318116640309	3118748085286	3248197502833
Busqueda local	150789828	150153731	150363072

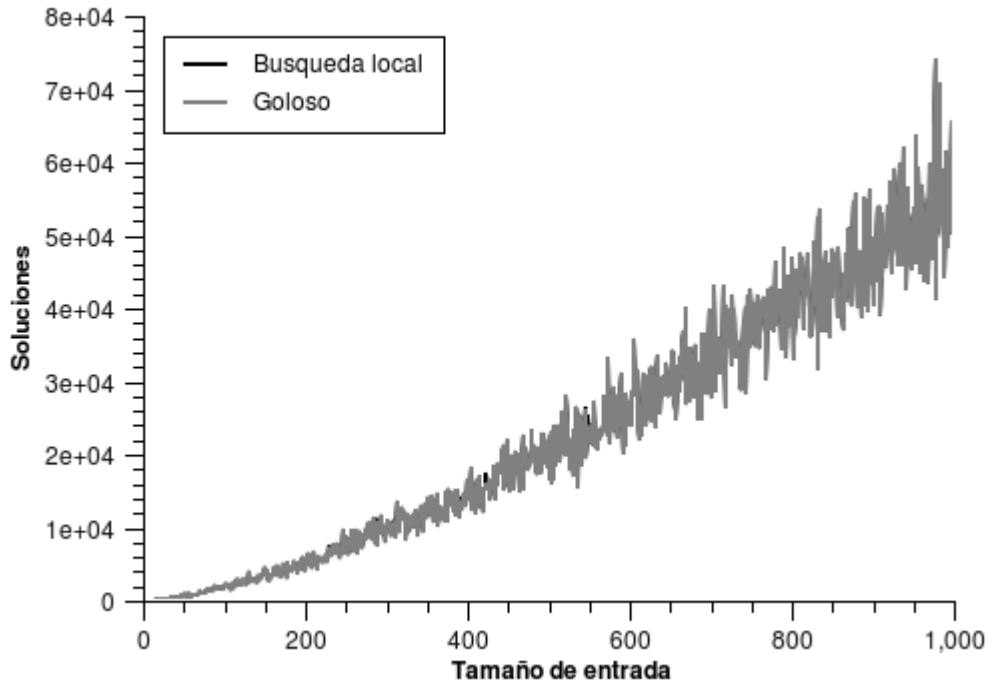
A diferencia del goloso, acá el algoritmo exacto hizo 22004 veces la cantidad de operaciones que hizo la búsqueda local.

Por lo tanto, comparando los dos caso, se mejoró un 2%, pero se hizo 10 veces la cantidad de operaciones que hacia el goloso. Aunque eso parece mucho en la cantidad de operaciones, para grafos de hasta 1000 vértices, la búsqueda local no llegó a tardar más de 15 minutos y por lo tanto, no es algo tan malo.

5.7.2. Heurística Constructiva

Ahora comparamos la solución de la búsqueda local contra la solución golosa. En estos casos, a diferencia de los anteriores, se van a usar grafos de hasta 1000 vértices. Primero comparamos la calidad de la solución entre cada uno de los algoritmos.

Para el primer archivo:



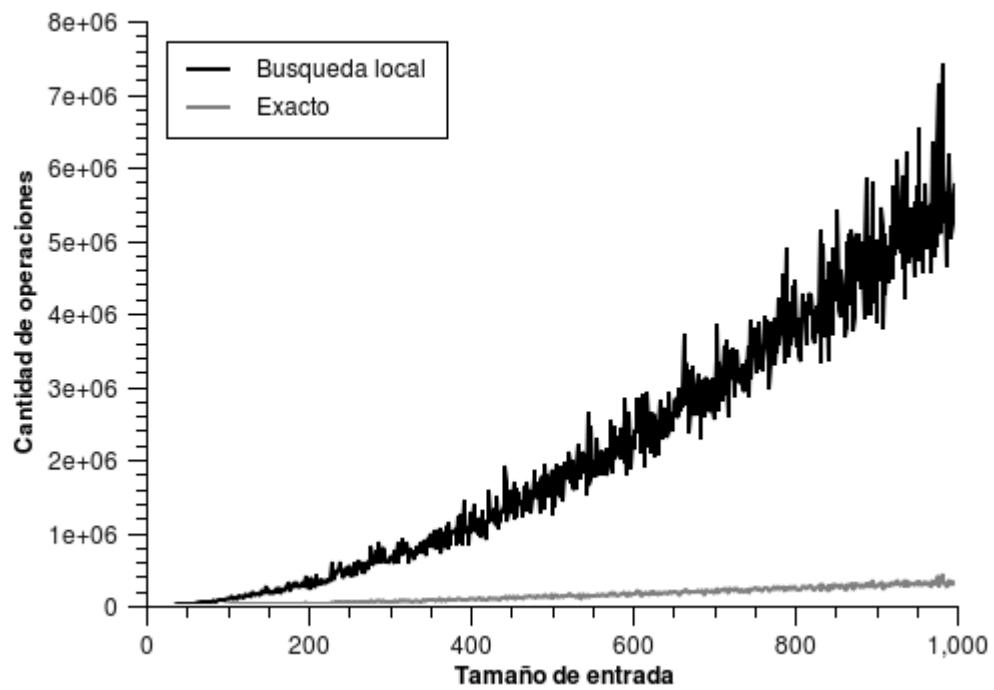
Al igual que para este archivo random, para los otros dos casos, no se puede notar la diferencia entre la solución golosa y la de la búsqueda local. Por lo tanto, muestro la diferencia a través de una tabla en la cual se muestra todos los valores de las soluciones sumados.

Algoritmo \ Archivo Random	1	2	3
Búsqueda local	22710656	22466608	22583301
Goloso	22681285	22420401	22541699

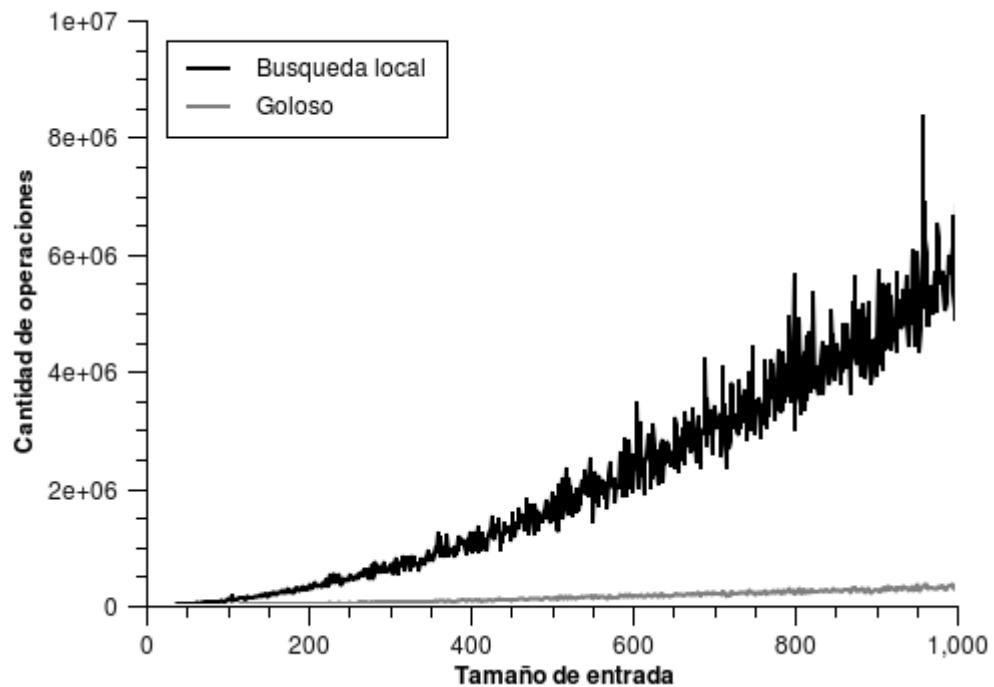
Es importante notar que los valores son diferentes a los calculados en las tablas anteriores, porque en estos casos se usaron grafos de hasta 1000 vértices mientras que anteriormente se usaban grafos de hasta 390 vértices. Como se puede notar en la tabla, la diferencia de la sumatoria de las soluciones es muy chica, cerca del 0,2%^c.

Ahora si comparamos la cantidad de operaciones para cada uno de los archivos generados al azar obtenemos lo siguiente:

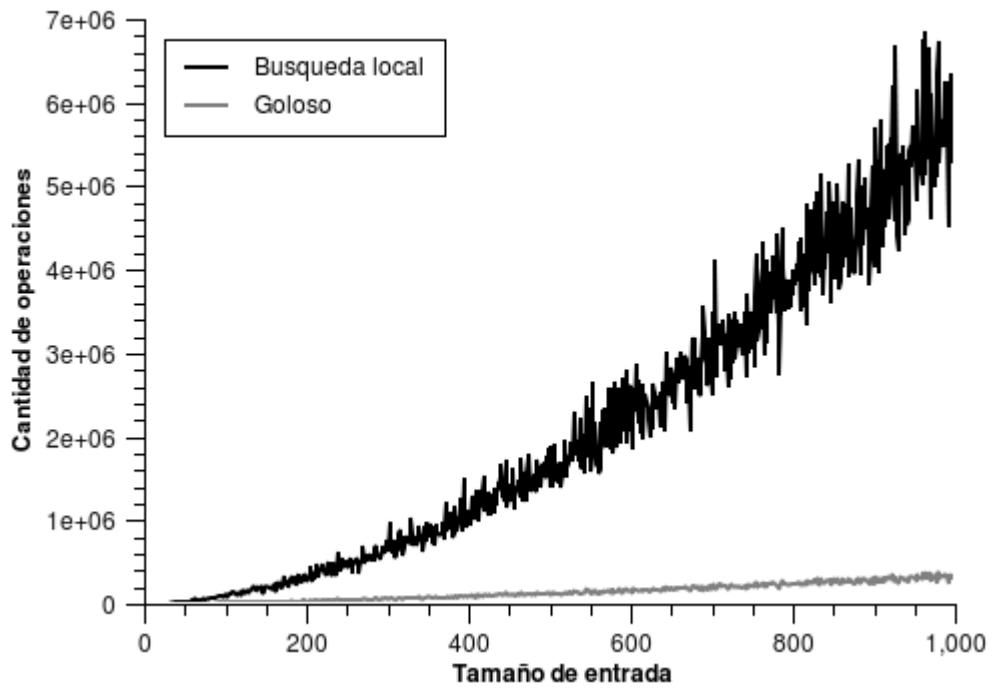
Para el primer archivo:



Para el segundo archivo:



Para el tercer archivo:

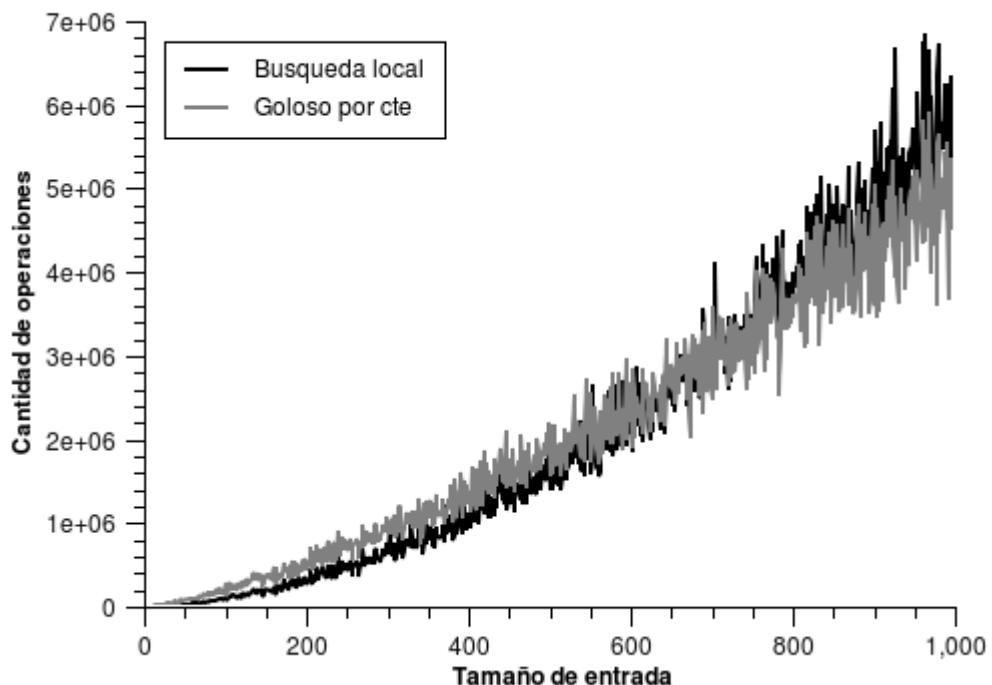


Acá a diferencia de los gráficos de operaciones anteriores, no fue necesario hacer el logaritmo de la cantidad de operaciones. Sin embargo, es claramente mayor la cantidad de operaciones hechas por la búsqueda local cuando se la compara contra el goloso.

En la siguiente tabla figura la sumatoria de la cantidad de operaciones hechas por cada algoritmo:

Algoritmo \ Archivo Random	1	2	3
Búsqueda local	2044450658	2042656469	2053337994
Goloso	136826739	136078290	137008314

De esa tabla se puede deducir que la búsqueda local hace cerca de 15 veces la cantidad de operaciones que hace el goloso. Para demostrar esto, dado que no usamos el logaritmo, gráfico el tercer caso, pero esta vez a la cantidad de operaciones hechas por el goloso las multiplico por 15.



En el mismo se nota que la cantidad de operaciones de la búsqueda local sigue una curva mas pronunciada que el goloso. Es decir, si en vez de haber llegado hasta 1000 vértices en el tamaño de entrada se hubiese llegado hasta 10.000 entonces el valor de la constante también sería mayor. Esto tiene sentido, dado que mientras mayor sean las componentes conexas, mas operaciones se tienen que hacer al sacar o agregar vértices, y también tiene mas iteraciones del ciclo principal.

6. Grasp

6.1. Introducción

La idea de este algoritmo es ejecutar una serie de veces la búsqueda local con una solución constructiva. Cada una de esas iteraciones va a dar una solución y se queda con la mejor de ellas. Para que no se de siempre la misma solución el Grasp introduce dos valores a la creación de la solución incial constructiva:

- α : Es un porcentaje. En la solución constructiva clásica se soluciona el elemento que tiene mejor valor. Donde *mejor valor* depende del criterio de la misma: por ejemplo:

- Menor Grado
- Mayor Beneficio

Lo que introduce ese valor es que se pueda seleccionar un elemento que difiera ese porcentaje del *mejor valor*.

- β : tambien es un porcentaje. El hecho de que se tenga un alpha indica que se va a tener en cuenta mas de un vértice a ser devuelto, y se lo tiene que seleccionar al azar. La idea de este valor es limitar la cantidad de vértices a ser tenidos en cuenta. Por ejemplo, si se pone un $\alpha = 70$, puede ser que se tengan en cuenta todos los vértices del grafo. Justamente para limitar la cantidad de vértices a tener en cuenta se setea el β que es un porcentaje con respecto a la cantidad de vértices del grafo. Por ejemplo, si $\beta = 50$, entonces solo se van a tener en cuenta 50 % de los vértices del grafo para seleccionar uno al azar.

Otro punto importante de GRASP, es que el algoritmo por si nunca termina. Por esto, es que pusimos la misma condición de parada que se puso en la búsqueda local:

- Se llego a una solución que contiene todos los vértices del grafo
- Que el algoritmo termine luego de haber alcanzado un cierto porcentaje de la cantidad de vértices del grafo ¹⁶

6.2. Pseudocódigo

Algorithm 15 grasp(grafo, iteracionesGRASP, alpha, beta, iteracionesBL, cantSacarBL)

Sea *soluciónAnterior* $\leftarrow \emptyset$

Sea *cantIteraciones* $\leftarrow 0$

```

while cantIteraciones  $\leq \frac{\text{iteracionesGRASP}*100}{\text{cantVertices}(\text{grafo})}$  do
    Sea soluciónActual  $\leftarrow$  goloso(grafo, alpha, beta)
    Sea soluciónActual  $\leftarrow$  busquedaLocal(grafo, maxIteracionesBl, cantSacarBL, soluciónActual)
    if peso(soluciónActual) > peso(soluciónMaxima) then
        soluciónMaxima  $\leftarrow$  soluciónActual
    end if
    Sea cantIteraciones  $\leftarrow +1$ 
end while
devolver soluciónMaxima

```

end

6.3. Detalle de implementación

Por un error en la escritura de código, la cantidad de iteraciones no es un porcentaje sino que es un número fijo. De todas formas, este valor se lee del archivo de configuración.

6.4. Complejidad

Por la demostración de la heurística de la búsqueda local sabemos que la misma tiene la complejidad de:

$$O(\text{iteracionesBL} * n^4)$$

donde n es la cantidad de vértices del grafo, y iteracionesBL es la cantidad de iteraciones que hacia el algoritmo. Tambien sabemos que la complejidad del algoritmo goloso es:

$$O(n^2 * \log(n))$$

¹⁶Por un error de implementación, esa variable no es un porcentaje sino que es un valor fijo.

Dentro de cada ciclo del GRASP se hacen tres operaciones:

- Ejecutar el algoritmo goloso.
- Ejecutar la búsqueda local usando como soluciónInicial la del algoritmo goloso.
- Comparar $soluciónActual$ con $soluciónMaxima$, y sabemos que esto se hace en $O(1)$.
- Fijarse si $termine$ se hace en $O(1)$, ya que todas las operaciones que hace la misma son $O(1)$

Por lo tanto, el orden de las operaciones que se realizan dentro del ciclo tienen orden:

$$O(\text{iteraciones}BL*n^4 + n^2 * \log(n)) \in O(\text{iteraciones}BL*n^4)$$

Como el ciclo se repite iteracionesGRASP de veces, entonces el orden final va a ser:

$$O(\text{iteracionesGRASP} * \text{iteraciones}BL * n^4)$$

Ahora calculemos el tamaño de entrada para este problema. El mismo va a estar dado por:

$$\begin{aligned} T &= \log(n) + \sum_{i=1}^n \log(\text{peso}(i)) + \sum_{i=1}^n \sum_{v \in \text{ad}(i)} \log(v) \\ T &\geq \log(n) + \sum_{i=1}^n \log(\text{peso}(i)) + \sum_{i=1}^n \sum_{v \in \text{ad}(i)} \log(v) \\ T &\geq \sum_{i=1}^n \log(\text{peso}(i)) + \sum_{i=1}^n \sum_{v \in \text{ad}(i)} \log(v) \\ T &\geq \sum_{i=1}^n \log(\text{peso}(i)) \\ T &\geq \sum_{i=1}^n 1 \\ T &\geq n \end{aligned}$$

Luego, la complejidad en función del tamaño de entrada es:

$$O(\text{iteracionesGRASP} * \text{iteraciones}BLT^4)$$

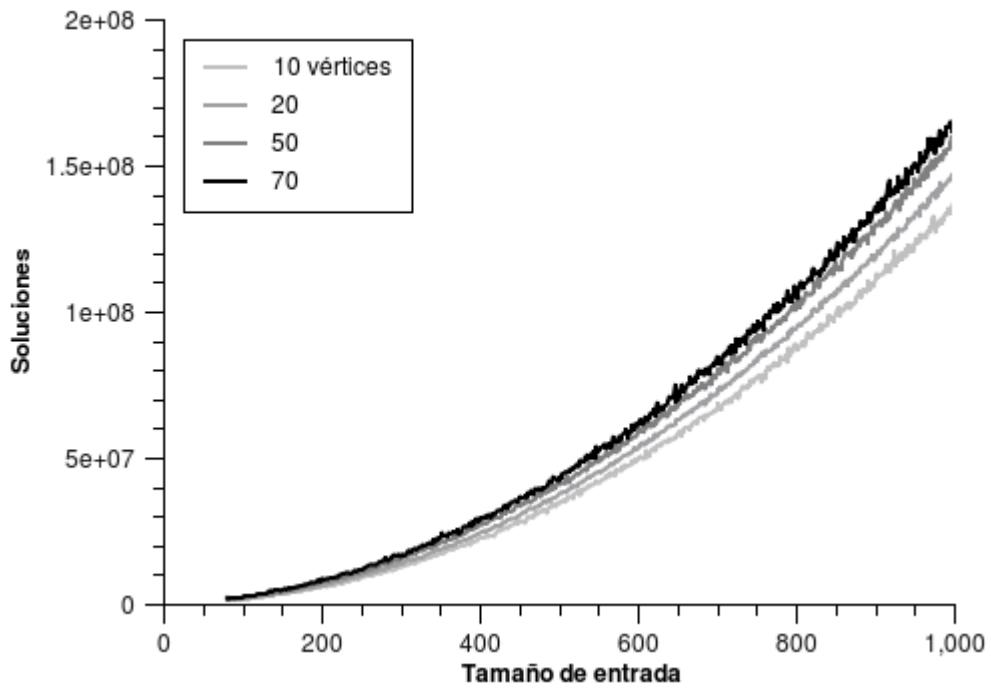
y la misma es polinómica

6.5. Gráficos de tiempo

Antes de hacer el gráfico, deberíamos de calcular cual es el peor caso para este algoritmo. Sin embargo, el mismo depende de tres cosas:

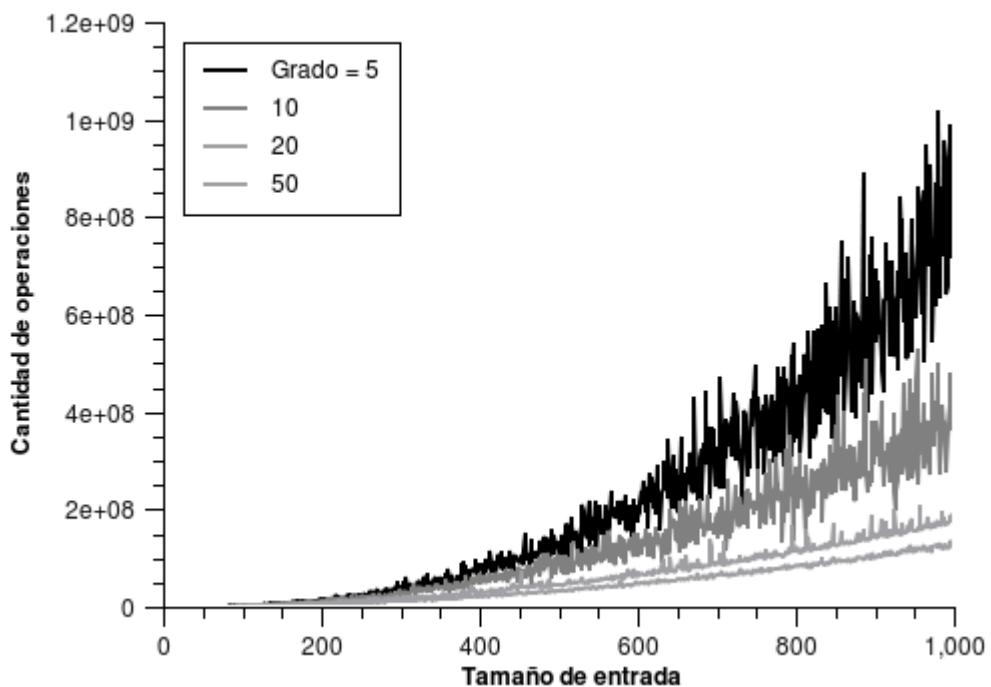
- La solución random generada por el algoritmo constructivo cuando se usan los valores de $alpha$ y $beta$. Esto es importante, ya que mientras mejor sea esa solución generada de forma pseudo random, menor cantidad de operaciones va a haber la búsqueda local.
- No se pudo determinar un peor caso para la búsqueda local. Es decir, no se encontró una entrada, en donde la cantidad de operaciones que se hagan sean máximas. Sin embargo, encontramos casos malos para la misma.

Por lo tanto, veamos como es que se comporta el algoritmo en casos específicos. Empezamos por el caso cuando el grafo tiene una gran cantidad de componentes conexas. En estos casos, el grado y el peso de los vértices se dejaba fijo, pero se variaba la cantidad de vértices por componentes conexas del grafo.



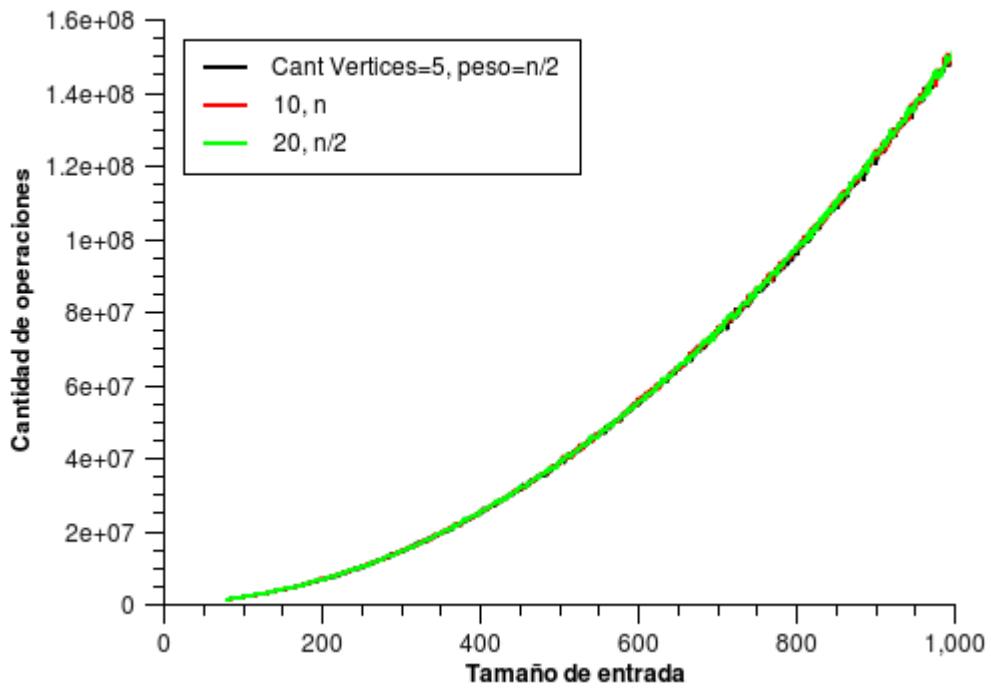
Al igual que en la búsqueda local, en este algoritmo tambien se incremente la cantidad de operaciones a medida que se incrementa la cantidad de componentes conexas.

Ahora vemos como es que se comporta el algoritmo cuando el grado de los vértices cambia, pero no cambia el peso de los mismos.



Tal como era esperado el algoritmo hace menos operaciones cuando mayor es el grado de los vértices. Como se comentó anteriormente esto sucede porque tanto en el goloso como en la búsqueda local se hacen menos operaciones porque se tienen que tener menos casos en cuenta.

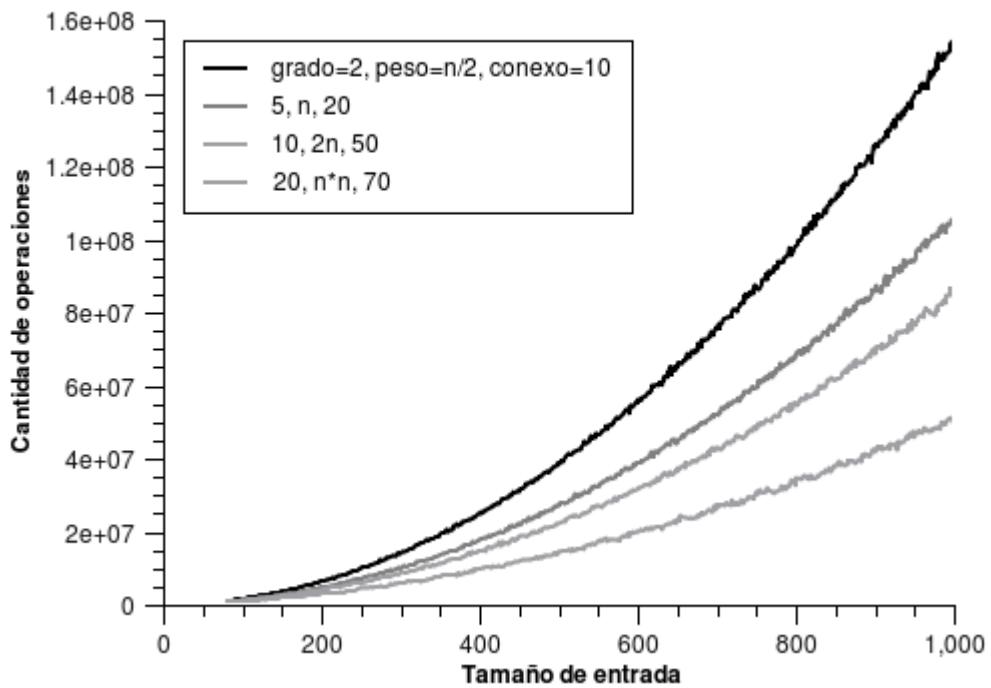
Ahora, vemos el tercer archivo de entrada, donde lo que cambia es el peso de cada vértice, pero donde el grado y la cantidad de vértices por componente conexa se dejan fijos.



En este caso todos los archivos dieron una cantidad similar de cantidad de operaciones, y esto es razonable. Ya que se dejó fijo el grado y la cantidad de vértices por componente conexa, entonces no hay mucha variación mas allá del resultado.

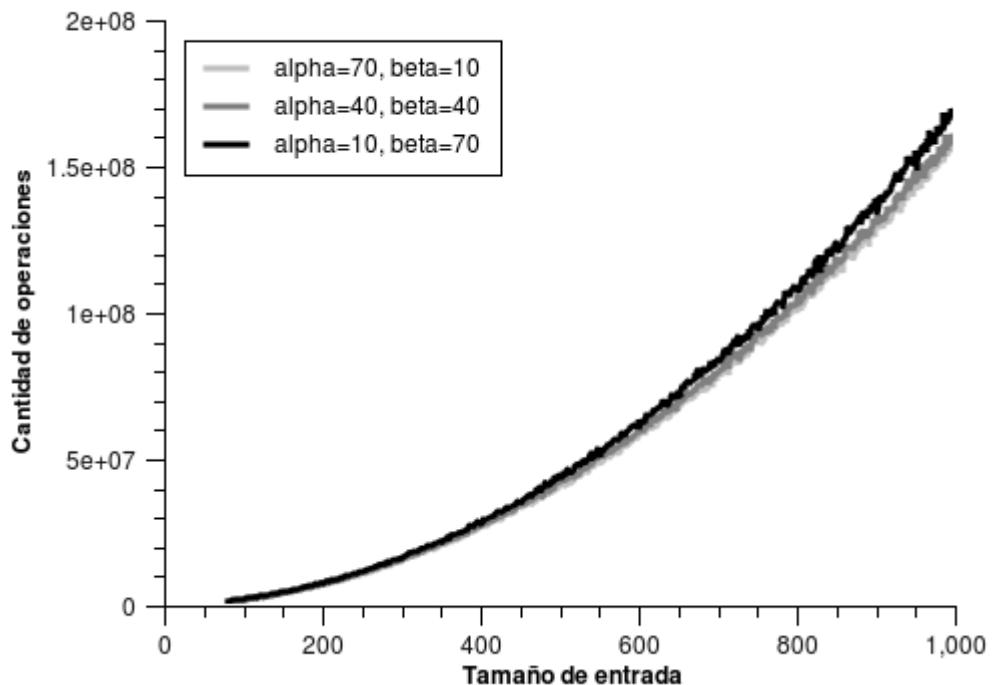
En el gráfico que figura a continuación se graficó archivos cuando se cambia:

- El grado de los vértices
- El peso de los vértices
- La cantidad de vértices por cada componente conexa



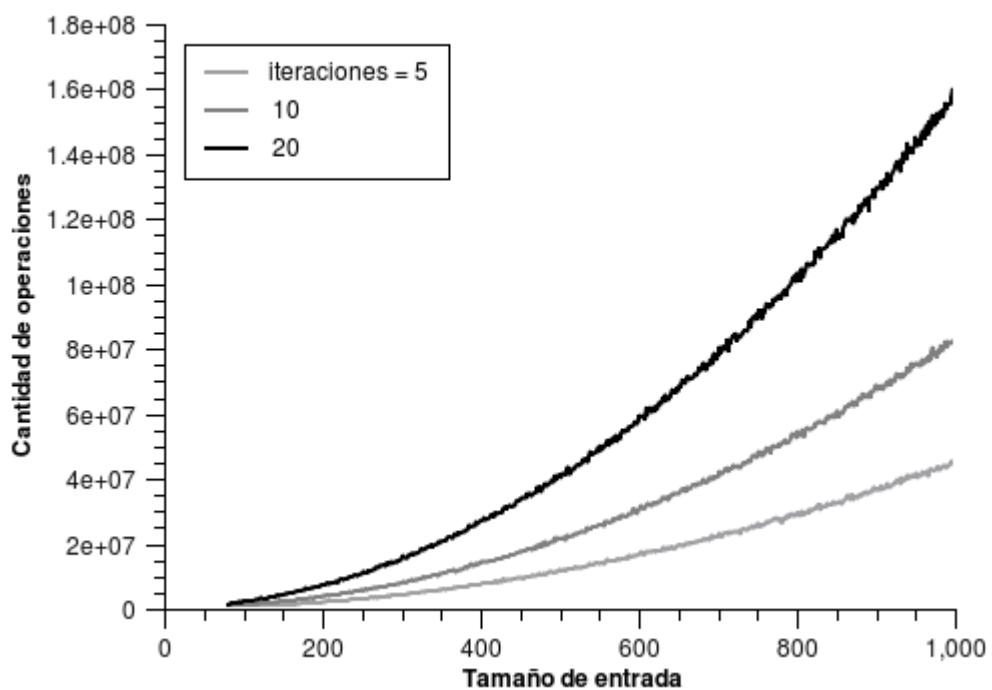
Hasta ahora todos los gráficos hechos tratan sobre los grafos, pero ninguno habla sobre las variables de configuración del GRASP. Así que vemos como es que cambian la cantidad de operaciones teniendo en cuenta los posibles valores para las configuraciones.

Ahora vemos como es que se comporta el algoritmo cuando para un mismo archivo se cambian los valores de α y β . Recordemos que mientras mas grande era β y mas chico era α entonces el goloso hacia mas operaciones. Fuera de variar α y β no se cambian la cantidad de iteraciones que se hace en el GRASP.



Como era de esperarse, mientras bajan los valores de α y suben los de β , se hacen mas operaciones. Por el otro lado esto no necesariamente tenía porque ser así. Si mientras se cambian los valores se cambia la solución que se le pasa a la búsqueda local. Por lo tanto, la misma puede llegar a tardar menos.

Por último, vemos como varía la cantidad de operaciones cuando se cambia el procentaje de la cantidad de iteraciones que hace el GRASP.



Como era de esperarse la cantidad de operaciones crece a medida que crece el valor del archivo de configuración. Como se comentó en *Detalle de implementación*, por un error de programación el valor no terminó siendo el porcentaje sino que directamente se iteraba el valor de ese contador.

6.6. Análisis de la solución

Para este algoritmo se corrieron muchos menos casos que para la búsqueda local, en total se corrieron 27 casos para cada uno de los archivos de entrada.

Las variantes que se usaron en este caso fueron las siguientes:

- α puede ser 10, 40, o 70.
- β puede ser 10, 40 o 70.
- $iteracionesGRASP$ puede ser 5, 10 o 20.

Se ejecutaron cada una de estas variantes usando los mejores valores de configuración para la búsqueda local y la variante de *Peso grado* del algoritmo goloso.

Se busco la mejor configuración de los parametros del GRASP entre todas las ejecuciones y se llego a que la misma es:

- $\alpha = 70$
- $\beta = 10$
- $iteracionesGRASP = 20$

Lo cual tiene cierto sentido porque:

- Si el α es demasiado chico entonces se pueden meter valores al azar que serían demasiados malos para formar una solución inicial.
- No tiene sentido que el valor de β sea grande cuando α también lo es. La razon de esto es que mientras mayor sea el α menos valores van a poder ser agregados a la lista de la cual luego se selecciona un valor al azar. Por lo tanto, como la lista tiene pocos valores, no tiene sentido que se le permita tener muchos.
- La máxima cantidad de iteraciones para que el algoritmo tenga mas posibilidades de encontrar una mejor solución.

De nuevo, como se hizo con la búsqueda local, se comparo para cada uno de los parametros de configuración, para que valores el mismo daba la mejor solución. En este caso se obtuvieron las siguientes 3 tablas:

Valor de α	Suma soluciones	Valor de β	Suma soluciones	Cantidad de iteraciones	Suma soluciones
10	313227828978	10	315019840135	5	312477053619
40	314169876048	40	314221789145	10	314465892766
70	315907009300	70	314063085046	20	316361767941

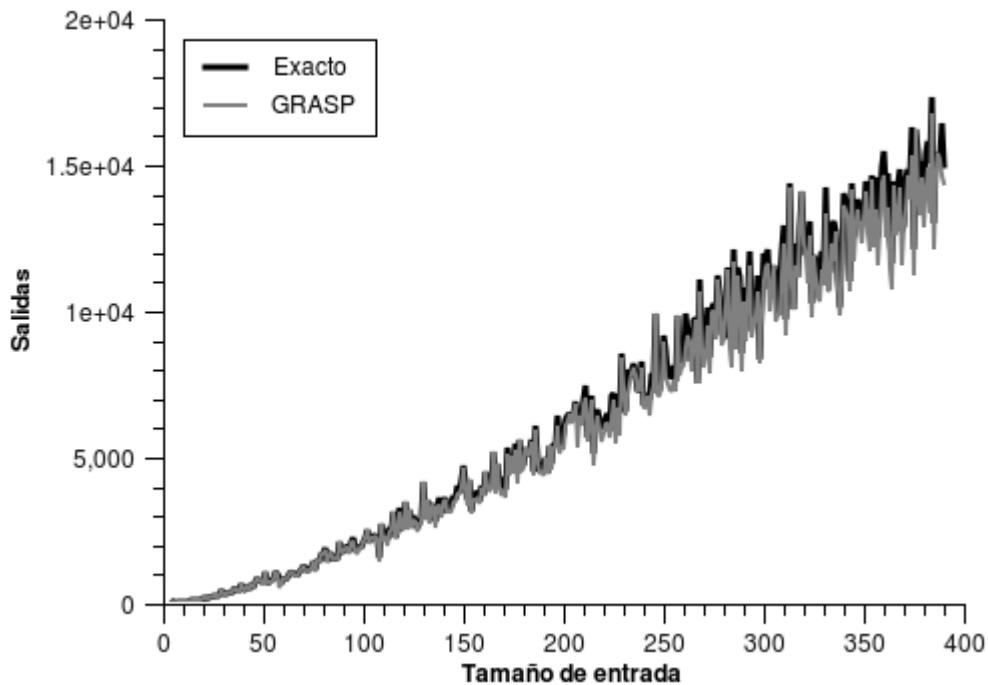
Los valores de esta tabla son razonables con la mejor solución elegida y lo comentado anteriormente de lo hacer que el algrotimo goloso sea demasiado random.

6.7. Comparaciones

Dado que ya sabemos cual de todas las posibilidades del GRASP es el mejor, comparamos contra los algoritmos anteriores. En todos los graficos y tablas que aparecen a continuación se usaron los archivos randoms.

6.7.1. Exacto

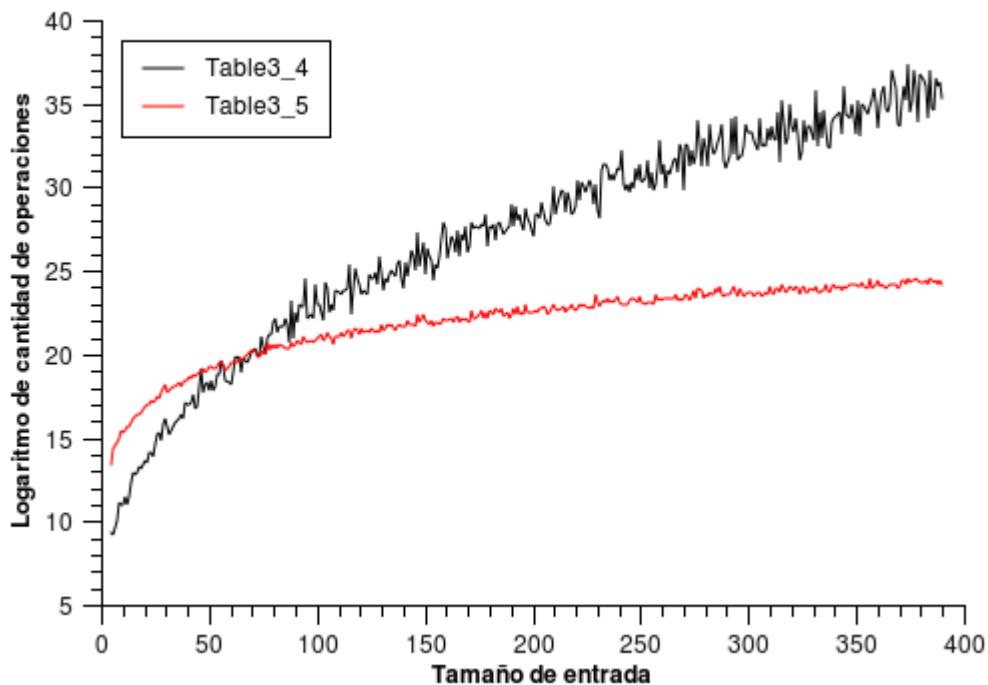
De nuevo se hace el gráfico donde se comprara la solución entre el GRASP y el algoritmo exacto. Para el primer archivo se obtiene el siguiente gráfico:



Como para los otros dos archivos, tambien se obtiene un gráfico similar no tiene sentido hacerlo ya que no se puede distinguir entre el GRASP y la solución exacta. Por lo tanto, vemos esta diferencia en la tabla:

Algoritmo \ Archivo Random	1	2	3
Exacto	2484885	2478760	2453158
Grasp	2424177	2413451	2384865

En este caso, el procentaje de diferencia entre el algoritmo exacto y el GRASP es de 2,5 % en promedio. Por lo tanto, se redujo en cerca de 5 el de la búsqueda local y en 8 el goloso. Ahora vemos la cantidad de operaciones hechas para llegar a ese resultado.



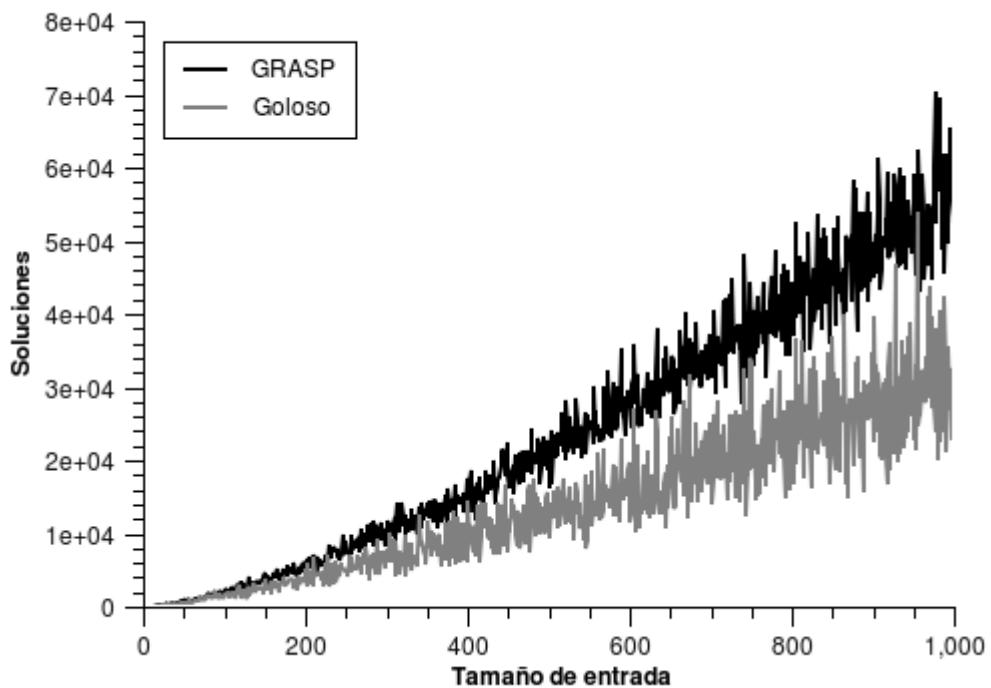
Ese gráfico es solo para el primer archivo, y dan casos similares en los otros dos. Es decir, casos donde el logaritmo de la cantidad de operaciones del exacto queda muy por arriba del logaritmo de las operaciones del GRASP. Por lo tanto, tambien hacemos la comparativa usando una tabla:

Algoritmo \ Archivo Random	1	2	3
Exacto	3318116640309	3118748085286	3248197502833
GRASP	3123032533	3076727332	3096906324

Por ultimo, el algoritmo exacto hizo 1062 veces la cantidad de operaciones que hizo el GRASP. Recoredemos que para la búsqueda local ese valor era de 22004, y por lo tanto uno se animaría a decir que el GRASP hace 20 veces la cantidad de operaciones que la búsqueda local. Lo cual tiene cierto sentido, teniendo en cuenta que se esta iterando tan solo 20 veces.

6.8. Goloso

Aca vamos a comparar las soluciones dadas por el goloso con las soluciones dadas por el GRASP. Para el primer archivo, la diferencia entre las dos soluciones da el siguiente gráfico:



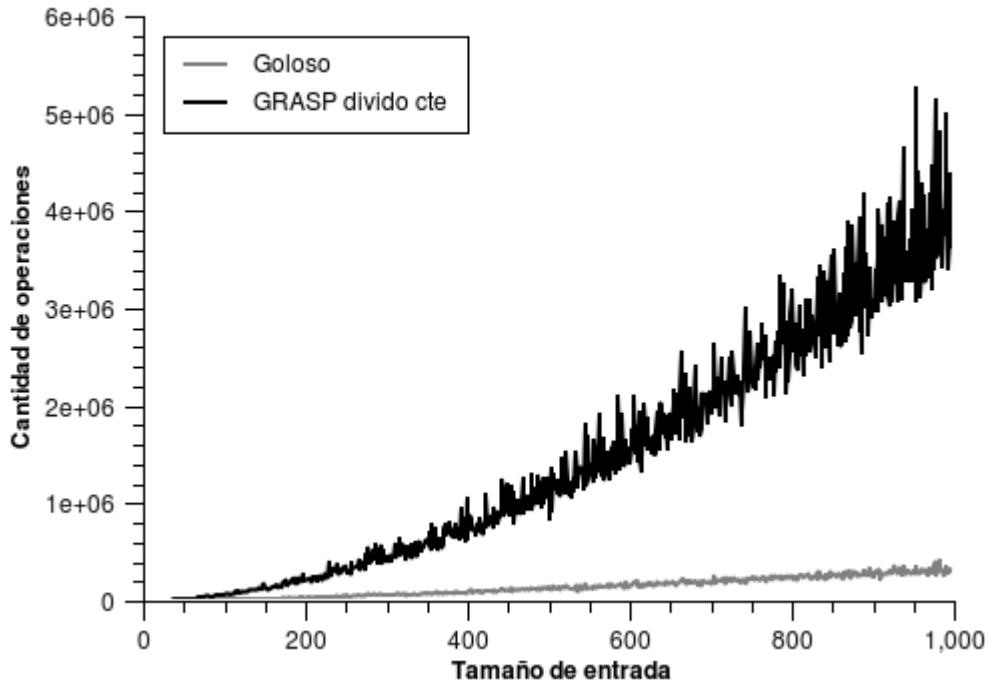
Lo mismo pasa para los otros dos, asi que veo cuanto es que se mejora teniendo en cuenta el promedio.

Algoritmo \ Archivo Random	1	2	3
GRASP	23546886	23415759	23393871
Goloso	22681285	22420401	22541699

Por lo tanto, en promedio, el algoritmo de GRASP es cerca de un 4% mejor que la solución del goloso. Es decir, la solución del GRASP es un 104% de la solución del goloso. Aunque me resulta raro que un 4% genere tal diferencia en el gráfico.

Ahora veamos la cantidad de operaciones que se hicieron para realmente mejorar ese 4%. En el gráfico que figura a continuación fue necesario dividir a la cantidad de operaciones del GRASP por una constante¹⁷ para que se pudiese notar donde es que se encuentran la cantidad de operaciones del goloso.

¹⁷El valor de esa constante es 30



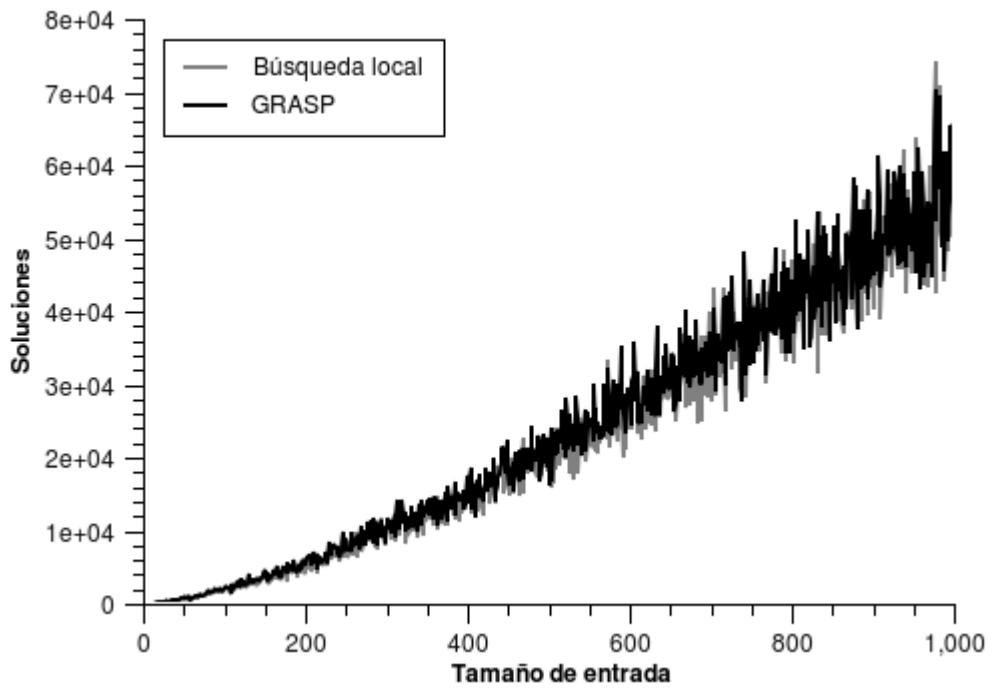
Por último calculamos la diferencia aproximada de la cantidad de operaciones usando la tabla de la suma de la cantidad de operaciones para todas las soluciones en los 3 archivos:

Algoritmo \ Archivo Random	1	2	3
GRASP	42633410488	42626896758	42562600129
Goloso	136826739	136078290	137008314

En este caso el GRASP hace 311 veces la cantidad de operaciones que hace el algoritmo goloso. Si se tiene en cuenta que a mayor cantidad de iteraciones es posible que se obtenga una mejor solución, y considerando que el tiempo promedio de la ejecución del GRASP es relativamente corto comparado contra el exacto, considero que es mejor opción usar el GRASP que el algoritmo goloso.

6.8.1. Búsqueda local

Si comparamos las soluciones de la búsqueda local contra la del GRASP se obtiene el siguiente gráfico:

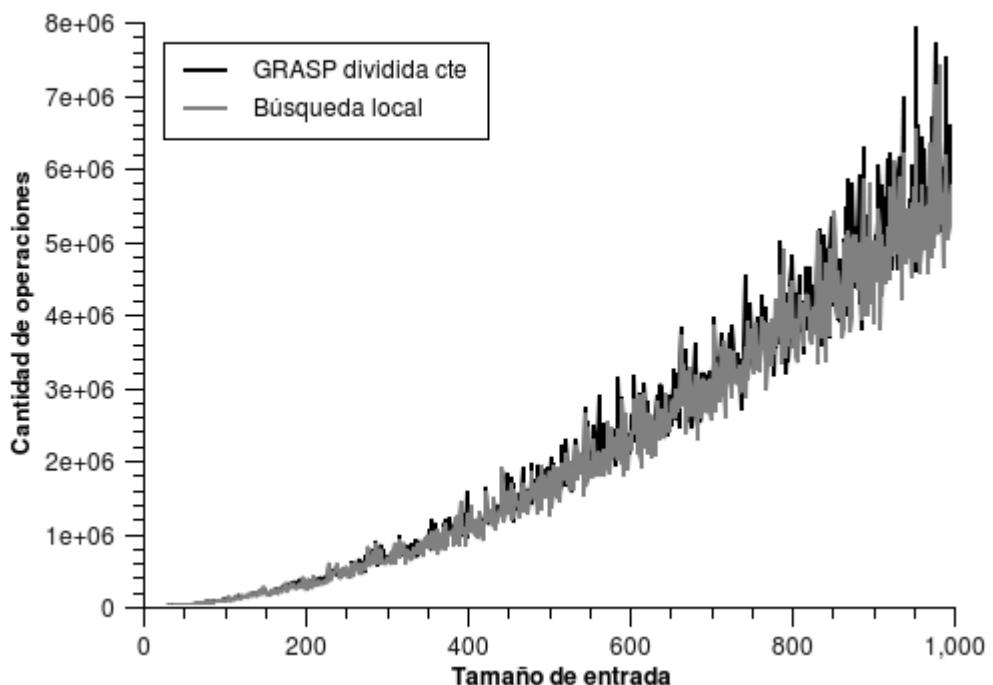


Como en el gráfico la diferencia entre los dos algoritmos es mínima, entonces calculo el porcentaje de la diferencia usando la misma tabla pero esta vez con los datos de la búsqueda local:

Algoritmo \ Archivo Random	1	2	3
GRASP	23546886	23415759	23393871
Búsqueda local	22710656	22466608	22583301

En este caso la mejora entre los dos algoritmos es cerca del 4 %. Sin embargo, en este caso las soluciones son mucho mas parecidas por lo que me hace pensar que el gráfico donde se comparan las soluciones entre el goloso y el grasp esta mal.

Por último se compara la diferencia de la cantidad de operaciones que hacen los algoritmos.



En el gráfico anterior, la cantidad de operaciones del GRASP estan dividas sobre 20, lo cual tiene sentido ya que el algoritmo principal itera 20 veces. Ahora si calculamos el promedio de la cantidad de operaciones, obtenemos el siguiente resultado:

Algoritmo \ Archivo Random	1	2	3
GRASP	42633410488	42626896758	42562600129
Búsqueda local	2044450658	2042656469	2053337994

Por lo tanto, en promedio GRASP hace 20 veces mas operaciones que la búsqueda local.

7. Apendice

7.1. Ejecución de los algoritmos

Para ejecutar el algoritmo, se tiene que escribir desde una linea de comandos lo siguiente:

```
java -jar Tp3.jar
```

Es necesario que en la misma carpeta en donde se encuentra el Tp3.jar, se encuentre tambien el archivo de entrada y el archivo de configuración del mismo.

7.2. Sobre la ejecucion del programa

El ejecutable para todos los algoritmos es el mismo. El algoritmo, con las diferentes variantes, depende de un archivo de configuración llamado *configuracion.ini* que se tiene que encontrar en donde se encuentra el ejecutable.

Es importante notar que este archivo solo se lee una unica vez, y eso es antes de ejecutarse cualquier algoritmo.

El archivo de configuración tiene las siguientes opciones:

Nombre de opción	Descripción
ALGORITMO	Indica cual es el algoritmo que se va a ejecutar. Esta variable no debería de ser cambiada ya que el valor de la misma es el correspondiente a la carpeta ¹⁸
CONSTRUCTIVO	<p>En caso de que se ejecute cualquier heuristica, indica cual es el algoritmo constructivo que se va a usar para la misma. Sus posibles valores son los siguiente:</p> <ul style="list-style-type: none"> ■ 0 = Heuristica Golosa, en donde el vértice agregado es el de mayor peso. ■ 1 = Heuristica Golosa, en donde el vértice agregado es el de menor grado. ■ 2 = Heuristica Golosa, en donde el vértice agregado es el de mayor beneficio¹⁹ ■ 3 = Algoritmo random. <p>Este valor es usado cuando el algoritmo especificado a ejecutar es el constructivo, o cuando se esta ejecutando el GRASP o busqueda local, para obtener una solución inicial.</p>
ALPHA	Es usado en las heurísticas golosas. Es el PROCENTAJE de diferencia que tiene que haber entre la solución optima y el valor del vértice actual para que sea tenido en cuenta. Se lo usa para el GRASP. Por una limitación de la implementación, si este valor es 100, entonces significa que directamente se tiene que devolver el valor optimo (por lo tanto no es random).
BETA	Es usados en las heurísticas golosas cuando se ejecutan desde el GRASP. Es el PROCENTAJE de la cantidad de vértices que pueden llegar a ser tenidos en cuenta cuando cumple que el valor del vértice actual se diferencia de un α del valor optimo.
ALGORITMO_AGREGAR	<p>Se lo usa para la busqueda local. Es el algoritmo que se va a usar para intentar agregar un vértice a la solución actual. Los posibles valores para el mismo son:</p> <ul style="list-style-type: none"> ■ 0 = Agregar en función del peso. Agrega desde los mas pesados a los menos pesados. ■ 1 = Agregar en función del grado. Agrega desde los que tienen menor grado a los que tienen mayor grado. ■ 2 = Agregar en función del beneficio. Agrega desde los que tienen mayor beneficio a los que tienen menor beneficio. ■ 3 = Algoritmo random. Dado todos los vértices posibles a agregar, los selecciona de forma random.
ALGORITMO_SACAR	<p>Se lo usa en la busqueda local. Indica cual algoritmo se va a usar para sacar vértices de la solución actual. Los posibles valores de esta opción son:</p> <ul style="list-style-type: none"> ■ 0 = Sacar en función del peso. Saca desde los menos pesados a los mas pesados. ■ 1 = Agregar en función del grado. Agrega desde los que tienen mayor grado a los que tienen menor grado. ■ 2 = Agregar en función del beneficio. Agrega desde los que tienen menor beneficio a los que tienen mayor beneficio. ■ 3 = Algoritmo random. Dado todos los vértices posibles a sacar, los elije de forma random.
CANT_NODOS_SACAR	Es usado por la busqueda local. Indica la cantidad de los vértices de la solución actual que se tienen que sacar. Es un porcentaje sobre la cantidad de vértices de la componente conexa que sobre la cual se esta ejecutando el algoritmo. Si el procentaje diese menor que 1, entonces solo se saca un vértice de la solución actual.
ITERACIONES_BL	Es usado por la busqueda local. Indica cuantas iteraciones tiene que hacer el algoritmo antes de verificar si mejoro lo necesario como para poder seguir ejecutandolo.
MEJORA_PEDIDA_BL	Es usado por la busqueda local. Dada el valor de la solución anterior y una solución actual indica cual es el procentaje de la solución anterior que tiene que haber mejorado la solución actual para poder seguir ejecutando el algoritmo.
MEJORA_PEDIDA_GRASP	Como MEJORA_PEDIDA_BL pero para el GRASP.
ITERACIONES_GRASP	Como ITERACIONES_BL pero para el GRASP.

7.3. Cantidad de ramas en el algoritmo exacto

Sabemos que la cantidad de iteraciones, van a ser la cantidad de ramas, y la longitud de cada una de ellas del árbol de todas las soluciones posibles. Primero, vemos la cantidad de ramas que tiene el mismo. Sabemos que la cantidad de ramas, empezando desde un vértice i van a ser:

$$\text{cantRamas}(i) = \begin{cases} 1 & \text{if } i=n, \\ \sum_{k=i+1}^n \text{cantRamas}(k) & \text{if } i \neq n. \end{cases}$$

Si escribimos la ecuación al revés, tenemos que:

$$\text{cantRamas}(i) = \begin{cases} 1 & \text{if } i=1, \\ \sum_{k=1}^{i-1} \text{cantRamas}(k) & \text{if } i \neq 1. \end{cases}$$

Demostramos por inducción que:

$$\sum_{i=1}^n \text{cantRamas}(i) \leq 2^n$$

Primero el caso base ($n=1$).

$$\begin{aligned} \sum_{i=1}^1 \text{cantRamas}(i) &= 1 \\ \sum_{i=1}^1 \text{cantRamas}(i) &\leq 2 \end{aligned}$$

Ahora vemos el caso inductivo. Suponemos que se cumple para $i = 1$ hasta j . Por lo tanto, nuestra hipótesis es que:

$$\forall i \leq j, \sum_{r=1}^i \text{cantRamas}(r) \leq 2^i$$

Queremos ver que es lo que sucede cuando $i=j+1$. Para ese caso:

$$\begin{aligned} \sum_{i=1}^{j+1} \text{cantRamas}(i) &= \sum_{i=1}^{j+1} \text{cantRamas}(i) \\ \sum_{i=1}^{j+1} \text{cantRamas}(i) &= \sum_{i=1}^j \text{cantRamas}(i) + \sum_{i=1}^j \text{cantRamas}(i) && (\text{Por definición de cantRamas}) \\ \sum_{i=1}^{j+1} \text{cantRamas}(i) &\leq 2^j + 2^j && (\text{Por HI}) \\ \sum_{i=1}^{j+1} \text{cantRamas}(i) &\leq 2^{j+1} \end{aligned}$$