DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

Heuristicas

19 / 12 / 2014

Algorítmos y Estructura de Datos III

Integrante	LU	Correo electrónico
Ariel Paez	668/09	twizt.hl@gmail.com
Patricio Capanna	866/10	patriciocapanna@gmail.com
Augusto Lamastica	954/11	mascittija@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. k-Partición de Mínimo Peso (k-PMP)

1.1. Introducción

En este último Trabajo Práctico de la materia vamos a afrontar la resolución del problema de encontrar una k-Partició de Peso Mínimo (de ahora en más k-PMP), el cual se trata de un problema que pertenece a la clase NP-Completo[1] y por lo tanto no se conoce un algoritmo en tiempo polinomial que lo resuelva.
Por tal motivo, vamos a buscar una solución exacta mediante un BackTraking y luego realizar algoritmos aplicando distintas heurísticas (Golosa, Busqueda Local, GRASP) para analizar las ventajas y desventajas de cada una.

1.2. Problema

Dado un grafo simple $G = (V, E)$ y un entero k, se define una k-partición de G como una partición de V en k conjuntos de vértices $V_1, \ldots, V_k$. Las aristas intrapartición de una k-partición, son aquellas aristas de G cuyos extremos se encuentran en un mismo conjunto de la partición, es decir, una arista $uv \in E$ es intrapartición si existe un $i \in 1, \ldots, k$, tal que $u, v \in V_i$. Dada una función de peso $\omega : E \to \mathbb{R}_+$, definida sobre las aristas de G, el peso de una k-partición es la suma de los pesos de las aristas intrapartición.

El problema k-PMP consiste en encontrar, dado un grafo simple $G = (V, E)$, un entero k y una función $\omega : E \to \mathbb{R}_+$, una k-partición, que tenga a lo sumo k subconjuntos, de V con peso mínimo.

1.3. Ejemplos

En el seiguiente ejemplo vamos a ver lo que sería una partición cualquiera del grafo y luego la k-PMP

Se el siguiente grafo:

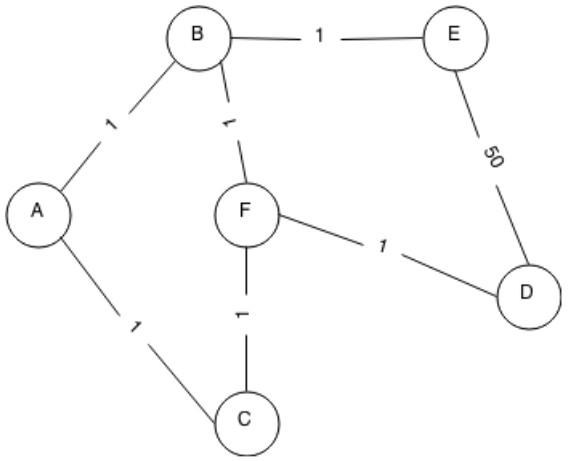

Figura 1: Gráfo Original.

[1] http://en.wikipedia.org/wiki/Minimum_k-cut

Podemos realizar la siguiente 2-partición

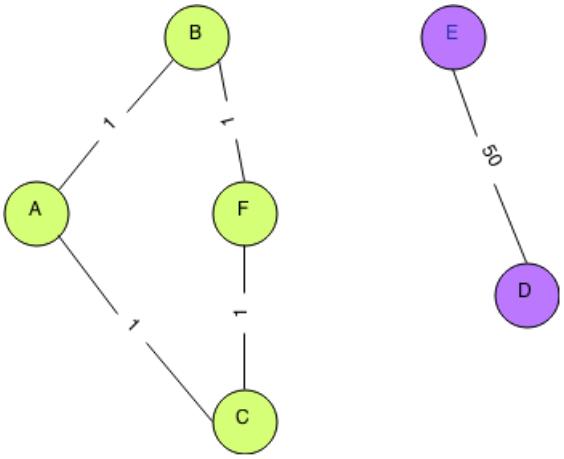

Figura 2: 2-Partición.

Se puede apreciar fácilmente que esta partición no cumple que sea mínima dentro de todas las 2-particiones posibles. Pues el peso total es de 54. Y por ejemplo existe la siguiente 2-particion en donde la suma del peso de sus aristas intraparticiones es igual a 0.

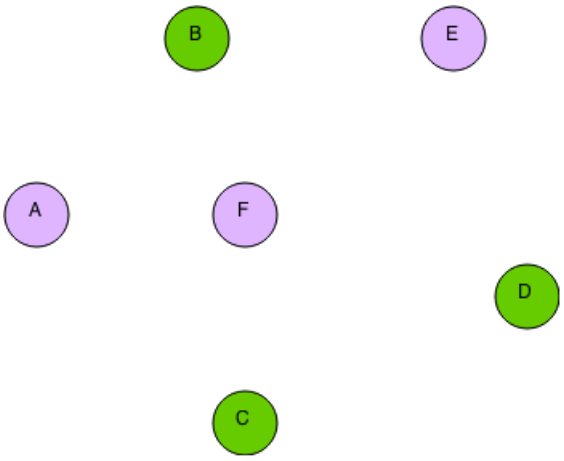

Figura 3: 2-Partición de peso mínimo.

1.4. Testing

Luego de implementar todos los algoritmos pertinentes para la resolusión del problema k-PMP, vamos a realizar distintos tipos de tests y análisis.

Por un lado vamos a generar tres familias de grafos de las cuales conocemos sus soluciones para poder medir tanto el rendimiento del algoritmo exacto y las distintas heurísticas, como la distancia de las soluciones de las heurísticas contra las del algoritmo exacto. Para esto tenemos un generador de grafos en el cual podemos configurar la cantidad de nodos y la cantidad de

aristas totales con pesos aleatoreos, siendo 0 % un grafo con n nodos aislados y el 100 % un grafo completo de n nodos. Con este generador vamos a generar las tres familias de grafos que se diferencian básicamente por su cantidad de aristas. Siendo estos tres grupos, grafos con el 15 % de aristas, 50 % y 100 % y vamos a variar n desde 6 a 15 por ser un numero manejable para el algoritmo exacto.

Luego para cada heuristica vamos a intentar identificar familias de grafos para las cuales se comporten razonablemente bien y ver si existen algunas familias para las cuales las mismas los resultados sean inadmisibles.

2. Comparativas k-PMP

2.1. k-PMP y Ej3-TP1

El ejercicio 3 del trabajo práctico 1 "Biohazard", consistía en distribuir químicos en camiones para que puedan ser transportados de una central de químicos a otra.
El problema estaba en que hay ciertos químicos que no pueden transportarse juntos pues presentan un gran nivel de peligrosidad. Por lo que se determinó un coeficiente de peligrosidad que determina cuan peligroso es transportar dos ciertos químicos juntos, y un umbral de peligro que cada camión no debe sobrepasar en la suma de coeficientes de peligrosidad que existe entre los quimicos que transporta.
Con esto y dado un conjunto de productos con sus coeficientes de peligrosidad se buscaba distribuirlos de forma de minimizar la cantidad de camiones utilizados, obviamente sin que cada uno de estos sobrepase el umbral de peligro.

El problema es básicamente el mismo que k-PMP, con la diferencia de que en k-PMP no existe umbral alguno, pero existe un límite en la cantidad de camiones, k, de manera que lo que se busca optimizar no es la cantidad de camiones, sino (llevando el ej Biohazard a k-PMP) la peligrosidad total de los k camiones.

2.2. k-PMP y Coloreo de Vértices

Se llama coloreo a la asignación de colores a los vértices de un grafo tal que dos vértices que compartan la misma arista tengan colores diferentes.
Un coloreo que usa a lo sumo k colores se llama k-coloreo. Un grafo al que se le puede asignar un k-coloreo se lo llama k-coloreable.
Por último, un subconjunto de vértices con el mismo color asignado en el coloreo, se llama clase de color y cada clase forma un conjunto independiente. De esta forma podemos definir un k-coloreo como una particion del conjunto de aristas del grafo en k conjuntos independientes.

(Para la siguiente explicación vamos a suponer que el grafo no posee aristas de peso 0 o en su defecto eliminarlas).
Como se explicó antes, el problema k-PMP consiste en particionar las aristas en k conjuntos de modo que la suma de los pesos de las aristas intrapartición sea mínima.
Cuando el peso de cada k-partición resulta en 0, quiere decir que dentro de cada particion no existen aristas intrapartición, lo cual quiere decir que se pudo dividir el grafo en k subconjuntos independientes de vértices, o en otras palabras, tenemos un grafo k-partito.
Ahora si a cada k-particion se le asigna un color, resulta en un coloreo válido del grafo dado.

2.3. k-PMP en la vida real

Con k-PMP se pueden modelar aquellos problemas en los que hace falta agrupar un conjunto de cosas de forma que se minimice la relación entre estos. Por ejemplo:

Arca de Noe
Suponemos que tenemos un arca con k sectores, en los cuales debemos repartir todos los animales de forma de minimizar el riezgo de riña entre ellos.
Para esto asignamos un valor de peligrosidad a los pares de animales que tienen alguna posi-

bilidad de pelearse entre sí.
En este caso los animales pueden ser representados con nodos y el valor de peligrosidad entre ellos esta dado como el peso de la arista que los une.

Campus Universitario
Se desea construir un campus universitario en el que se edificarán k pabellones, y se pretende poder distribuir un total de n carreras en los pabellones de forma que aquellas carreras que esten directamente relacionadas se destinen a un mismo edificio. Para esto se define un valor x a la relación entre cada carrera, el cual determina cuan relacionadas están entre ambas, siendo 0 el coeficiente que identifica la mayor relación.

Torneo de Futbol Extraterrestre
Cayó un extraterrestre en la tierra y tomo el cuerpo de un alto dirigente de Futbol de la FIFA sin saberlo. Resulta que la actividad del humano al cual invadió era organizar k torneos cortos de partidos amistosos entre selecciones que maximicen la ganancia de los eventos.
Para tal motivo se tiene estipulado la ganancia que generan los equipos al competir entre ellos, siendo la mínima ganancia >0.
Como viene de otro planeta se le ocurre aplicar k-pmp (que con su tecnología pudieron demostrar que P=NP(cuando N = 1 (?))) tomando como pesos de las aristas la inversa de la ganancia, con lo cual equipos que generan mucho dinero, están unidos por una arista de poco costo.

3. Algoritmo exacto: Backtraking

3.1. Algoritmo

Para resolver el problema del k-PMP vamos a utilizar un algoritmo que se basa en la técnica de Backtracking. Este recorre todas las posibles particiones del Grafo y basta con devolver aquella que cumpla que la sumatoria de los pesos de sus aristas intraparticiones sea mínimo y tenga menos de k-Particiones. Luego se aplican algunas podas para mejorar un poco el tiempo de ejecución de esta solución, como guardar la mejor solución hasta el momento o no calcular aquellas que tenga mas de k particiones pues no pueden ser solución del problema.

Básicamente el algoritmo exacto va iterando los nodos y colocandolos en las particiones para ir verificando si voy generando una solución mejor. En el algoritmo sin podas agrego un nodo a una partición y calculo la sumatoria de las aristas intrapartición del grafo que si es la mínima hasta el momento, la guardo como solución del problema, para luego llamar recursivamente a la función con el próximo nodo. Luego de salir de la recursión saco el nodo que habia colocado en la particion y pruebo colocandolo en la próxima partición para reiterar los cáluclos y ver si puedo disminuir el peso de la sumatoria de las aristas intrapartición.

COMBINAR($list < Particion > particiones, VerticeverticeAUbicar, doublepesoAcumulado$)

```
1  for (p in particiones)
2  pesoViejo = p.getPeso()
3  pesoNuevo = pesoDelGrafoConVerticeEnParticion(p,verticeAUbicar)
4  if pesoAcumulado <= pesoGrafoConModificacion
5      then continue
6      else
7  p.setPeso(pesoNuevo)
8  pesoAcumulado = pesoGrafoConModificacion
9  combinar(particiones, proximoVertice(verticeAUbicar), pesoAcumulado)
10 pesoAcumulado = pesoGrafoConModificacion − pesoNuevo
11 p.sacarNodo(verticeAUbicar)
12 p.setPeso(pesoOld)
```

3.1.1. Podas

La poda que aplicamos al algoritmo de BackTraking es bastante sencilla pero efectiva. Básicamente me guardo la última mejor solución. Y si en el camino de recorrer las particiones, al agregar un nodo en alguna de las particiones, este genera una peor solución que la que ya tengo calculada, directamente la descarto y paso a intentar colocar este nodo en la proxima partición.

En la sección de experimentación vamos a mostrar los resultados de correr los algoritmos con o sin poda.

3.2. Código Relevante

El esta sección del código pondremos partes del código relevante.

```
class Arista {
  public:
    Arista(Vertice v, Vertice w);
    Arista(Vertice v, Vertice w, double peso);
    Vertice getVertice1();
    Vertice getVertice2();
    double getPeso();

  private:
    Vertice v;
    Vertice w;
    double peso;
};

class Grafo{
  public:
    Grafo(int n, list<Arista> aristas);
    Arista* getArista(Vertice v, Vertice w);
    Arista** getAristas(Vertice v);
    list<Arista*>* getAristas();
    int getCantVertices();

  private:
    Arista *** ady;
    list<Arista*>* aristaList;
    int n; \\ |V| := cantidad de vertices
};

class Particion{
  public:
    Particion(int nro);
    double cuantoPesariaCon(Grafo *G, Vertice vertice);
    double cuantoPesariaSin(Grafo *G, Vertice vertice);
    void agregar(Grafo *G, Vertice vertice);
    void agregarSinActualizarPeso(Vertice vertice);
    void quitar(Grafo *G, Vertice vertice);
    void quitarUltimo(Grafo *G);
    void quitarUltimoSinActualizarPeso();
    int getNro();
    double getPeso();
    void setPeso(double peso);
    list<Vertice> getVertices();

  private:
    int nro;
    list<Vertice> vertices;
    double peso;
    double pesoConVerticeX;
    Vertice verticeX;
};

void Exacto::combinar(list<Particion> &k_particion, Vertice verticeAUbicar,
    double pesoAcumulado){

  //Lo siguiente equipara a decir: Si no hay mas quimicos para cargar..
  if (verticeAUbicar == G->getCantVertices()) {
    if (pesoAcumulado < this->mejorPeso){
      this->mejorPeso = pesoAcumulado;
      this->mejorKParticion = k_particion;
```

```
      if (mostrarInfo) mostrarPotencialSolucion(this->mejorKParticion, this->
          mejorPeso);
    }
    return;
  }

  list<Particion>::iterator itParticion;
  for (itParticion = k_particion.begin(); itParticion != k_particion.end();
      itParticion++){

    double pesoOld = itParticion->getPeso();
    double pesoNew = itParticion->cuantoPesariaCon(G, verticeAUbicar);
    double difPeso = pesoNew - pesoOld;

    if (this->podaHabilitada && (this->mejorPeso <= pesoAcumulado+difPeso))
      continue;

    // 'agregar' no vuelve a calcular el peso. Ya lo calcul'o en
        cuantoPesariaCon donde se cachea.
    itParticion->agregar(G, verticeAUbicar);
    pesoAcumulado += difPeso;

    combinar(k_particion, verticeAUbicar+1, pesoAcumulado);

    pesoAcumulado -= difPeso;
    itParticion->quitarUltimoSinActualizarPeso();
    itParticion->setPeso(pesoOld);
  }


  // Si hay menos de k particiones el vertice se puede ubicar en una particion
       nueva.
  if (k_particion.size() < k) {
    Particion particionNueva(k_particion.size());
    particionNueva.agregar(G, verticeAUbicar);
    k_particion.push_back(particionNueva);
    //No hace falta fijarse que pesoAcumulado sea menor a mejorPeso porque el
        pesoAcumulado no se modifica al hacer una particion nueva sin aristas.
    combinar(k_particion, verticeAUbicar+1, pesoAcumulado);
    k_particion.pop_back();
  }

}
```

3.3. Complejidad

El analisis de complejidad lo vamos a realizar en base a la cantidad de particiones posibles en k-subconjunto. Ya que si no aplicamos ninguna poda las estaríamos generando todas para verificar cual de ellas es la mínima.
Para el primer vertices v_1, tenemos una única partición posible en donde ubicarlo, que la vamos a denominar p_1.
Para el segundo vertice v_2, lo podemos ubicar tanto en la primer partición como en una nueva llamemosla p_2.
Para el tercer nodo v_3, si ya habíamos ubicado a v_1 y v_2 en p_1, podemos colocarlo también en p_1 o bien crear una nueva partición p_2 que no sería la misma que la anterior pues esa no existiría producto que v_1 y v_2 se encuentran en p_1, pero en cambio si ubicamos a v_1 en p_1 y

a v_2 en p_2, podemos entonces ubicarlo en alguno de estos dos conjuntos o príamos crear una tercera partición p_3 (siempre y cuando k >=3) para ubicar a v_3.

Así sucesivamente hasta llegar a k vértices. Luego, para v_1 hay 1 posibble parrición, para v_2 hay 2, para v_3 hay 3, hasta v_k con $k$particiones posibles y eso es nos deja $k!$.

Para los $(n-k)$ vértices restantes tenemos k subconjuntos por cada uno, por lo que es $k^{(n-k)}$.

Finalmente podemos concluir que la complejidad del algoritmo es del orden de $O(k! * k^{(n-k)})$ en el peor caso, sin considerar las podas que reducen la cantidad de soluciones que computo. Para este analisis se considero que el resto de las operaciones son $O(1)$ lo cual es así pues todas son operaciones elementales.

3.4. Experimentación

3.4.1. Casos Bordes

En esta sección vamos a analizar ciertas familias de grafos en las cuales sabemos como se debería comportar el algoritmo exacto y que resultado tendría que devolver, para corroborar su correcto funcionamiento. Esto no es una demostración de su correctitud, pero es una buena forma de ver si no se nos escapa la torutga.

Las familias que vamos a analizar son las siguientes:

- Grafos con n nodos aisldos
- Grafos en forma de estrella
- Grafos que sean ciclos simples de n nodos
- Grafos completos con todos pesos 1 en sus aristas

GRAFOS CON N NODOS AISLADOS

Para esto generamos instancias de 0 a 100 nodos y no agergamos ninguna arista. Luego corrimos el algoritmo para k entre 2 y 50.
Pudimos verificar que el resultado de el algoritmo para cualquier grafo sin importar ni la cantidad de nodos, ni las particiones el resultado es siempre 0 como peso de la suma de las aristas intrapartición lo cual es efectivamente cierto pues no existen aristas.

Corroboramos que el algoritmo exacto funciona correctamente para este tipo de instancias.

GRAFOS ESTRELLA

Para esto con el generador de instancias seleccionamos un nodo de forma aleatoria y lo conectamos al resto de los nodos.
Luego corrimos el algoritmo y nuevamente verificamos que se para cualquier cantidad de nodos y cualquier cantidad de particiones el peso total de la sumatoria de sus aristas es igual a 0. Es razonable pues al nodo central se lo puede ubicar en alguna particion y al resto en cualquier otra, con lo cual no hay aristas intraparticion.

CICLOS SIMPLES (C_n)

Generamos grafos que eran ciclos de n elementos. Y esperabamos que tambien nos de 0 la sumatoria de las aristas sin importar la cantidad de nodos ni las particiones. Pero nos olvidamos que los ciclos simples de longitud impar no eran bipartitos. Con lo cual para cualquier grafo que sea un ciclo de longitud impar si corro el algoritmo para encontrar una 2-particion mínima esta nunca va a ser 0.

Pudimos corroborar que para todos los grafos que sean circuitos simples para cualquier k mayor o igual a 3 su k-PMP es 0. Y para todos los que sean circuitos simples de longitud impar y para una 2-particion la sumatoria de sus aristas intraparticion es igual a la menor de las aristas

K_n *CON PESOS* 1

Con el generador de isntancias creamos grafos completos de n elementos con todos sus pesos en 1.

Pudimos verificar que el peso de las sumatoria de sus aristas intraparticion es:

$$k * (((n\ div\ k) * ((n\ div\ k) - 1)/2)) + (n\ mod\ k) * (n\ div\ k)$$

Básicamente esto es así pues en cada partición voy a tener como mínimo n div k nodos que se conectan todos entre todos por ser un grafo completo. Con lo cual esa partición va a aportar ((n div k) * ((n div k) - 1)/2) al peso total pues todas sus aristas intraparticion pesan 1. Luego tengo k particiones y por ultimo el resto de dividir a n por k son los nodos que me restan que los tengo que ubicar en alguna particion y estos se conectan con el resto de los nodos de esa particion.

3.4.2. Rendimiento

Para el análisis de rendimiento, corrimos el algoritmo variando la cantidad de particiones de 2 a 7, 100 instancias aleatoreas para cada n entre 5 y 20 de las cuales sacamos el promedio de tiempos, para las tres familias de grafos con el 15 %, 50 % y 100 % de sus aristas, tanto con el algoritmo sin podas y con podas.

La idea que tenemos es que el exacto sin podas al recorrer el arbol entero de solucioens solo varia en funcion de n y k por las colocacioens que puede realizar y agregandole la poda tendría que najar considerablemente el tiempo pues excluye una gran cantidad de ramas de las soluciones.

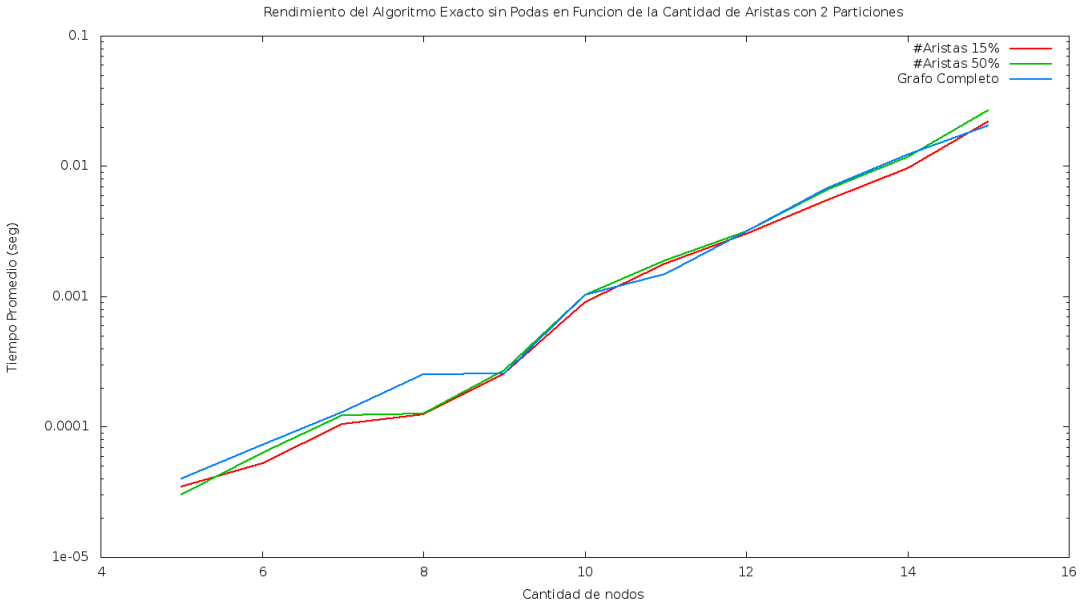

Figura 4: Rendimiento Exacto sin poda y K = 2

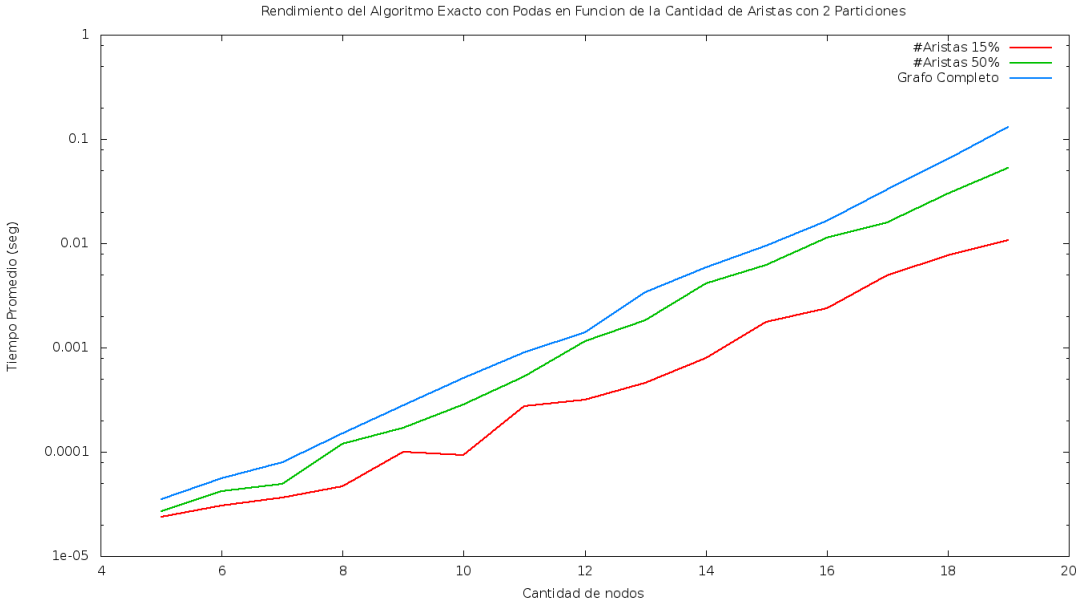

Figura 5: Rendimiento Exacto con poda y K = 2

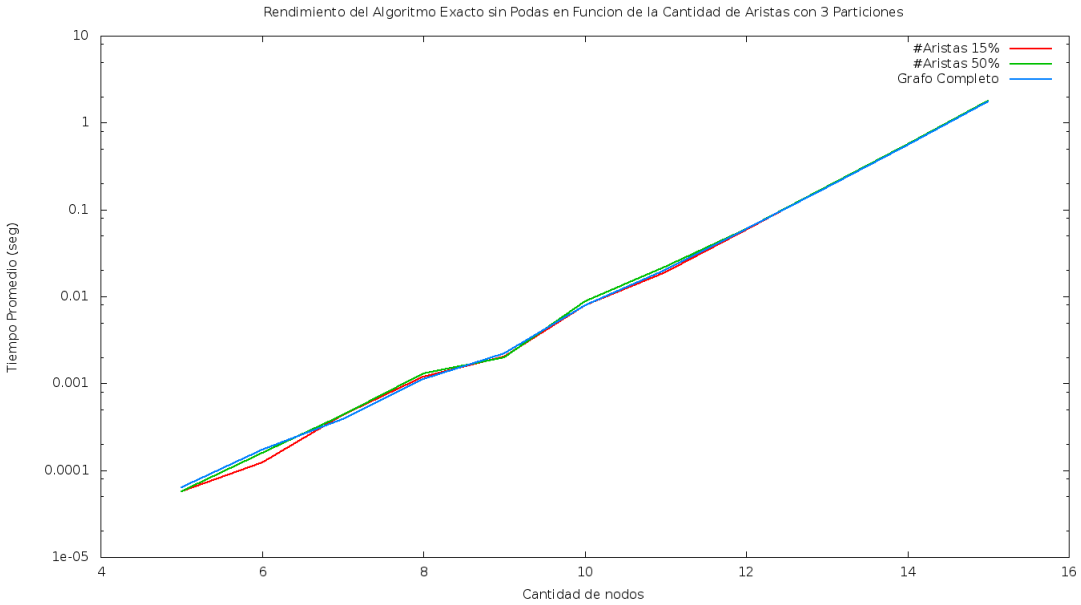

Figura 6: Rendimiento Exacto sin poda y K = 3

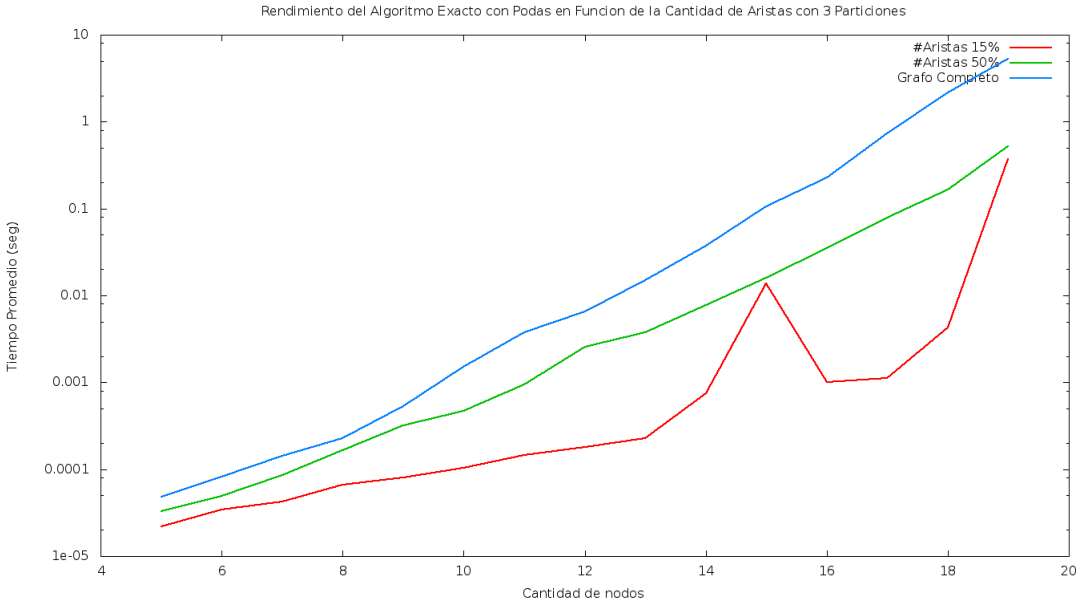

Figura 7: Rendimiento Exacto con poda y K = 3

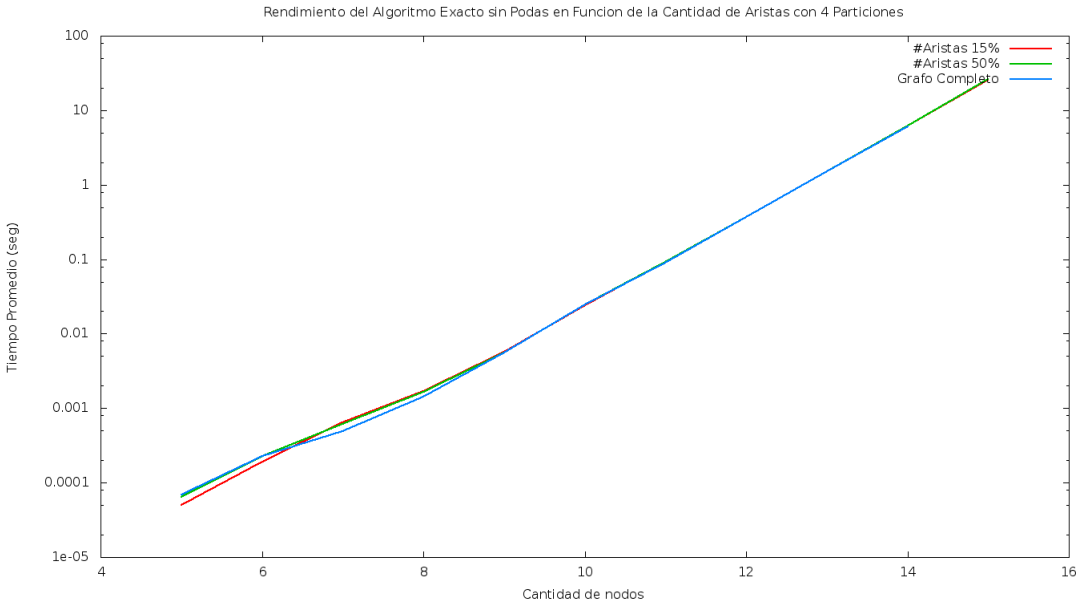

Figura 8: Rendimiento Exacto sin poda y K = 4

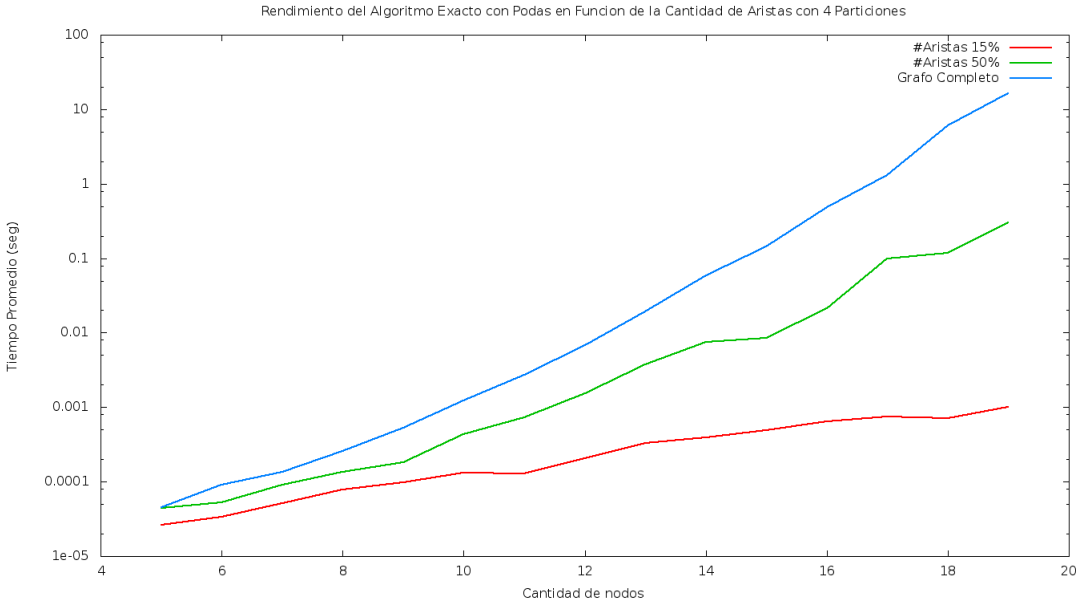

Figura 9: Rendimiento Exacto con poda y K = 4

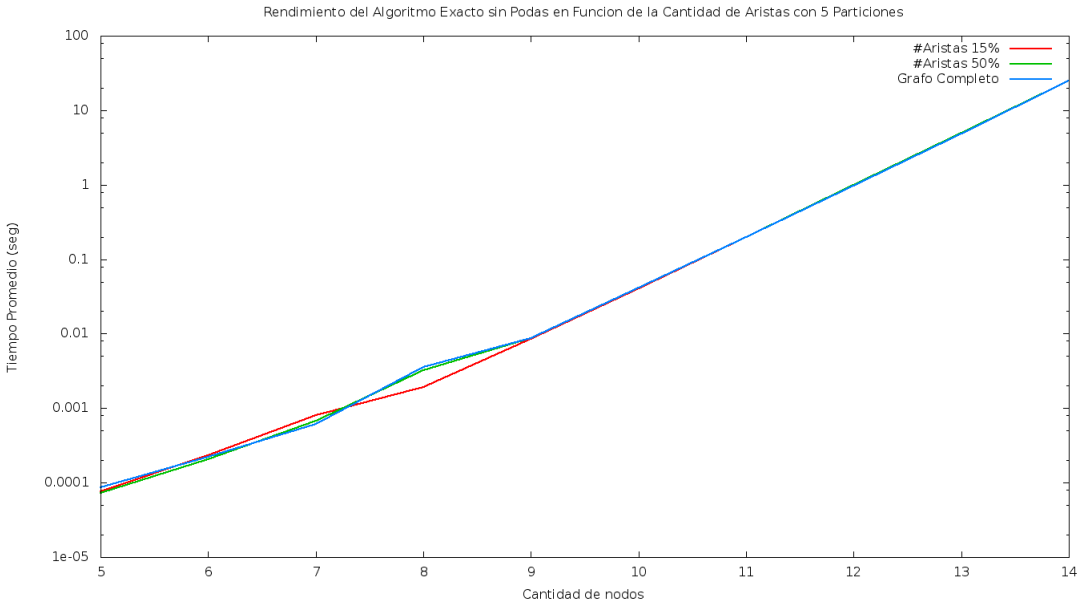

Figura 10: Rendimiento Exacto sin poda y K = 4

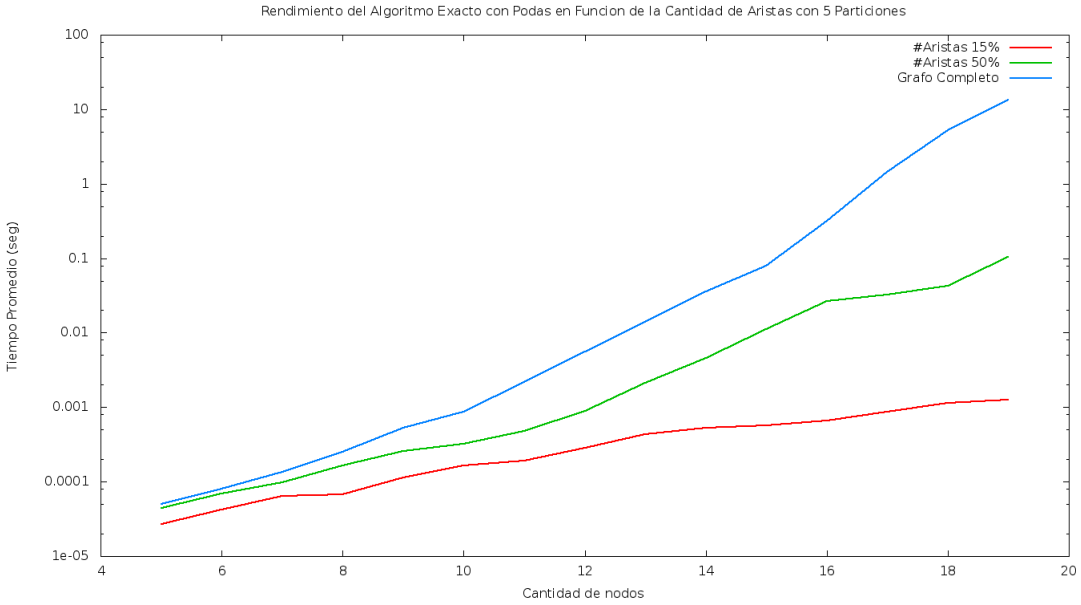

Figura 11: Rendimiento Exacto con poda y K = 5

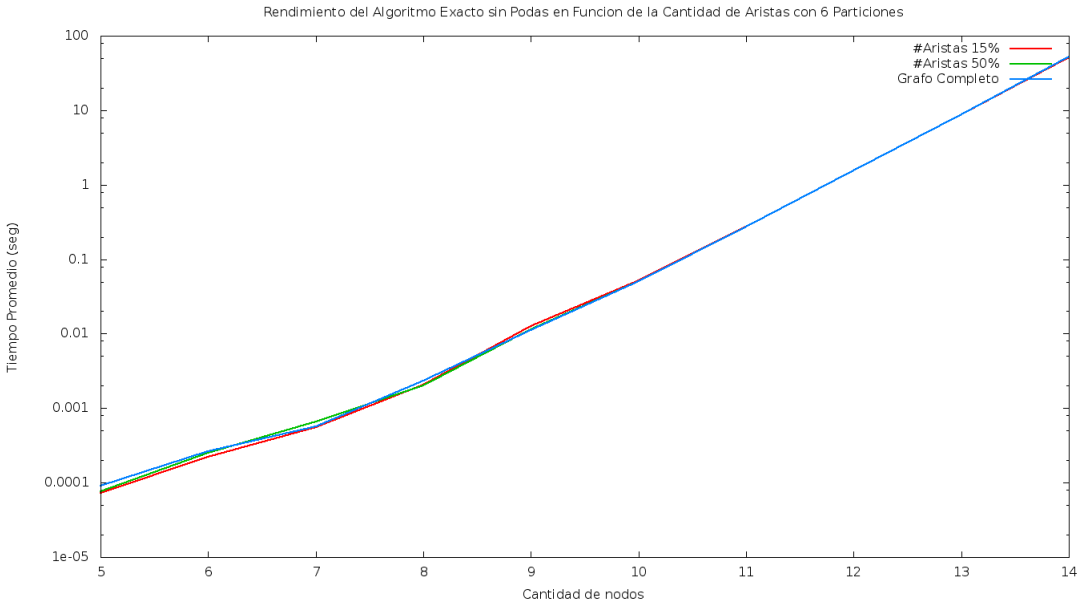

Figura 12: Rendimiento Exacto sin poda y K = 6

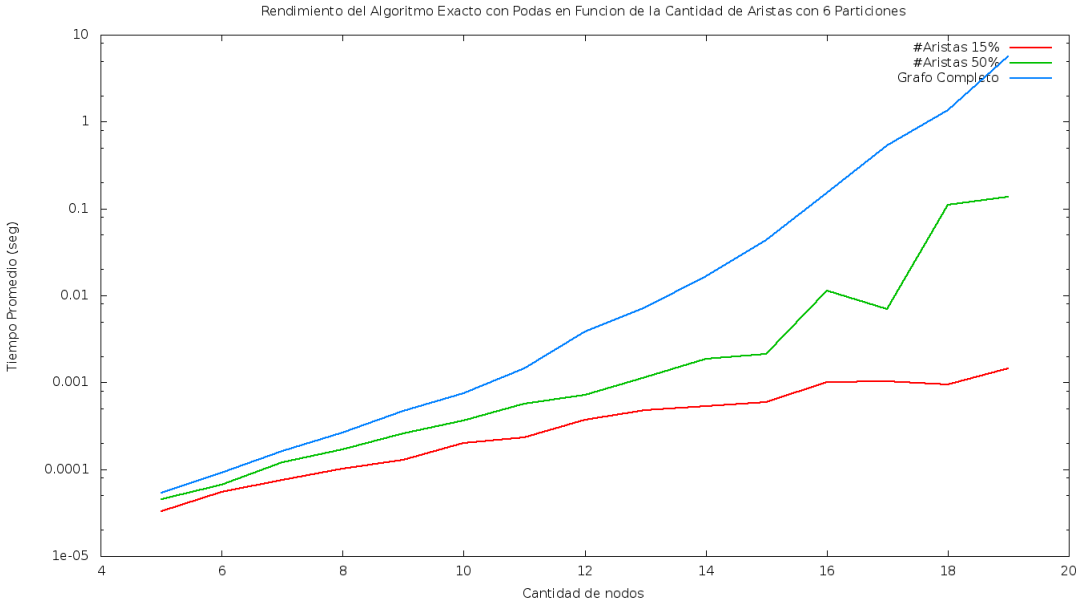

Figura 13: Rendimiento Exacto con poda y K = 6

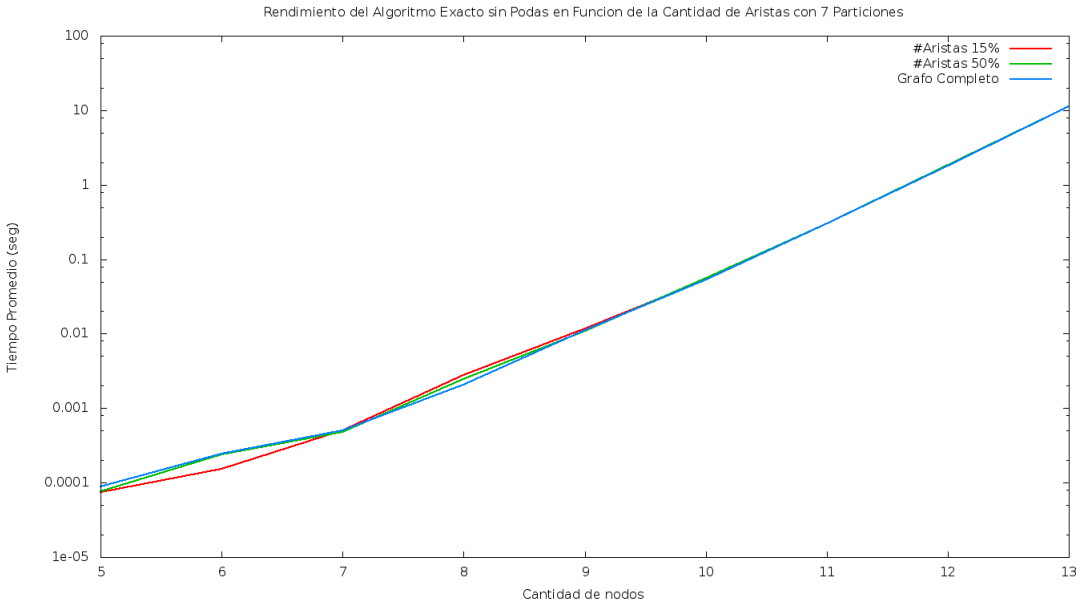

Figura 14: Rendimiento Exacto sin poda y K = 7

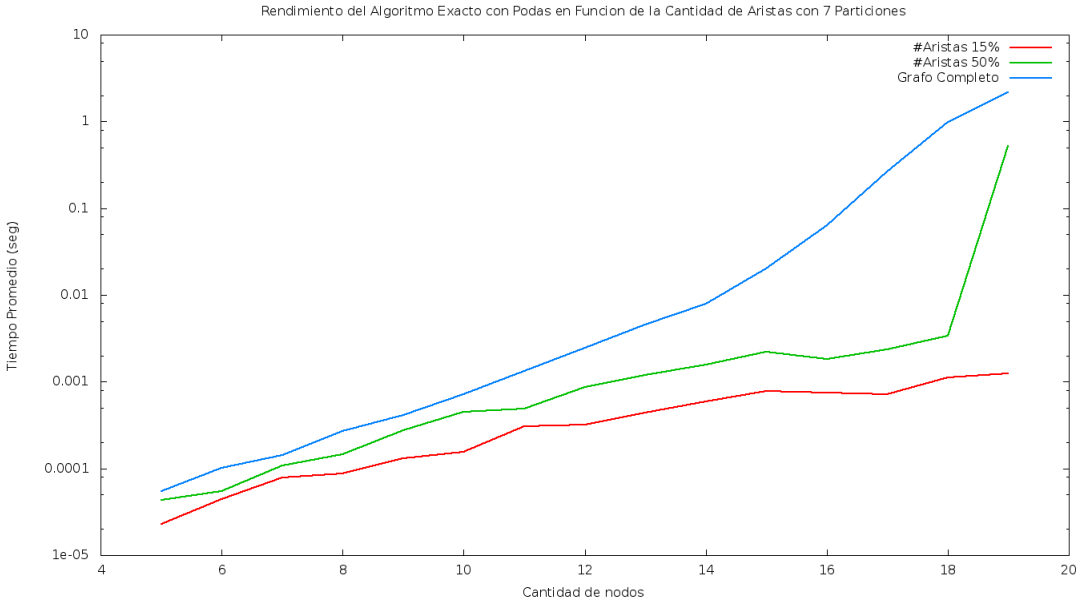

Figura 15: Rendimiento Exacto con poda y K = 7

Primero algo que podemos apreciar rápidamente es que el algoritmo exacto sin podas no es afectado por la cantida de aristas del grafo. Se comporta de la misma manera para el 15 % de las aristas como para el 100 % Algo que era de esperarse.

Luego podemos ver que ya para valores de 15 nodos el exacto sin poda contra el mismo con las podas difiere en dos ordendes de magnitud para cualquier k particion que se tome.

Y por otro lado podemos ver como al ir aumentando la cantida de particiones en funcion

que aumenta la cantidad de nodos, el algoritmo con podas al principio diverge en 1 orden de magnitud con los grafos del 15 % de aristas contra los del 100 % y para 7 particiones vemos que la diferencia es de alrededor de 4 ordenes de magnitud.

La diferencia en la ejecución en base a la densidad de los grafos para el algoritmo sin podas inferimos que es producto de que tan rápido se puede encontrar soluciones buenas que recorten la cantidad de soluciones a recorrer. Básicamente con el 15 % de las aristas, rápidamente podemos intuir que no alcanza para que sea conexo el grafo. Menos aun cuando algun nodo tiene varias arisatas incidentes. Con lo cual esto proboca tener varias componetes conexas en el mismo grafo, que generan conjuntos independientes con aristas intraparticón = 0. Se pueden repartir los nodos que si sean conexos en las distintas particiones y el reesto agergarlos en cualquiera total no aportan peso.

4. Heurística Golosa

4.1. Resolución

Para nuestra heurística golosa comenzamos ordenando las aristas por peso de mayor a menor. Una vez obtenida la lista de aristas ordenadas por peso la iteramos escogiendo primero un nodo (arbitrariamente) de la arista más pesada y si la arista aun no se encuentra ubicada, se agrega a un nuevo conjunto siempre y cuando no hayamos sobrepasado la cantidad k de conjuntos creados. En caso de que ya tengamos k conjuntos o en caso de que hayamos terminado de recorrer las aristas se termina el ciclo. Ahora verificamos si la cantidad de conjuntos creados es menor a k, en caso afirmativo habremos ubicado todos los nodos con aristas en alguno de los conjuntos. Si quedan nodos sin ubicar, estos se agregan a cualquier conjunto, como ya agregamos todos los nodos de grado mayor a uno, los que restan se puede decir que tienen grado 0, por lo tanto, no importa en que conjunto sean agregados, estos no crearán aristas intraparticion y como consecuencia no sumarán peso a ningún conjunto. Si la cantidad de conjuntos creados es igual a k, actuamos de forma diferente, aquí recorremos todas las aristas, actuando solo con las que no hayan sido agregadas a ningún conjunto de la siguiente manera: verificamos cuanto peso agregaría en cada conjunto para luego realmente añadirlo al conjunto que sume menos peso, en caso de encontrar uno en el que no cree aristas intraparticion, es agregado a este sin seguir revisando los restantes.

Como se puede observar usamos dos componentes greedys en la heurística: La primera esta cuando se ordenan las aristas decrecientemente y se crean los k conjuntos a medida que se separan los nodos de las aristas de mayor peso, para que estos no generen aristas intraparticion. La segunda componente greedy se encuentra cuando al finalizar de crear los k conjuntos se recorre nodo por nodo verificando si ya se encuentra agregado y en caso negativo, verificando en que conjunto genera menos peso para finalmente agregarlo a este.

A modo de ejemplo presentamos algunas imágenes para mostrar el funcionamiento de la heurística:

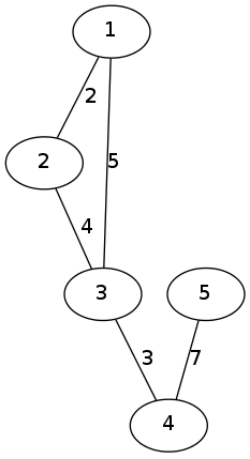

Figura 16: (1) Grafo de ejemplo

La primer imagen corresponde al grafo al que se le aplicará k-PMP (1)

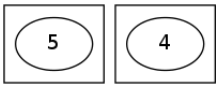

Figura 17: (2) Conjuntos 1 y 2 luego de separar los nodos de las aristas mas pesadas

Luego de ordenar las aristas escoge las de mayor tamaño y separa sus nodos en k conjuntos k=2 para este ejemplo. (2)

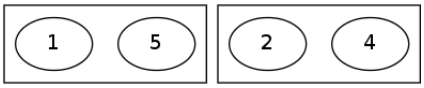

Figura 18: (3) Conjuntos luego de comenzar a ingresar los nodos restantes

Luego como ya hay 2 conjuntos creados procede a verificar nodo los nodos y agregándolos al conjunto que menos sume. En el caso del nodo 1, se agrega de inmediato al primer conjunto porque no genera arista intraparticion. Para el nodo 2 primero verifica en el 1er conjunto, aquí sumaria 2 de peso, se verifica en el siguiente conjunto, y como no suma peso, se ingresa ahí. (3)

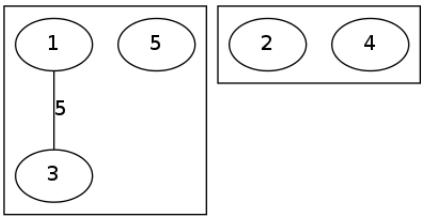

Figura 19: (4) Resultado de haber corrido la heurística greedy al grafo (1)

Finalmente resta agregar el nodo 3 el cual genera en el conjunto uno un peso igual a 5, y en el conjunto 2 un peso igual a 7 al unirse con los nodos 2 y 4. Por lo tanto el nodo 3 es introducido en el conjunto 1. (4)

A continuación presentamos un pseudo-gráfico del algoritmo

- ordenar *aristas* por peso en orden decreciente
- mientras restan *aristas*
  - si no hay k conjuntos, crear conjunto y agregar *nodo*1 de *arista* (si este no pertenece a ningún conjunto)
  - sino cortar el ciclo
  - si no hay k conjuntos, crear conjunto y agregar *nodo*2 de *arista* (si este no pertenece a ningún conjunto)
  - sino cortar el ciclo

- Si hay menos de k conjuntos creados
    - agregar las aristas que quedaron fuera a alguno de los conjuntos creados
- Sino
    - desde nodo 0 a nodo n-1
        - si el nodo no esta en ningún conjunto
            - ◇ verificar conjunto por conjunto cuanto peso generaría agregarlo a este
            - ◇ agregar el nodo al conjunto en el que genere menos peso

4.2. Análisis de complejidad

Vamos a analizar el código paso por paso analizando la complejidad temporal del peor caso. Lo primero que hacemos es ordenar las aristas, para esto utilizamos la función *sort* de la *std* que posee una complejidad temporal de $O(n.log(n))$, en este caso como se aplica a las aristas esto es $O(m.log(m))$ siendo m la cantidad de aristas. A continuación, y haciendo uso del pseudocódigo proporcionado:

- mientras restan *aristas*
    - si no hay k conjuntos, crear conjunto y agregar *nodo*1 de *arista* (si este no pertenece a ningún conjunto)
    - sino cortar el ciclo
    - si no hay k conjuntos, crear conjunto y agregar *nodo*2 de *arista* (si este no pertenece a ningún conjunto)
    - sino cortar el ciclo

Este scope tiene una complejidad $O(m)$ ya que se recorren todas las aristas, en el peor caso creando conjuntos y agregándolos a estos, pero estas dos acciones tienen una complejidad constante.

A continuación si se crearon menos de k conjuntos se agregan los nodos que restan a uno de estos. En el peor caso esto pertenece al orden de $O(n)$ siendo n la cantidad de nodos. Si hay k conjuntos creados se procede de la siguiente manera:

- desde nodo 0 a nodo n-1 $O(n)$
    - si el nodo no esta en ningún conjunto $O(1)$
        - verificar conjunto por conjunto cuanto peso generaría agregarlo a este $O(k+n)$
        - agregar el nodo al conjunto en el que genere menos peso $O(1)$

Esta ultima parte del algoritmo tiene una complejidad acotada en el peor caso de $O(n^2)$ ya que por cada nodo no agregado (n en total en el peor caso) se verifica cuanto pesaría agregarlo a cada conjunto. Para esto se recorren los nodos ya ingresados en el conjunto actual (peor caso n) y se agrega al que menor peso aporte. Esto tiene relación con el k también ya que en el peor caso se verificará en los k conjuntos.

4.3. Instancias desfavorables

Probamos la heurística con varias familias de grafos, grafos estrella...(ETC) Para el caso de los grafos estrella nuestra heurística funcionaba mal, particularmente cuando el numero del nodo central era superior a la cantidad k de conjuntos. Para aclarar, en un principio nuestro algoritmo constaba solo de la segunda componente greedy de la que se habla en la sección Resolución. Por lo que comenzaba desde el primer nodo hasta el ultimo probando en que conjunto convenía poner dicho nodo de forma de disminuir lo mas posible el peso total de las aristas intraparticion. Volviendo al caso del grafo estrella, lo que pasaba era que ingresaba en los

4.4. Experimentación

Para poder observar la performance del algoritmo en términos de tiempo de ejecución en función al tamaño de la entrada nos construimos un script en python que genera tres tipos de grafos, grafos con un 15 % de aristas, uno con un 50 % de aristas y el último con el 100 % de aristas, tomando para el porcentaje la cantidad de aristas que llevaría un grafo completo. Para cada uno de estos variamos los valores de n y de k con n entre 100 y 500. Por cada uno de esos valores de n, variamos el valor de k entre 2 y 10. Cada una de todas estas instancias fueron corridas entre 20 y 30 veces, (20 para las instancias mas grandes) para poder calcular un promedio. Ya que consideramos que el procesador no esta ejecutando solo nuestra heurística, por lo que el tiempo que le toma correr el algoritmo hasta obtener una solución seguramente es mayor al real. Por otro lado también pensamos en la posibilidad de que el procesador cachee las soluciones al estar haciendo muchas veces lo mismo, razón por la cual decidimos no superar las 30 repeticiones. Entre estas dos cosas donde una desfavorece y otra favorece el tiempo de ejecución, nos pareció razonable realizar un promedio. Con los valores obtenidos presentamos algunos gráficos para tener una descripción mas visual y los analizamos.

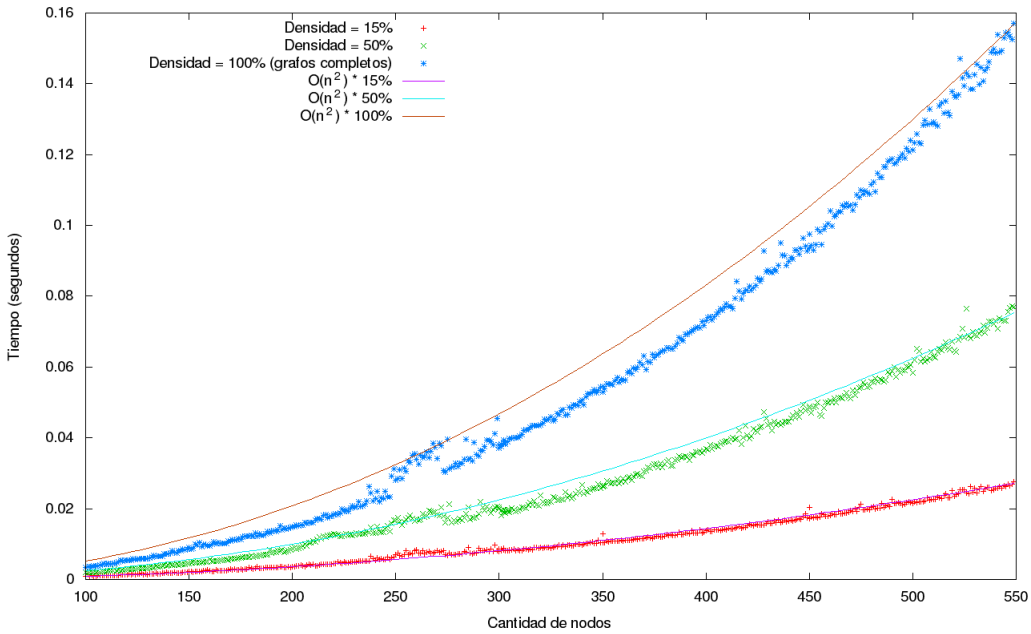

Figura 20: Gráficos con k = 2

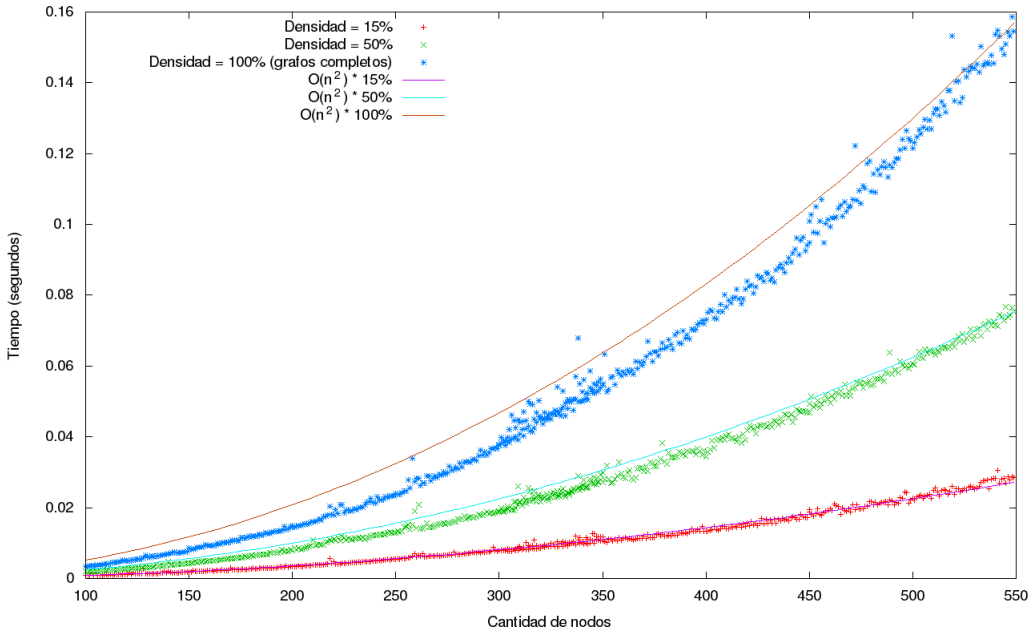

Figura 21: Gráficos con k = 5

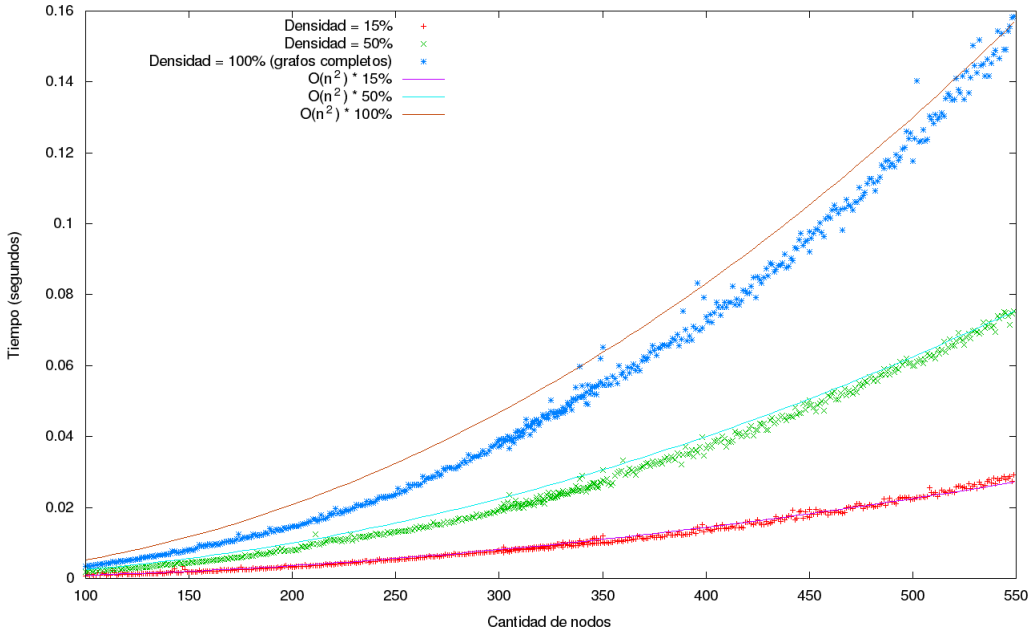

Figura 22: Gráficos con k = 10

En estos gráficos podemos observar que el algoritmo tiene una complejidad de n^2 en relación a la cantidad de nodos, la cual obviamente se ve afectada al mismo tiempo por la cantidad de aristas que el grafo posee. Esta observación se puede notar ya que cada grafo presenta la complejidad temporal para una densidad de aristas de 15 %, 50 % y 100 % comparada con un valor n^2 multiplicado por una constante obtenida a partir de dividir el tiempo sobre la

cantidad de nodos, y a su vez este valor es multiplicado por 0.15, 0.50 y 1. Al hacer esto y verificar que los valores se asemejan demasiado podemos concluir que la cantidad de aristas del grafo afecta directamente al rendimiento de forma lineal.

En los tres gráficos utilizamos los mismos valores para $O(n^2)$* densidad de cantidad de aristas, y como se puede observar comparando los distintos gráficos, que presentan una variación en la cantidad k de conjuntos, los valores resultantes de los tests son casi los mismos, no se nota ninguna variación apreciable.

5. Heurística de Búsqueda Local

5.1. Resolución

En nuestra heurística de búsqueda local decidimos tomar la solución desde la que partimos y mejorarla tomando el primer nodo *nodo*1 y moviéndolo a otro conjunto, verificando aquí si la solución del peso total de todos los conjuntos mejora o empeora (tomándonos la libertad de decir que empeora aun si el peso total es el mismo). Si empeora continuamos de la solución que ya teníamos tomando el próximo nodo, *nodo*2 y repitiendo el mismo proceso de probarlo el los demás conjuntos. En caso de que la solución mejore guardamos el nodo que genera esta mejora y el conjunto al que se debería mover. Una vez que se realizo este proceso en todos los nodos terminaremos teniendo una solución mejor o en el peor caso la misma con la que empezamos. Si la solución pudo ser mejorada, tomamos esta y se vuelve a ejecutar este procedimiento para esta nueva solución hasta que se llegue a una solución que no puede ser mejorada (mediante esta heurística). Como se puede notar para cada solución factible definimos el conjunto de soluciones "vecinasçomo: aquellas en las que difiere la posición de un solo nodo sobre los k conjuntos, iterando sobre estas hasta hallar la mejora de forma mayoritaria la solución. Esto se repite hasta que se encuentra un valor máximo local (máximo en cuanto a la optimalidad de la solución) que corresponde al mínimo peso obtenido.

Pensamos también plantear la misma solución cambiando la vecindad de soluciones, en esta nueva heurística los vecinos no son las soluciones que difieren en la posición de un nodo, sino que los vecinos los tomamos como aquellos en los que la diferencia se da por el switcheo de dos nodos que pertenecen a distintos conjuntos. Finalmente descartamos esta segunda heuristica porque presentaba muchos casos desfavorables, por ejemplo la solucion inicial que tiene a todos los nodos en un solo conjunto, como la vecindad esta en intercambiar dos nodos, no podremos sacar ninguno de donde estan. O asi mismo una solucion que presente uno de los k conjuntos vacios. Nunca se podrian agregar nodos a este..

A modo de ejemplo presentamos una demostración gráfica de la heurística.

Partimos de la siguiente solución (1)

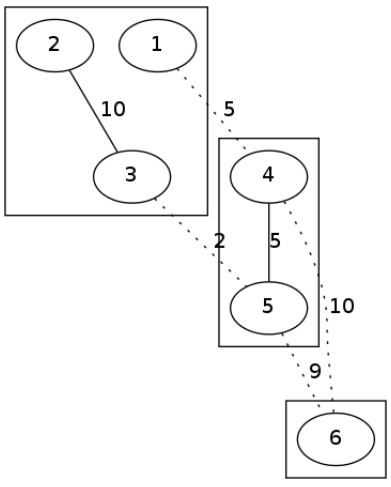

Figura 23: (1) Solución inicial con k=3

A partir de ahora verificara si moviendo un nodo de conjunto puede llegar a una mejor solución, en este momento la suma de los pesos de las aristas intraparticion es igual a 15. Moviendo el nodo 1 a cualquiera de los otros dos conjuntos no se logra disminuir el peso total, por lo que queda donde esta, ahora el nodo 2 logra disminuir en 10 moviéndolo a cualquiera de los dos conjuntos restantes, se decide moverlo al segundo. (2)

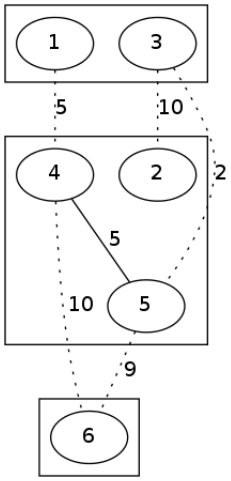

Figura 24: (2) Solución vecina habiendo movido el nodo 2

Ahora el nodo 3 aumenta el peso total si se mueve al siguiente conjunto y no aporta nada moverlo al ultimo, por lo que queda en el conjunto en el que está. Lo mismo pasa con el nodo 4, moviéndolo a uno aumenta el peso y al otro se mantiene. Sigue el nodo 5 que si se mueve al conjunto donde están los nodos 1 y 3 disminuye el peso total a 2, y si se mueve al conjunto restante aumenta el peso en 5, por lo que se ubica en el primer conjunto. Finalmente el nodo 6 no produce cambios de peso favorables moviéndolo, por lo que la solución a la que llega la heurística es la de la figura (3).

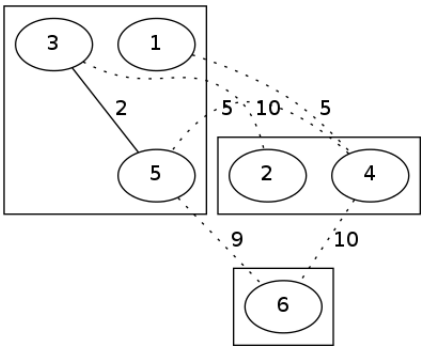

Figura 25: (3) Solución vecina habiendo movido el nodo 5 a partir de la solución (2).

Habiendo encontrado la solución (3) se llama al recursivamente y se vuelve a realizar el mismo procedimiento. En este caso los nodos 1, 2, 3 y 4 no mejoran el peso moviéndose de donde están, en cambio el nodo 5 logra encontrar un peso menor moviéndose al conjunto donde

se encuentra el nodo 6 siendo este cero. Al encontrar una solución óptima ningún movimiento podrá mejorarla, razón por la cual la solución final es la que muestra la imagen (4).

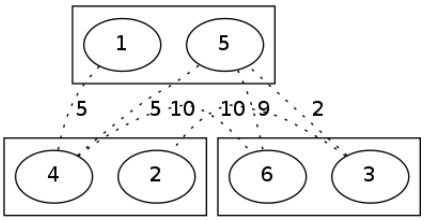

Figura 26: (4) Solución máxima local.

A continuación presentamos un pseudo-gráfico del algoritmo realizado:

- RESOLVER
  - Para cada vértice v
    - Para cada partición p
      - Si v esta en la partición k, Se calcula cuanto peso 'perdería' restando las aristas intraparticion que inciden en k
      - Sino se calcula cuanto peso 'sumaria' mover v a p guardando el peso, v y p si suma menos peso que con los v y p anteriores.
    - Si se llego a una solución mejor, se guarda
  - Si se encontró alguna solución mejor a la inicial se la toma y se llama recursivamente a RESOLVER con esta.

6. Heurística GRASP

6.1. Resolución

La metaheuristica GRASP es una combinacion de una heuristica golosa "aleatorizada" y un procedimiento de busqueda local. En detalle GRASP se define de la siguiente manera:

- Mientras no se alcance el **criterio de terminacion**
  - Obtener $s \in S$ mediante una heuristica golosa **aleatorizada**.
  - Mejorar s mediante busqueda local
  - Recordar la mejor solucion obtenida hasta el momento.

Para nuestro GRASP, basamos la heuristica golosa aleatorizada en la segunda componente golosa descrita en el punto 4.
Entrando en detalle, la heuristica golosa itera sobre todos los vertices y decide sobre cada uno a que particion va a ser agregada. La manera en que lo hacia en la heuristica golosa previamente especificada era elegir la particion tal que el peso agregado a la solucion sea el menor posible. Para ello requeria iterar sobre todos las Particiones y preguntarse cuanto pesaria de ser agregada alli.
En la heuristica randomizada cambia en que primero comenzamos con k particiones vacias. Y segundo que en vez de elegir directamente la mejor opcion, armamos una RCL (lista Restringida de Candidatos), donde los candidatos son Particiones. Luego se elige aleatoriamente un destino cualquiera entre los candidatos y se agrega el vertice en el.
Definimos la restriccion de los candidatos utilizando dos parametros alpha y beta.
Decimos que los candidatos no pueden tener un valor menor que un cierto porcentaje alpha del valor del mejor candidato. Empezamos por identificar al mejor candidato como el menor peso agregado a la solucion, y al peor candidato como el mayor peso agregado a la solucion. La restriccion luego excluye a aquellos candidatos cuyo peso agregado a la solucion no se comprenda entre el menor peso y el mayor peso menos el porcentaje alpha de la distancia entre ambos pesos.
Por ultimo tambien la RCL puede contener como maximo los beta mejores candidatos.

En cuanto al procedimiento de busqueda local utilizamos nada mas y nada menos que la misma heuristica concebida en el punto anterior.

Y finalmente como criterios de terminacion, definimos dos. Uno es la cantidad maxima de 10475. Es decir la cantidad de veces que se itera sobre el ciclo principal de GRASP. Y el otro la cantidad maxima de 10475 seguidas sin cambios. Es decir el maximo de veces que itera sobre el ciclo principal, luego de haber encontrado una mejora, sin encontrar otra.

A continuación presentamos un pseudo-codigo del algoritmo goloso aleatorizado:

- desde vertice 1 a vertice n
  - por cada particion existente
    - verificamos el peso de agregar el vertice a la particion

        - Si el peso resultado de agregarlo es el maximo encontrado hasta el momento lo recuerdo
        - Si el peso resultado de agregarlo es el minimo encontrado hasta el momento lo recuerdo
        - Agregamos la particion destino con peso asociado a la lista RLC
        - Si el tamano de RLC es mayor a beta:
            - Eliminamos la particion destino con mayor peso
- Eliminamos de RLC las particion destino cuyo valor de peso no se encuentre entre el rango alpha de mejores
- Elegimos aleatoriamente una particion destino entre las candidatas y agregamos el vertice en esa particion

6.2. Analisis de complejidad

Analisamos la complejidad del pseudo-codigo del algoritmo goloso aleatorizado:

Lo primero que hacemos es iterar por todos los vertices. De 1 a n. Esto es O(n)

- desde vertice 1 a vertice n $O(n)$
    - por cada particion existente $O(k)$
        - verificamos el peso de agregar el vertice a la particion $O(n?)$
        - Si el peso resultado de agregarlo es el maximo encontrado hasta el momento lo recuerdo $O(1)$
        - Si el peso resultado de agregarlo es el minimo encontrado hasta el momento lo recuerdo $O(1)$
        - Agregamos la particion destino con peso asociado a la lista RLC (ORDENADAMENTE) $O(min(k, beta))$
        - Si el tamano de RLC es mayor a beta: $O(1)$
            - Eliminamos la particion destino con mayor peso $O(1)$
- Eliminamos de RLC las particion destino cuyo valor de peso no se encuentre entre el rango alpha de mejores $O(min(beta, k))$
- Elegimos aleatoriamente una particion destino entre las candidatas y agregamos el vertice en esa particion $O(min(beta, k))$

Haciendo un analisis mas profundo podemos notar que el costo en complejidad de verificar el peso de agregar el vertice a una particion es: O(1) por la k_i particion + la cantidad de vertices que contenga. El costo para un vertice i luego es O(i-1) + O(k) comparaciones por todas las particiones y los vertices totales contenido en ellas, i es creciente desde 1 hasta n. La complejidad resultado acotando a k por n es O(n).
Agregar ordenadamente a una lista ordenada cuesta el tamano de la lista. Y la lista contiene como maximo todas las particiones, es decir como maximo k. Recordemos que k podemos acotarlo por n, entonces la complejidad es absorida por el item anterior deducido O(n). La complejidad de eliminar particiones de una lista y de elegir aleatoriamente tambien se acotan por O(n). La complejidad final es O(n)*O(n) = O(n2).

La complejidad de busqueda local ya la calculamos en el punto anterior y es O(n2).

Finalmente la complejidad de Grasp es O(max10475*n2). Dado que logicamente la cantidad de 10475 seguidas sin cambios es menor a la cota maxima cantidad de 10475.

Los costos de copia de lista de particiones como resultado de busqueda local o greedy random o comparacion con anterior mejor solucion son despreciados por encontrar un maximo costo de O(n).

6.3. Criterios de parada y RCL

En primer lugar queremos definir un **criterio de calidad**, el cual utilizaremos para decir que tan **bueno** es el resultado obtenido por la implementacion de una heuristica. Decimos que los mejores resultados posibles son aquellos cuyo peso sea el optimo. Dado un grafo, una manera accesible para nosotros de saber cual es el peso optimo es corriendo la implementacion del algoritmo exacto.
Decimos tambien que los peores resultados posibles son aquellos cuyo peso sea maximo. Dado un grafo, una manera de saber cual es el peso maximo es sumar todas las aristas de un grafo. Una peor solucion posible podria ser una k-particion tal que una contenga todos los nodos y el resto sean vacios.

Definimos el resultado de la calidad de un algoritmo como un valor real entre 0 y 1 no excluyente, donde 1 es el mejor peso posible (el minimo) y 0 el peor peso (el maximo).
Luego la calculamos de la siguiente manera: $(1 - (pesoObtenido - mejorPeso)/(peorPeso - mejorPso))^2$
La multiplicacion al cuadrado del valor tiene varios motivos. Uno, es por simple impacto visual para desmotivar los valores alejados del mejor resultado. Y dos es que luego compararemos la calidad en funcion del tiempo para elegir mejores criterios de terminacion y de esta manera le podemos dar un poco mas de fuerza al peso.

Como comentamos anteriormente tenemos dos parametros fundamentales en la seleccion de la RCL. Alpha y beta.
Con el objetivo de encontrar **buenos** parametros implementamos un ejecutable llamado testingGrasp.
La idea de testingGrasp es alternar los valores de alpha y beta y testear la calidad de grasp contra grafos generados aleatoriamente. A su vez por cada parametro variamos la cantidad de aristas m, la cantidad de particiones k y la cantidad de nodos n.
Debido a que hay tantos parametros en juego, acotamos un poco las combinaciones.
Probamos con n = 4, 8, 12, 16. Solo hasta n=16 dado que para valores mas grandes, el algoritmo exacto empieza a ser muy lento y hace el testeo imposible.
Con m = 2, 4, 8 , ... , (n-1)*n.
Y con k = 2, 4, 6, .. n.
Alpha va desde 0 a 1 entre saltos de 0,05. Y beta desde 1 hasta 16 (valor maximo de n).
Tambien por cada combinacion de parametros, se randomiza el grafo unas 50 veces y se ejecuta grasp unas 5 veces por cada una.
Como resultado obtenemos el promedio de la calidad en funcion de los parametros especificados para las mejores 10 calidades promedio.
A continuacíon los valores retornados:

alpha	beta	iteraciones	calidad
0.4	3	104750	0.986533
0.85	6	104750	0.986485
0.85	15	104750	0.98636
0.8	2	104750	0.98636
0.55	8	104750	0.986334
0.85	3	104750	0.986306
0.05	2	104750	0.986298
0.55	9	104750	0.986296
0.95	12	104750	0.986283
0.5	3	104750	0.986278

Luego elegimos por pequenicima diferencia el mejor alpha y beta.
Alpha = 0,4. Y beta = 3.
En cuanto criterios de terminacion, utilizamos una herramienta que creamos llamada comparaciones.py.
La misma se encarga de generar instancias aleatorias de grafos y ejecutar varias algoritmos simultaneamente. A su vez compara la calidad de esos algoritmos. Dado distintos valores de maxima cantidad de iteraciones e iteraciones sin cambios. Nos queremos quedar con aquellos cuyo valor epsilon = tiempoTardado*calidad sea optimo.

7. Conclusiones

En esta seccion vamos a realizar una conclusión general de los distintos algoritmos. En base a la performance de los mismos y las distancias con respecto a la solución Exacta.

Primero planteamos un experimento en el cual corrimos para instancias aleatoreas, nuevamente con las tres densidades utilizadas a lo largo de todo el Trabajo Práctico(15 %, 50 % y 100 %), todos los algoritmos (Exacto con podas, Heuristica Golosa, Heuristica de Busqueda Local y Heuristica GRASP) unas 50 veces para cada cantidad de nodos entre 5 y 20 (por ser numeros manejables para el algoritmo exacto) para sacar un promedio de cuanto difiere cada Heuristica contra el algoritmo Exacto.
Para esto calculamos la peor solución para cada instancia (que sería ubicar todos los nodos en una única partición, con lo cual la sumatoria de las aristas intrapartición será la sumatoria total de las aristas) y generamos una formula que nos representa un valor entre 1 y 0 que tanto diverge la Heuristica en relación al exacto, pero considerando el valor entre la peor solución y la solución ideal. La formula es:

1 - ((solucionHeuristica - solucionExacto)/(peorSolucion - solucionExacto))

Si la Heuristica tiene el mismo valor que el Exacto esta relación nos devuelve que es 1. Todo lo contrario si la heuristica devuelve la peor solución, esta formula nos retorna 0.
Con lo cual es un buen indicador para saber cuan buena es la solución de una de las heuristicas.

7.1. Divergencia de las Heuristicas

7.1.1. 2 Particiones

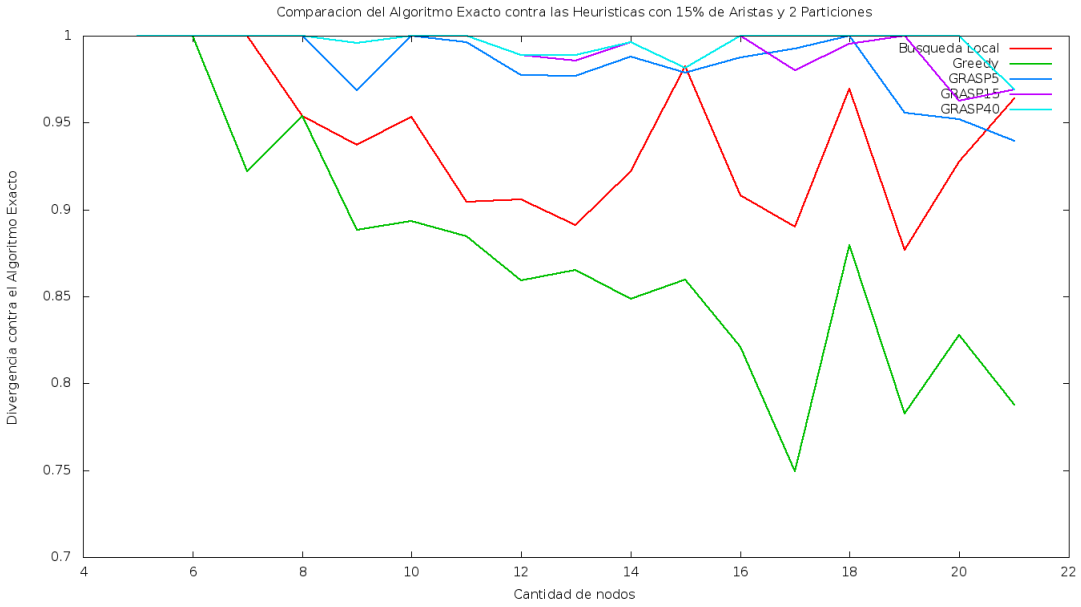

Figura 27: Distancias de las Heuristicas para K = 2 y 15 % de aristas

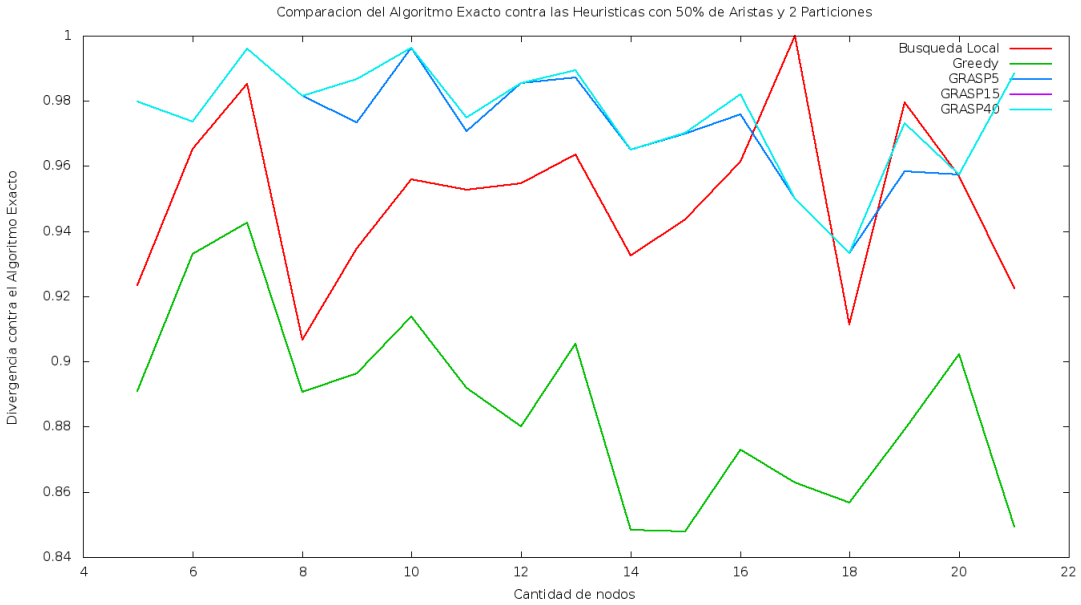

Figura 28: Distancias de las Heuristicas para K = 2 y 50 % de aristas

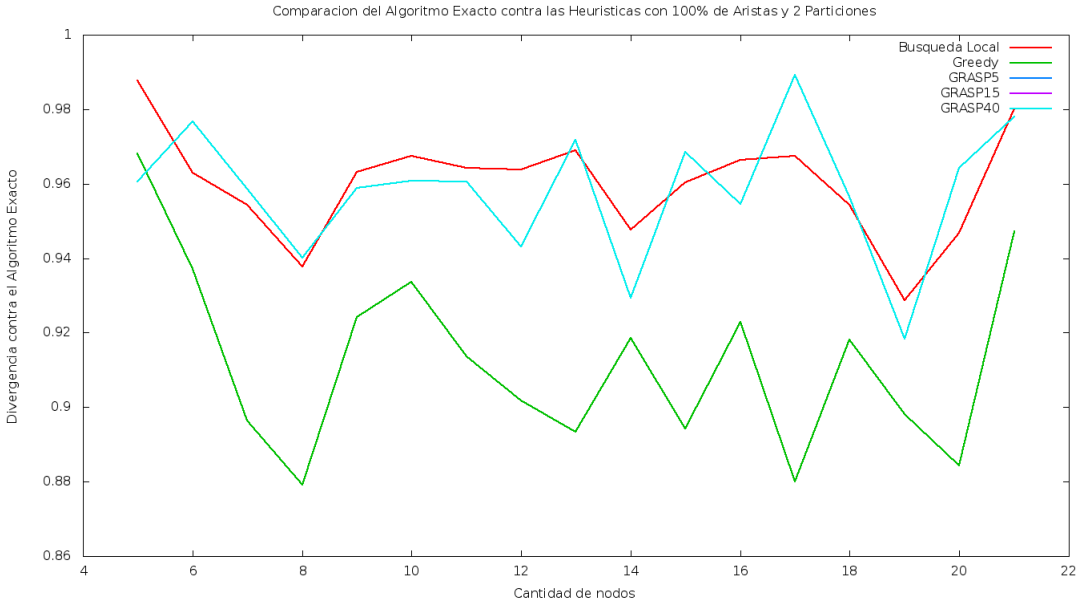

Figura 29: Distancias de las Heuristicas para K = 2 y 100 % de aristas

7.1.2. 3 Particiones

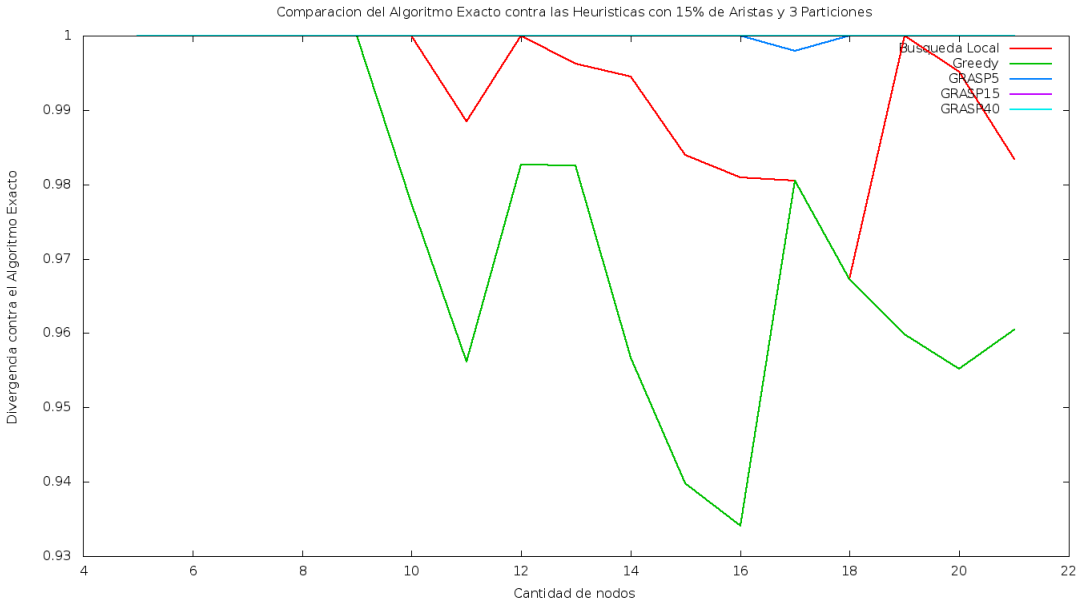

Figura 30: Distancias de las Heuristicas para K = 3 y 15 % de aristas

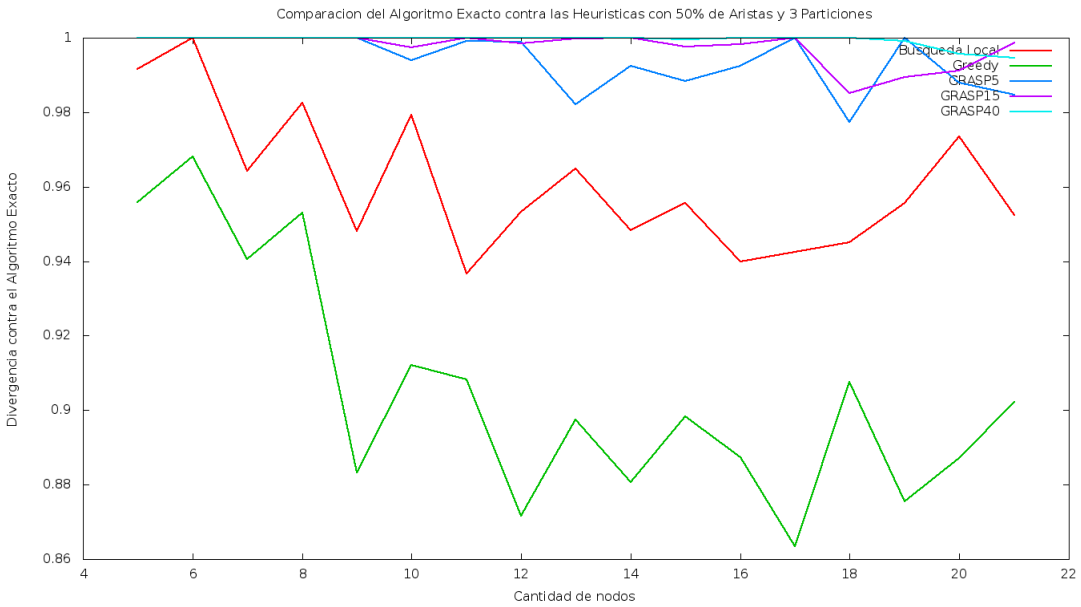

Figura 31: Distancias de las Heuristicas para K = 3 y 50 % de aristas

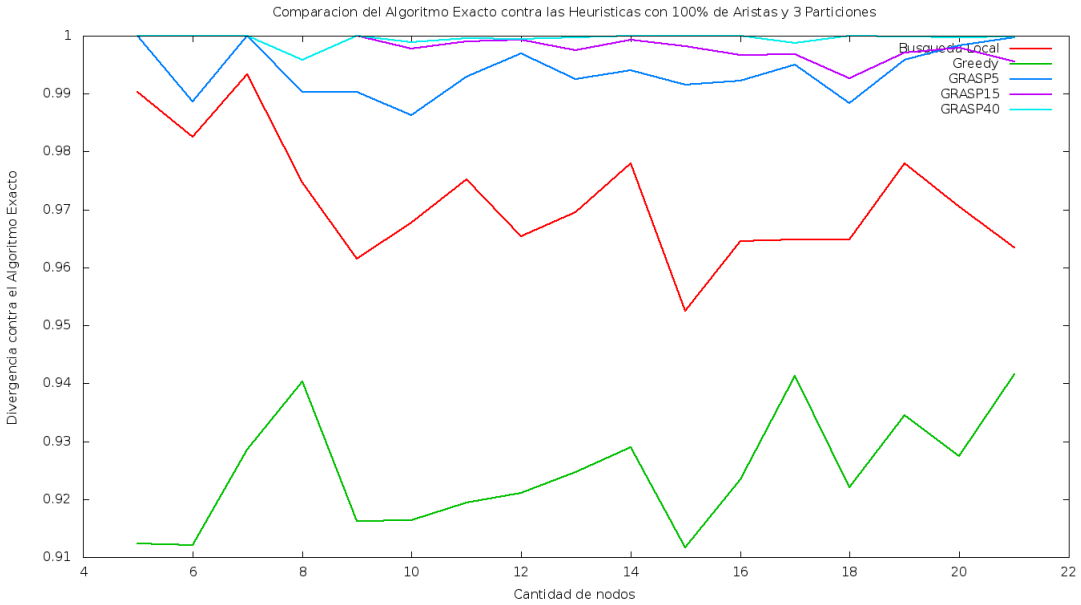

Figura 32: Distancias de las Heuristicas para K = 3 y 100 % de aristas

7.1.3. 4 Particiones

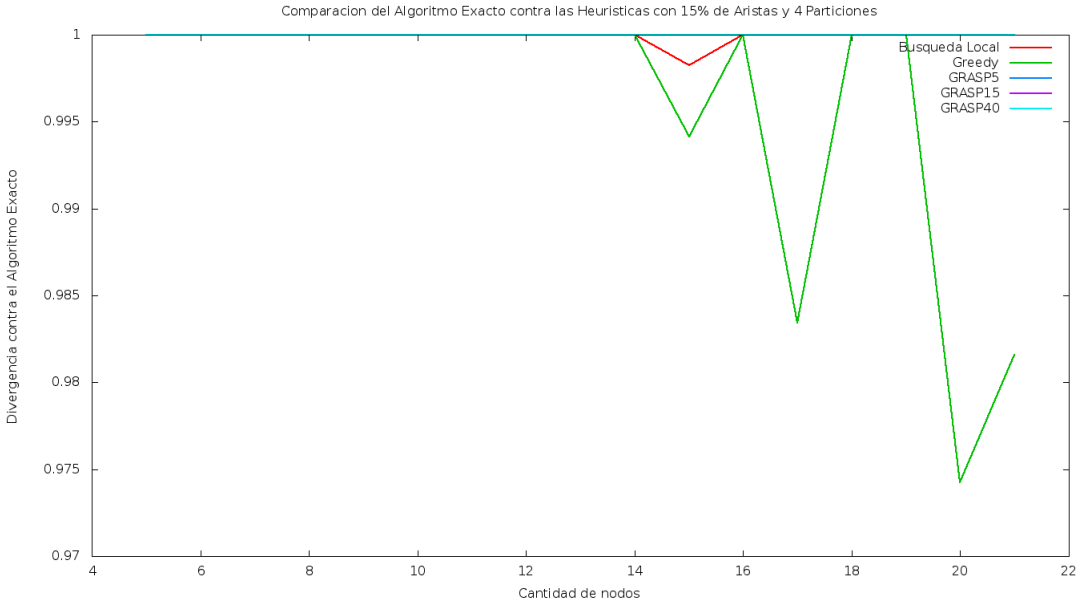

Figura 33: Distancias de las Heuristicas para K = 4 y 15 % de aristas

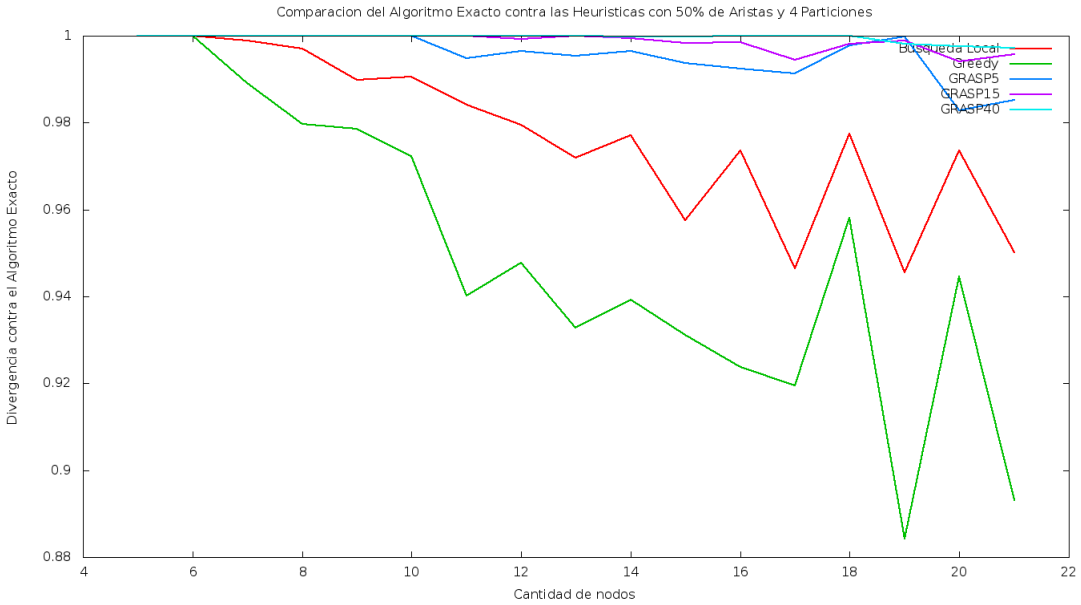

Figura 34: Distancias de las Heuristicas para K = 4 y 50 % de aristas

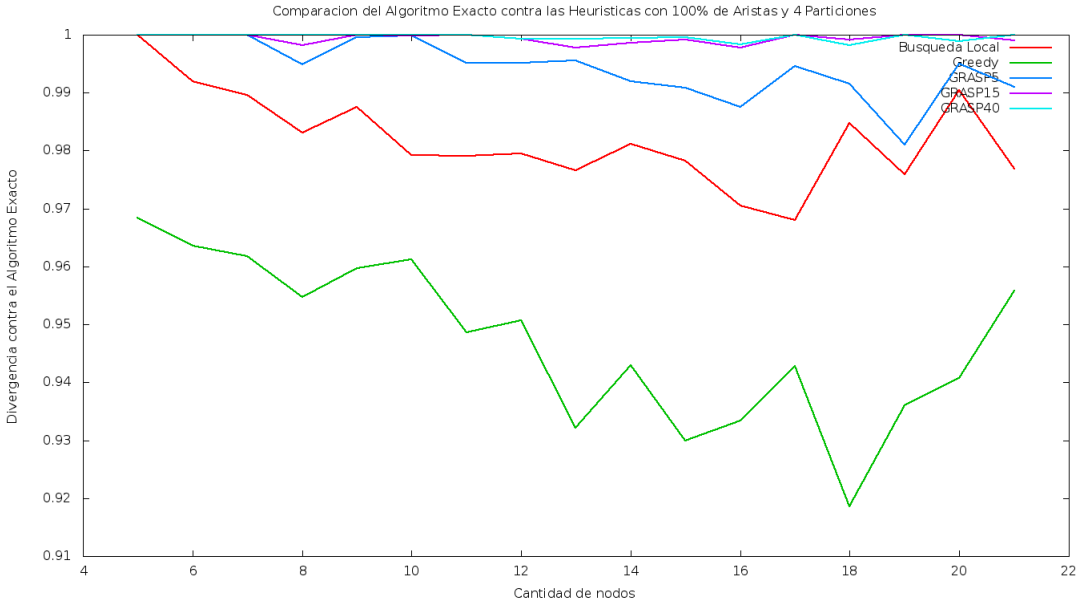

Figura 35: Distancias de las Heuristicas para K = 4 y 100 % de aristas

7.1.4. 5 Particiones

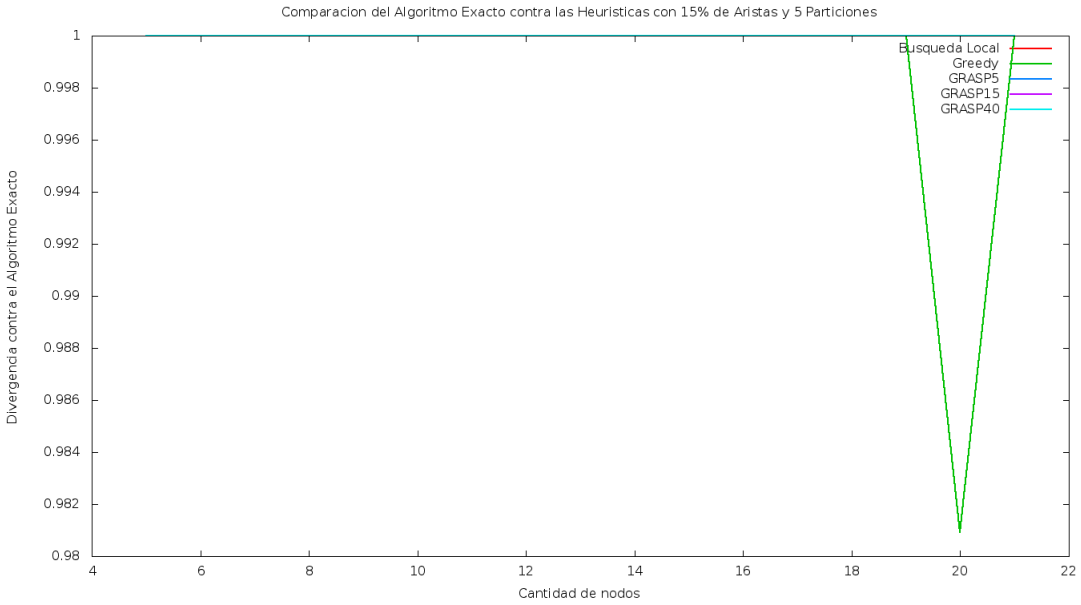

Figura 36: Distancias de las Heuristicas para K = 5 y 15 % de aristas

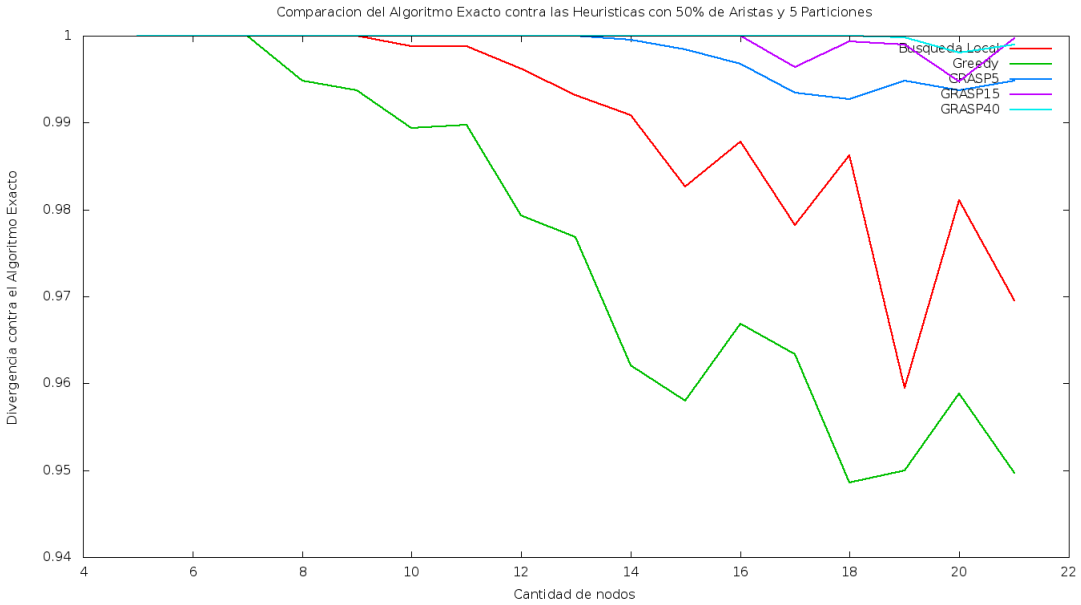

Figura 37: Distancias de las Heuristicas para K = 5 y 50 % de aristas

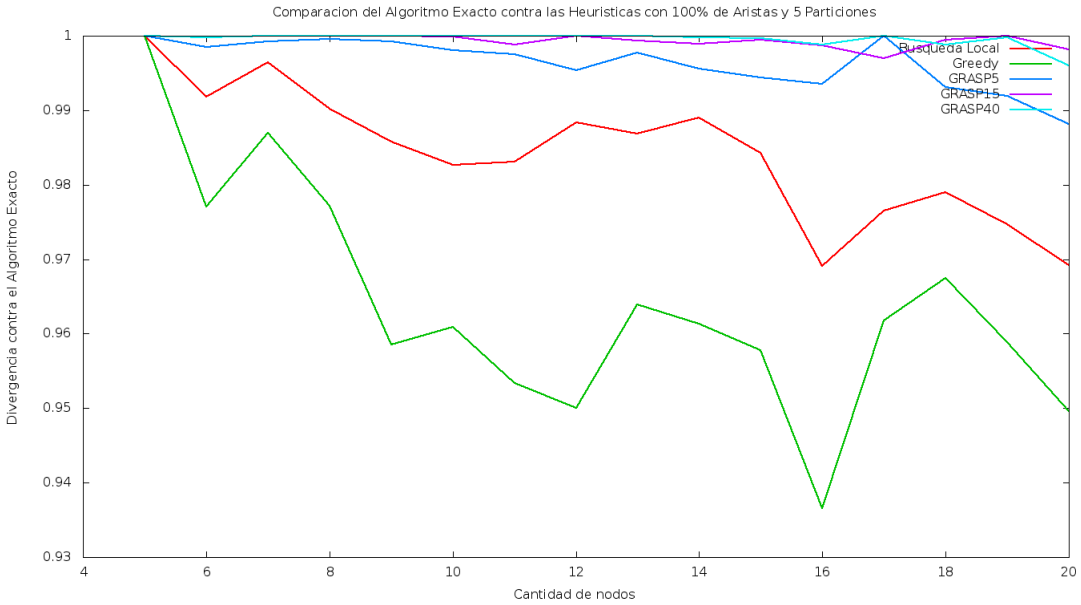

Figura 38: Distancias de las Heuristicas para K = 5 y 100 % de aristas

7.1.5. 6 Particiones

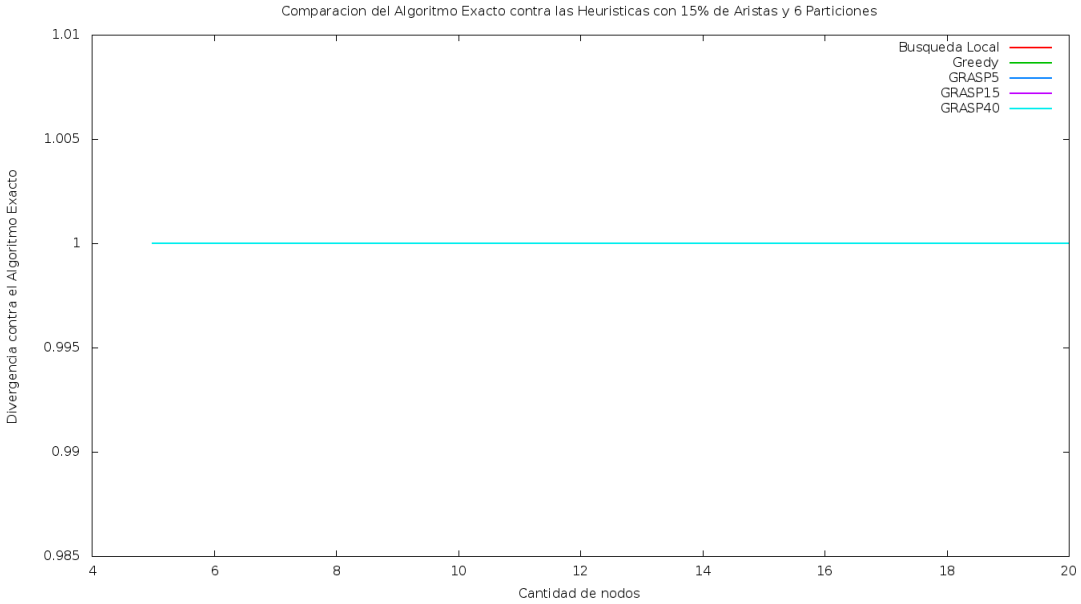

Figura 39: Distancias de las Heuristicas para K = 6 y 15 % de aristas

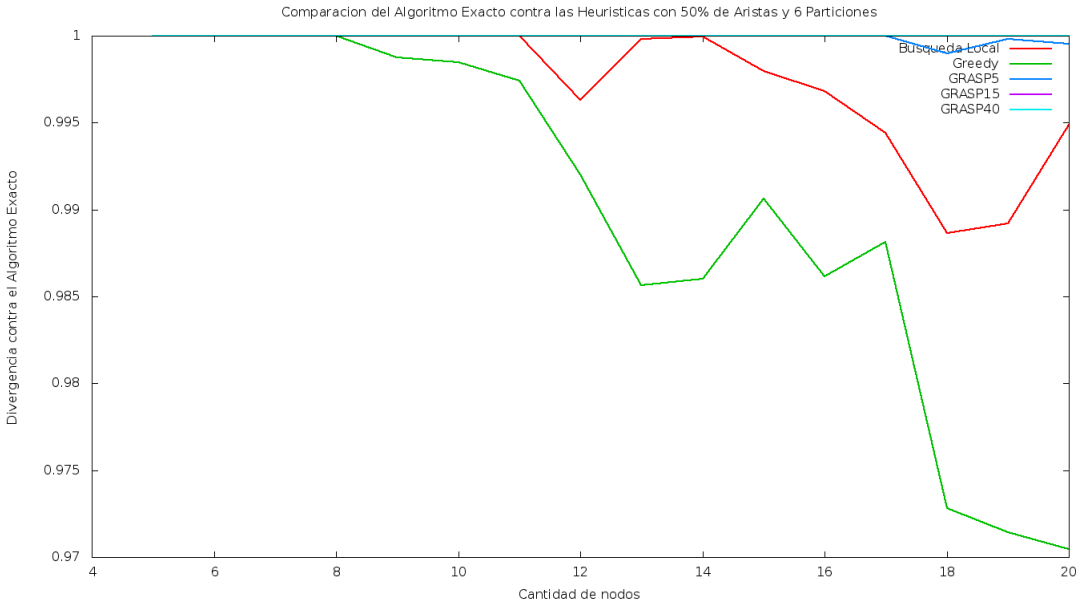

Figura 40: Distancias de las Heuristicas para K = 6 y 50 % de aristas

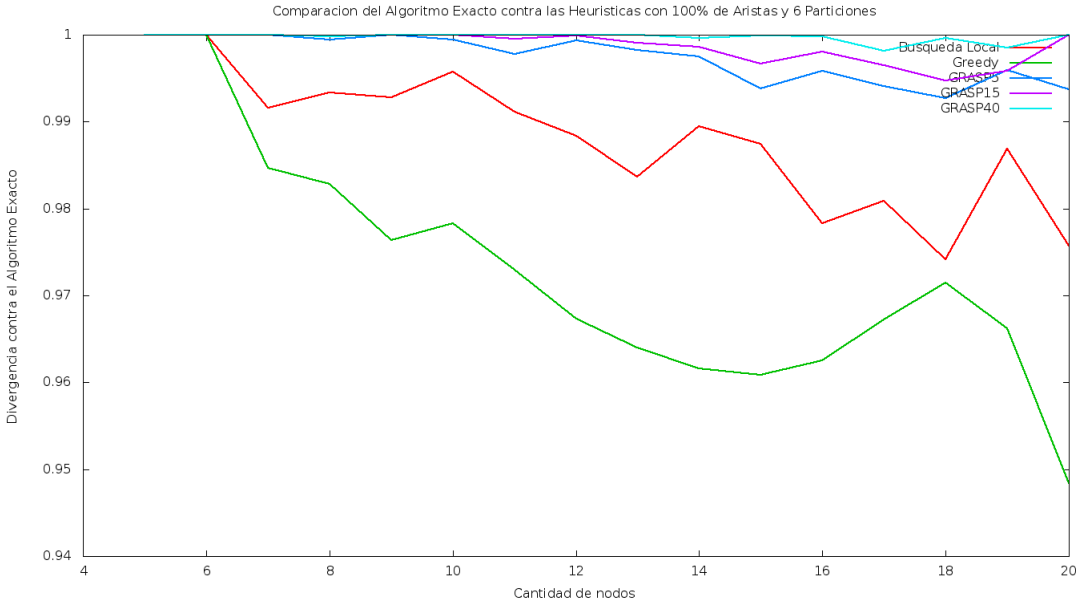

Figura 41: Distancias de las Heuristicas para K = 6 y 100 % de aristas

7.1.6. 7 Particiones

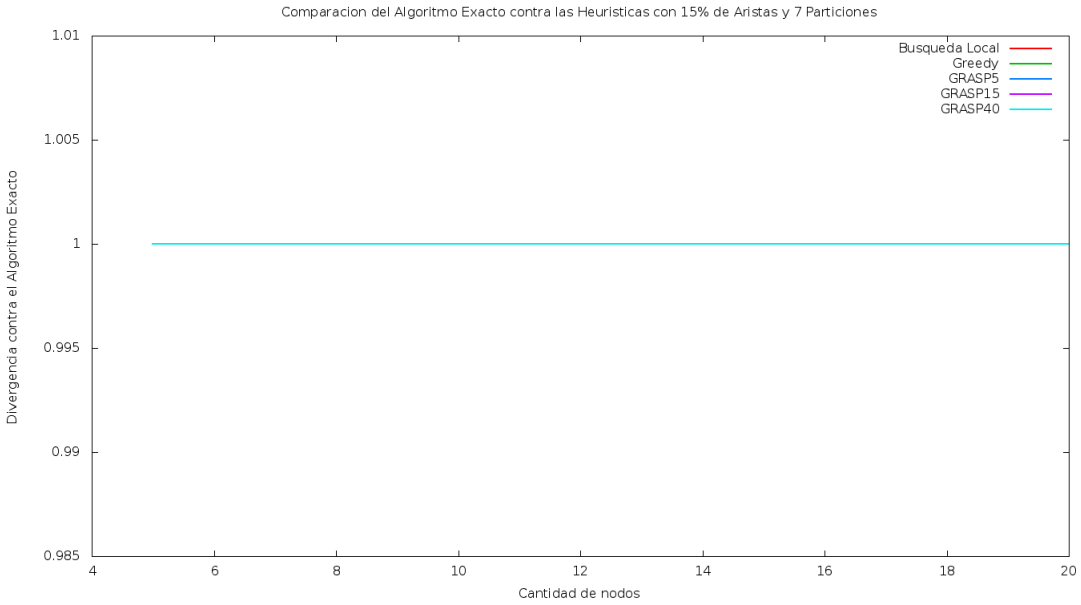

Figura 42: Distancias de las Heuristicas para K = 7 y 15 % de aristas

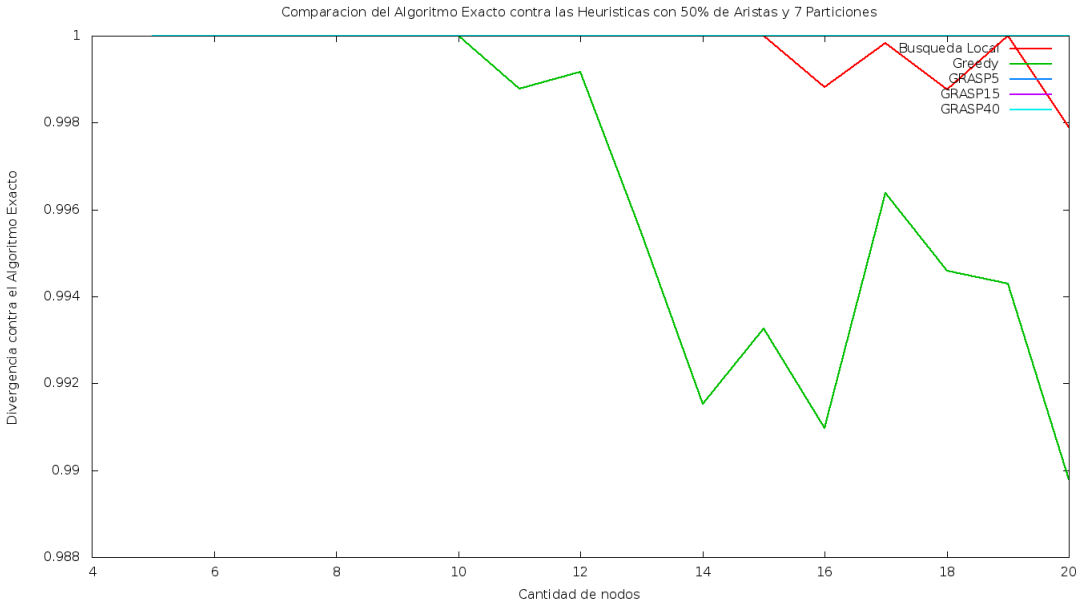

Figura 43: Distancias de las Heuristicas para K = 7 y 50 % de aristas

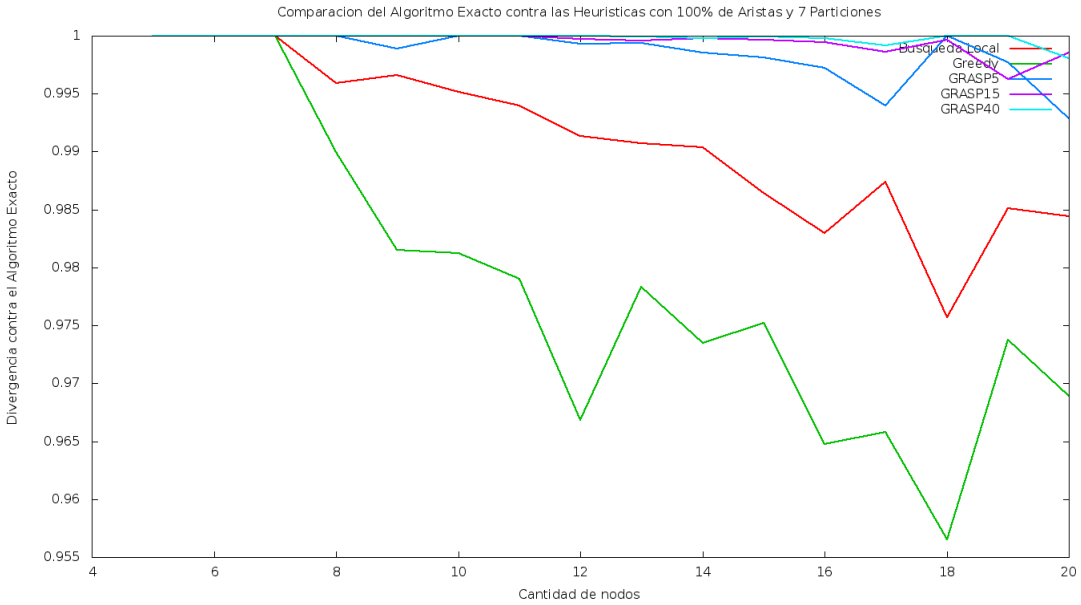

Figura 44: Distancias de las Heuristicas para K = 7 y 100 % de aristas

Podemos ver como a medida que aumenta la cantida de nodos los valores de divergencia de las Heuristicas comienzan a aumentar.

Otro punto fácilmente apreciable es que la Heurística Greedy es la que mayor distancia tiene con respecto a la solución Exacta para todos los valores de n.

Referencias