DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Algoritmos Sobre Grafos

Algoritmos 3

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Ricardo Colombo	156/08	ricardogcolombo@gmail.com.com
Federico Suarez	610/11	elgeniofederico@gmail.com
Juan Carlos Giudici	827/06	elchudi@gmail.com
Franco Negri	893/13	franconegri2004@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. Introducción y la relación con otros problemas

Para este trabajo práctico se nos pide, a partir de un grafo simple $G = (V, E)$ con pesos en las aristas, encontrar la k-particion tal que minimice el peso de las aristas intrapartición.

Para ello primero intentaremos relacionar este problema con otros problemas conocidos, pensaremos varias maneras de abordarlo y emprenderemos la búsqueda de algoritmos eficientes para resolverlo.

Como veremos luego, este problema es 'difícil' de resolver, por lo que para instancias grandes, el algoritmo dejará de ser viable, por lo que también desarrollaremos distintas heurísticas que resuelvan el problema de manera aproximada. Probaremos con una heurística golosa, dos heurísticas de busquedas locales y a partir de estas, haremos un GRASP.

1.1. Entrada y salida

Todos los algoritmos tomarán como entrada lo siguiente:

- Un entero **n** $\rightarrow$ Representará el numero de nodos del grafo G.
- Un entero **m** $\rightarrow$ Representará la cantidad de aristas del grafo.
- Un entero **m** $\rightarrow$ Representará la cantidad de conjuntos distintos que disponemos para poner los ejes.
- **m** filas donde cada fila i consta de 3 enteros:
  - **u v w** $\rightarrow$ donde **u** y **v** son los nodos adyacentes y **w** el peso de las aristas entre ellos.

La salida, por su parte, constará de una fila con:

- n enteros i_1 i_2 ... i_n

Donde cada i_k representa en qué conjunto se encuentra el nodo k

1.2. Ejemplo

Para ejemplificar el problema a resolver, pensemos en un grafo G con 4 nodos, 5 vértices y 2 particiones.

Supongamos ademas que los nodos estan conectados de la siguiente manera:

- $1 - 2$ con peso 2
- $1 - 3$ con peso 3
- $1 - 4$ con peso 3
- $2 - 4$ con peso 1
- $3 - 4$ con peso 2

A primera vista podríamos elegir el nodo con mas aristas, el nodo 1, y ponerlo en una partición obteniendo k-PMP = {{(1)},{}}. Por otro lado tenderíamos a poner nodos que no esten conectados en otra partición, por ejemplo los nodo 2 y 3 ya que estos no los conecta ninguna arista, obteniendo k-PMP = {{(1)},{(2),(3)}}. Ahora en nuestro ejemplo nos queda el nodo 4 y al tener $k = 2$ lo tenemos que poner en alguna, elegiremos la partición donde minimice el peso total, en este caso es indiferente, dado que agregar el nodo 4 a la primer partición suma 3 y a la segunda partición también suma 3, resultando k-PMP = {{(1),(4)},{(2),(3)}}.

De esta forma, mas adelante vamos a ver que un algoritmo goloso que intentase resolver este problema con un esquema similar a lo descripto podria llegar a generar soluciónes tan malas como se quiera.

1.3. Relación con el problema de Colorear un Grafo

Vamos a relacionar el problema de $k - PMP$ con el problema de coloreo. Supongamos que podemos encontrar un k-coloreo para los vértices del grafo G, entonces podríamos subdividir al conjunto de vértices V en k subgrupos según su color, es decir, en un mismo grupo sólo habrá vertices que compartan color, generando la siguiente partición: k-PMP = $\{ V_1, ..., V_k \}$.

Por la definición de coloreo, dos vértices de un mismo color no pueden tener aristas entre sí, por lo tanto los grupos que armamos son conjuntos independientes. Esto quiere decir que cada k-partición no tiene aristas intrapartición ya que cada una es un conjunto independiente, luego el peso total de la misma es 0. Además es un peso mínimo ya que no hay aristas con peso negativo, obteniendo la mejor solución a nuestro problema.

En primer instancia podemos relacionar el problema de $k-PMP$ con el problema de coloreo. Supongamos que podemos encontrar un k-coloreo para los vértices del grafo G, entonces podríamos subdividir al conjunto de vértices V en k subgrupos según su color, es decir, en un mismo grupo sólo habrá vertices que compartan color. Por la definición de coloreo, dos vértices de un mismo color no pueden tener arista entre sí, por lo tanto los grupos que armamos son conjuntos independientes. Esto quiere decir que mi k-partición no tendría aristas intrapartición ya que cada partición sería un conjunto independiente, y entonces el peso total de la misma sería 0. Y además sería un peso mínimo ya que no hay aristas con peso negativo y entonces tendríamos la solución a nuestro problema. Hasta aquí hemos visto que con un coloreo igual a k se obtiene la solución al problema, pero observemos que los conjuntos de la k-partición resultado no tienen que ser necesariamente no vacíos, lo que nos lleva a pensar que también nos bastaría conseguir un coloreo menor a k para resolver nuestro problema y ahora veremos cómo puede ser esto posible. Dado un k'-coloreo con $k' < k$, por lo dicho anteriormente podemos armarnos k' subgrupos de vértices agrupandolos por color, los cuales serán conjuntos independientes, es decir, no existirán aristas que incidan en dos nodos de un mismo grupo. Sin embargo, ya no me queda ningún vértice para meter en algún grupo y el problema me pide k particiones, por lo que me estarían faltando otras $k - k'$ particiones más que agregar a mi k-partición. Pero si recordamos nuestra observación que decía que las particiones no deben ser necesariamente no vacías, entonces podríamos agregar a nuestra k-partición $k - k'$ particiones de vértices vacías, con lo cual no estaríamos agregando ninguna arista intrapartición. Entonces mis k' particiones iniciales, por lo visto previamente, no tienen ninguna arista intrapartición y los conjuntos vacíos que agregué posteriormente claramente tampoco tienen aristas intrapartición, por lo que he llegado nuevamente a una k-partición de peso 0, la cual es solución de mi problema y además es óptima.

Por otro lado, dada una solución al problema de k-PMP de peso estrictamente mayor a 0 para un grafo G determinado, podemos afirmar que no existe un k-coloreo para ese grafo. Si existiera un k-coloreo para dicho grafo, entonces, por lo probado en el anterior párrafo, también existiría una k-PMP de peso 0 para tal grafo, lo cual es absurdo ya que partimos de una k-PMP de peso estrictamente mayor a 0.

1.4. Relación con el problema 3 del Trabajo Práctico 1

Si analizamos la situación con cierto detenimiento, podemos ver que el grafo G podría modelar perfectamente los productos químicos a transportar y sus respectivos coeficientes de peligrosidad de a pares, es decir, cada nodo representaría un producto y cada arista indicaría que existe un coeficiente de peligrosidad entre los productos o nodos sobre los que incide y su peso sería el valor de este coeficiente. Por otra parte, el peso de una partición se mide como la suma de las aristas intrapartición, lo cual es análogo al nivel de peligrosidad de un camión, que es la suma de las peligrosidades de a pares de los items transportados, con lo cual concluímos en que las particiones son un buen modelo de los camiones. La diferencia con el problema de los camiones, es que este último lo que busca minimizar es la cantidad de camiones utilizados para el transporte de los productos, mientras que en el problema k-PMP ya viene dado un número k de particiones y lo que se quiere minimizar es la suma de los pesos de todas las particiones, o sea, el equivalente a la suma de los niveles de peligrosidad de cada camión. Además, el nivel de peligrosidad de cada camión, para cumplir con las normas vigentes de seguridad, no puede superar un cierto umbral M y en cambio las particiones no tienen un límite de peso. Utilizando la k-PMP estaría resolviendo un problema similar al de los camiones, es decir, estaría buscando minimizar la suma de los

niveles de peligrosidad de los camiones y además tendría una cantidad limitada k de camiones para usar y además no existiría una cota de peligrosidad por camión.

2. Algoritmo Exacto

Dado un grafo simple G = (V , E) se quiere buscar la solución con un algoritmo exacto al problema de k-PMP, se realiza un algoritmo con la técnica de backtracking para obtener la misma.

2.1. Desarrollo

Para comenzar el desarrollo es necesario definir previamente cada una de las estructuras que se utilizan en el mismo. Sabemos que el grafo esta representado por un conjunto de vértices, cada uno de los mismos tiene una etiqueta en este caso es un numero desde 1 hasta n, siendo n la cantidad de vértices del grafo, a su vez tenemos un conjunto de tuplas que representan los ejes en el grafo. Toda esta información la volcamos a una matriz de adyacencias, donde cada posición se tiene 0 si las aristas no están conectadas o el valor del peso del eje en caso que estén conectados.

Una solución, llamada $solFinal$, será representada con un conjunto de k subconjuntos, donde cada uno contendrá las etiquetas de los vértices, disjuntos de a pares (un vertice pertenece a un único subconjunto) y el peso del conjunto será la suma de cada uno de los pesos de los subconjuntos, donde el peso del subconjunto se entienden a la suma de los pesos de las aristas que conecten dos vértices del mismo subconjunto.

Diremos que una k partición A es mejor que otra B si el peso total es menor, es decir si A es mejor que B entonces $peso(A) < peso(B)$.

La clave del algoritmo de Backtracking es que explorara de forma recursiva todas las posibles combinaciones de subconjuntos, de forma que en cada llamado recursivo intentara insertar un vértice a la solución que esta armando, en caso de no tener una mejor solución tratara de insertar ese vértice en otro subconjunto que no haya probado, así hasta haber probado todos los subconjuntos y sin ir mas lejos si se realiza este ejercicio estaremos realizando todas las combinaciones de los vértices en los k subconjuntos.

Nuestro $solFinal$ en principio va a ser el conjunto donde todos los vértices estén en el primer subconjunto.

Algorithm 1: backtracking(solParcial,solFinal,numeroVertice,cantidadSubConjuntos)

```
1: Si Puedo insertar
2:    para cada i desde 1 hasta cantidadSubConjuntos:
3:       agrego el numeroVertice al subconjunto i en SolParcial
4:       backtracking ( solParcial , solFinal, numeroVertice+1,k,adyacencias)
5:       saco el elemento que agregue al ultimo conjunto
6: Si Puedo insertar
7: devuelvo False
```

Ahora de esa manera lo único que estamos haciendo es probar todas las combinaciones, pero no estamos quedándonos con la mejor k-partición. Para esto debemos definir una función que nos diga cuándo una solución es mejor que otra y además debemos saber si ya insertamos todos los vértices.

Para lo cual definimos una función que devuelve un número indicando 0 si inserté todos los vértices llegando a otra solución, luego tengo que ver si es mejor que la que tengo hasta el momento y 1 si tengo que seguir insertando vértices.

Quedándonos de la siguiente manera:

Algorithm 2: entero check(adyacencias, solParcial,solFinal, numeroVertice,cantidadVertices)

```
1: SI numeroVertice=cantidadDeVertices
2:    devolver 0
3: SI NO
4:    devolver 1
```

El algoritmo modificado:

Algorithm 3: backtracking(solParcial,solFinal,numeroVertice,cantidadSubConjuntos,adyacencias, cantidadVertices)

```
1: entero resultadoCheck = check(adyacencias,solParcial,solFinal,numeroVertice,cantidadVertices)
2: si resultadoCheck = 0
3:    Remplazo mi solFinal por solParcial
4: Si NO
5:    para cada i desde 1 hasta cantidadSubConjuntos:
6:       agrego el numeroVertice al subconjunto i en SolParcial
7:       backtracking ( solParcial , solFinal, numeroVertice+1,k,adyacencias)
8:       saco el elemento que agregue al ultimo conjunto
9: devuelvo False
```

Con este algoritmo estaremos probando todas las combinaciones de todos los vértices en los k subconjuntos, en cada paso insertamos un vértice a un subconjunto y llamamos a la recursión con el siguiente vértice, una vez que vuelva de la recursión quitará el vértice del conjunto donde lo ingresó y seguirá el ciclo insertándolo en otro subconjunto y llamando a la recursión de nuevo.

2.2. Podas

Para mejorar los tiempos del algoritmo es necesario realizar algún tipo de validación intermedia para no probar casos que no nos van a llevar a una solución óptima, por ejemplo cuando estamos insertando vértices a la solución que ya tiene más peso que nuestra mejor solución o cuando nos quedan una cantidad de vértices es menor estricta a la cantidad de subconjuntos vacíos, estas podas son las que realizamos en nuestro algoritmo agregándolas a la función check al Backtracking de la siguiente manera:

Algorithm 4: numero check(adyacencias, solParcial,solFinal, numeroVertice,cantidadVertices)

```
1: SI peso(solParcial)> peso(solFinal)
2:    devolver 2
3: SI numeroVertice=cantidadDeVertices
4:    devolver 0
5: Si cantidadDeCajasVacias(solParcial)> cantidadVertices
6:    devolver 2
7: devolver 1
```

La función $cantidadDeCajasVacias$ se fija si algun conjunto no tiene.

De esta manera el algoritmo va a realizar las podas en las llamadas recursivas intermedias antes de insertar todos los vértices descartando casos que no son útiles para llegar a una solución óptima en menor tiempo.

Para improvisar una mejora en la poda al comparar los pesos de nuestras soluciones parcial y final, lo que realizamos es generar otra solución final asignando cada uno de los vértices de manera creciente a cada uno de las k subconjuntos diferentes y si esta otra solución es mejor a la solución que tiene todos los nodos en un solo conjunto la tomamos como solución final para nuestras comparaciones. De esta manera podrían tomarse otro tipo de soluciones para luego obtener la solución final de menor peso y podar más ramas del Backtracking.

Algorithm 5: backtracking(solParcial,solFinal,numeroVertice,cantidadSubConjuntos,adyacencias, cantidadVertices)

```
1: entero resultadoCheck = check(adyacencias,solParcial,solFinal,numeroVertice,cantidadVertices)
2: si resultadoCheck = 0
3:    Remplazo mi solFinal por solParcial
4: si resultadoCheck = 2
5:    devuelvo false porque mi solParcial es peor que mi solFinal o tengo mas cajas vacías que vertices para poner l
6: Si NO
7:    para cada i desde 1 hasta cantidadSubConjuntos:
8:       agrego el numeroVertice al subconjunto i en SolParcial
9:       backtracking ( solParcial , solFinal, numeroVertice+1,k,adyacencias)
10:      saco el elemento que agregue al ultimo conjunto
11: devuelvo False
```

2.3. Complejidad

Dado que el algoritmo exacto sin las podas prueba todas las soluciones del problema esto hace que sea poco eficiente en términos de tiempo a comparación de las heurísticas que iremos viendo a lo largo del informe. Pero conociendo las soluciones que se van a ir armando pueden definirse podas para evitar seguir resolviendo ramas de la solucion que no me van a llevar a una solución óptima o a una solución que no es mejor que la que ya formé hasta cierto punto. Dado el algoritmo exacto mostraremos las complejidades de las diferentes partes del mismo para obtener cuál sería la complejidad final.

Algorithm 6: numero check(adyacencias, solParcial,solFinal, numeroVertice,cantidadVertices)

```
1: SI peso(solParcial) ¿peso(solFinal) O(1)
2:    devolver 2
3: SI numeroVertice=cantidadDeVertices O(1)
4:    devolver 0
5: Si cantidadDeCajasVacias(solParcial) > cantidadVertices O(k)
6:    devolver 2
7: devolver 1
```

Para la funcion check la complejidad final sera $\mathcal{O}(1)$ + $\mathcal{O}(1)$ + $\mathcal{O}(k)$ dando como complejidad total $\mathcal{O}(k)$.

El ciclo de las lineas 7-10 se ejecuta k veces en cada llamado iterativo, cada vez que se llama a la función backtracking es por un vértice diferente. Para el primer nodo tenemos k conjuntos para probar, luego con el segundo vértice tenemos para cada uno de esos conjuntos donde coloqué el primer nodo k alternativas de conjuntos para probar quedándonos k^2 combinaciones para esos dos vértices. Luego para cada una de esas k^2 combinaciones tenemos k conjuntos quedando k^3 combinaciones, y así seguirá en las llamadas recursivas por todos los vértices. Con lo cual con el vértice n tendremos k^n combinaciones para este vértice.

Por lo tanto la complejidad total del algortimo es del orden de $\mathcal{O}(k^n)$ siendo k la cantidad de subconjuntos y n la cantidad de vértices del grafo.

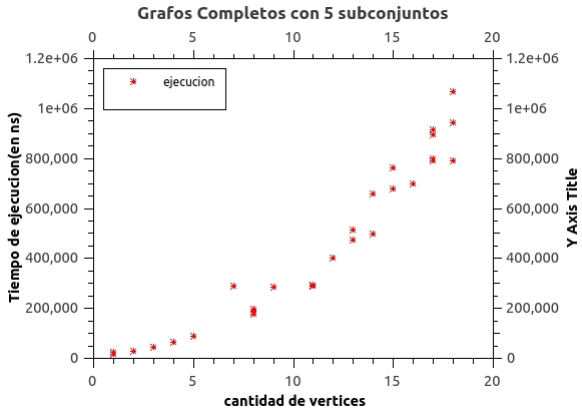
2.4. Experimentacion

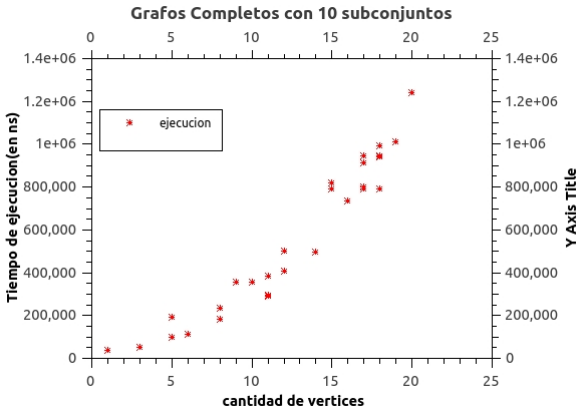
Para el análisis de este algoritmo se generaron 30 instancias de manera random de grafos completos con vértices entre 1 y 20 elegidos de manera aleatoria mediante un script en python con la función ran-

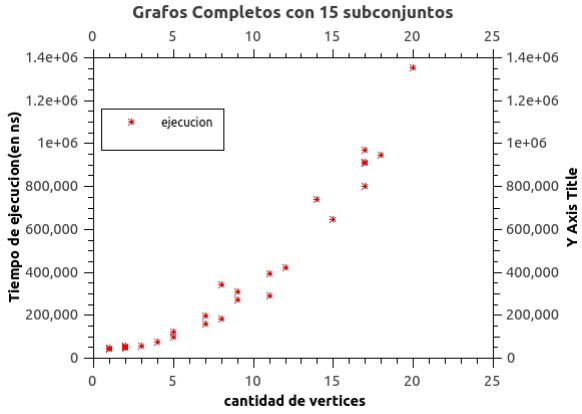
Algorithm 7: backtracking(solParcial,solFinal,numeroVertice,cantidadSubConjuntos,adyacencias, cantidadVertices)

1: entero resultadoCheck = check(adyacencias,solParcial,solFinal,numeroVertice,cantidadVertices)$\mathcal{O}(k)$
2: si resultadoCheck = 0 $\mathcal{O}(1)$
3: Remplazo mi solFinal por solParcial $\mathcal{O}(k)$
4: si resultadoCheck = 2 $\mathcal{O}(1)$
5: devuelvo false porque mi solParcial es peor que mi solFinal o tengo mas cajas vacías que vertices para poner $\mathcal{O}(1)$
6: Si NO
7: para cada i desde 1 hasta cantidadSubConjuntos:
8: agrego el numeroVertice al subconjunto i en SolParcial
9: backtracking (solParcial , solFinal, numeroVertice+1,k,adyacencias)
10: saco el elemento que agregue al ultimo conjunto
11: devuelvo False

dint. Luego para esas instancias se corrió el algoritmo asignando 5, 10 y 15 subconjuntos para ver como se comportaba el algoritmo, obteniéndose como resultado lo siguiente:

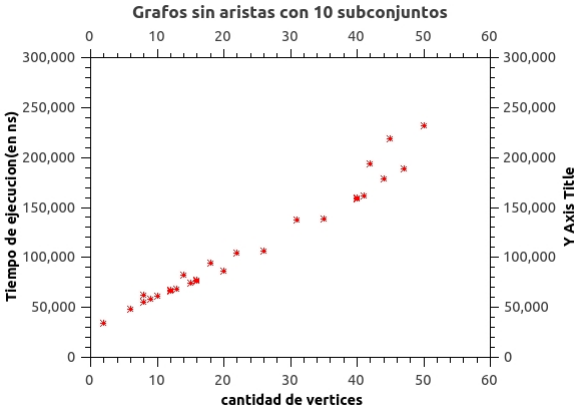




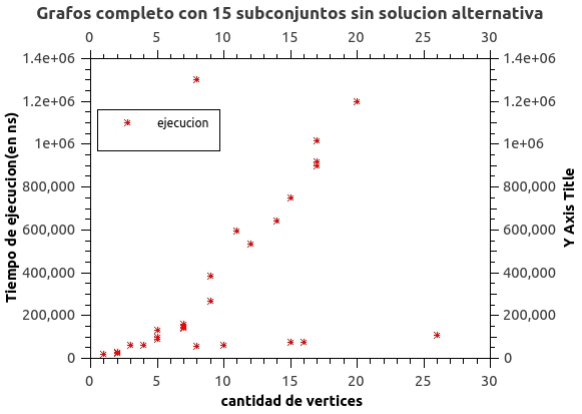


Se puede ver cómo la curva se hace más pronunciada al cambiar la cantidad de subconjuntos, con esto se ve que al cambiar el k la complejidad va aumentando de manera exponencial.

Por otro lado realizamos un analisis sobre las podas, para esto primero generamos 30 instancias de grafos para ver cómo se comportaba el algoritmo con grafos sin aristas, obteniendo el siguiente resultado:



Como puede verse en el gráfico, los tiempos de ejecución son similares a una lineal, esto se debe a que como la solución inicial tiene peso 0 ya que no tiene aristas, en cada llamado recursivo se devuelve false porque no se obtiene nada mejor en cuestión de peso del conjunto. Por otro lado como mencionamos en el desarrollo del algoritmo, se agregó una genacion de una solucion inicial alternativa a poner todos los vértices en un solo subconjunto, por lo que realizamos una experimentación para ver cómo afectaba la no presencia de esta solución alternativa, corrimos las mismas instancias de grafos para 15 subconjuntos que corrimos en los primeros test donde se encontraba esta solución alternativa y obtuvimos lo siguiente:



Como puede observarse los tiempos empeoran ya que la primera solución a nuestro problema es el grafo con todos las aristas y esto puede no podar muchas ramas del backtracking que con la solución alternativa sí se están podando. Concluyendo que podrían mejorar los tiempos si estamos eligiendo una buena solucion inicial.

3. Heuristica Golosa Constructiva

3.1. Idea general

Un primer intento para conseguir una solución a este problema es utilizar un algoritmo goloso. La idea del mismo es sencilla, numeramos los nodos de 1 a n, y luego tomando de a uno los agregamos a alguno de los conjuntos de 1 a k intentando que sume a la solución el menor peso posible.

Más formalizado el algoritmo quedará de la siguiente manera:

Algorithm 8: Goloso()

1: Numero los vertices de 1 a n
2: Creo una cantidad k de conjuntos donde iré guardando vertices
3: Para cada nodo i de 1 a n:
4: Para cada conjunto
5: Sumo todos los pesos de las aristas de (i,j) con j los vertices que estan en el conjunto
6: Agrego la el vertice i para el cual la suma dio menor
7: Imprimo por salida estandar la respuesta

Claramente este algoritmo no devuelve la solución exacta, y como se verá mas adelante, la solución que devuelve puede estar tan lejos como se quiera de la optima, lo que lo hace un algoritmo no muy bueno.

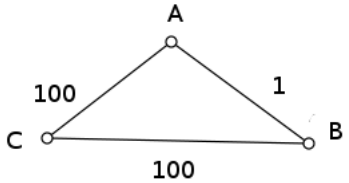
El análisis de complejidad es sencillo, se itera por cada vértice sobre cada conjunto. Dado que cada nodo sólo estará en un conjunto, entre todos los conjuntos a lo sumo tendrán n nodos, lo que hace que se deba iterar n veces a lo sumo sobre n nodos, luego la complejidad será $O(n^2)$, por lo que, al menos en lo que respecta a tiempos, es ampliamente superior que el algoritmo exacto.

Por lo tanto, dada la baja complejidad de este algoritmo, podría utilizarse como una cota superior, si bien algo grosera, para la solución.

3.2. Problemas del Algoritmo Goloso

Como ya adelantamos, la solución para este algoritmo no siempre da una solución exacta, y puede ser tan mala como se quiera. Esto surge principalmente de darle un orden a los nodos, como mostraremos en el siguiente ejemplo.

Supongamos que tenemos un grafo como el que se muestra en la figura y $K = 2$



Es claro que la mejor solución posible es poner en uno de los conjuntos a A y B y en el otro a C, así la suma intrapartición es 1.

Sin embargo, supongamos que nuestro algoritmo goloso toma como primer nodo al nodo A, dado que no hay otros nodos, lo agrega en cualquiera de los dos conjuntos y se obtiene peso 0. Ahora supongamos que el algoritmo toma el nodo B, si lo pone en el mismo conjunto que el nodo A obtiene peso 1, si lo pone en el otro conjunto obtiene peso 0, así que lo pone en el otro conjunto. Pero ahora falta agregar el nodo C, y agregándolo en cualquiera de los dos conjuntos se obtiene peso 100. Así que la solución para k-PMP que encontrará el algoritmo goloso será 100. De ahí se desprende que cambiando ambos pesos 100 por cualquier valor puede obtenerse una solución tan mala como uno quiera.

Sin embargo, se puede observarse otra cosa de este algoritmo, si se hubiese tomado el nodo C como primer nodo o como segundo nodo, se hubiera llegado a la solución óptima. De aquí surge la idea de que si se eligieran diferentes órdenes para los nodos y se corriera el algoritmo para cada uno de estos

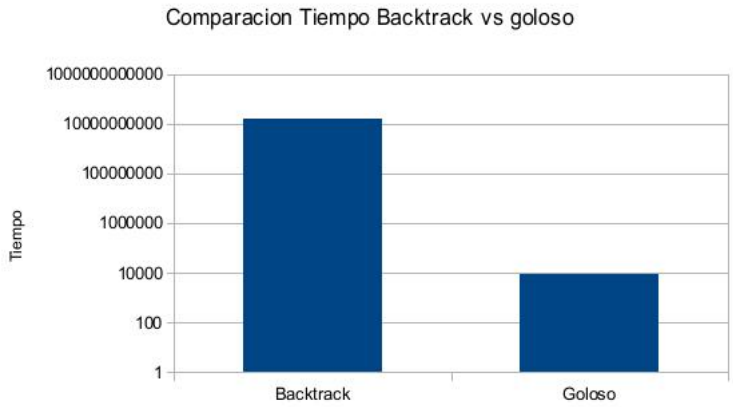
órdenes, podría obtenerse diferentes cotas, y con un poco de suerte en alguna de ellas no sucederá este caso que acabamos de ver.

Esta idea la utilizaremos más adelante para el GRASP, correremos con distintos órdenes de nodos el algoritmo goloso de manera tal de obtener en cada una de estas iteraciones una respuesta diferente y posiblemente mejor que la anterior.

3.3. Testing

En esta sección, realizaremos diferentes experimentos para comprobar que el algoritmo escala de acuerdo a la complejidad teórica, así como también, veremos qué tan buenos resultados obtenemos en la práctica para este algoritmo.

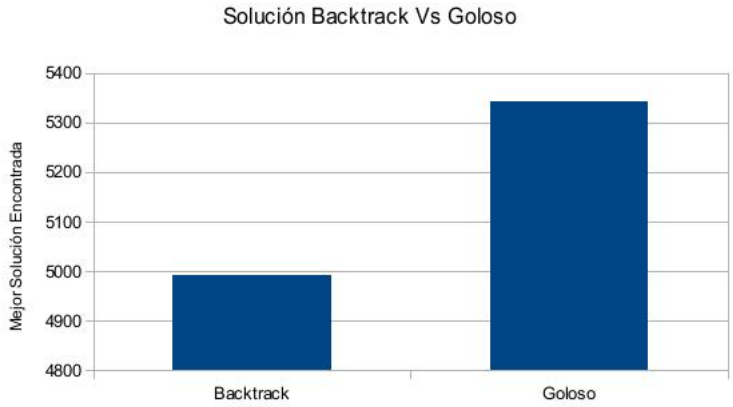
Primero se corre el algoritmo con instancias de 25 a 200 nodos y se grafican los tiempos obtenidos:



Dividiendo por una función $f(i) = i^2$ obtenemos que una linea constante, por lo que podemos comprobar así de manera empírica, que el resultado obtenido es efectivamente cuadrático.

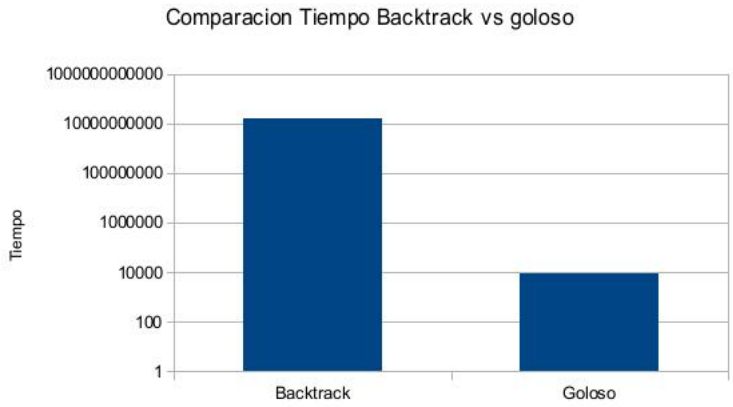
Ahora tomaremos grafos completos de 23 nodos (El backtrack es el factor limitante, de tomar más nodos empieza a tardar un tiempo excesivo), y compararemos los resultados dados tanto por la heurística golosa como por el backtrack. Cabe notar que las aristas de los grafos serán elegidas al azar entre un numero del 1 al 100 para no tomar casos demasiado particulares que favorezcan a uno u otro algoritmo.

Tomando el promedio, las respuestas que se obtuvieron son las siguientes:



Como puede verse el algoritmo goloso, en promedio, da soluciónes significativamente peores que la respuesta exacta.

De las anteriores 100 muestras también se calculó el tiempo que se tardó en obtener las respuestas, nuevamente promediando se obtiene:



Donde se ve que el algoritmo goloso es 6 órdenes de magnitud más rapido que el backtrack, incluso para una instancia modesta de 23 nodos.

4. Heuristica de Busqueda Local

5. Introducción

En esta sección se implementarán heurísticas de búsqueda local para tratar de resolver el problema de $k - PMP$, intentando con diferentes métodos para alcanzar una solución y diferentes vecindades.

Para ello, partiendo de una solución generada al azar, se intentará a través de sucesivas iteraciones analizar cierta vecindad para intentar mejorar la solución existente y aproximarse más a una 'buena' solución en un tiempo aceptable.

Por lo tanto debemos tener en cuenta de no hacer las vecindades ni muy grandes, ya que esto puede llevar a una pérdida de performance, ni muy pequeñas, ya que esto puede llevar a que el algoritmo explore sólo una pequeña cantidad de soluciones y devuelva una solución muy alejada del óptimo.

Como primera idea para una vecindad tomaremos cada nodo del grafo, vemos cuánto peso agrega en la suma intraconjunto en que se encuentra, lo quitaremos de este conjunto e intentamos meterlo en todos los demás, viendo si en alguno logra minimizar esta suma. En caso afirmativo, lo sacamos de su antiguo conjunto y lo ponemos en el nuevo. Realizamos esto hasta que deja de ser posible mejorar la solución y en este punto la devolvemos.

Otra vecindad que plantearemos será buscar el nodo que más peso está generando en la suma intra-partición, quitarlo de la partición donde se encuentra y agregarlo a alguna otra.

Finalmente se implementará una tercera busqueda local que quite dos nodos que están en una misma partición e intente buscar alguna otra donde los mismos sumen un menor peso intrapartición.

Los algoritmos escritos de manera formal serán así:

Algorithm 9: Busqueda1(SoluciónInicial)

1: Guardo SoluciónInicial en SolucionPrevia
2: Mientras en el paso anterior se haya mejorado SolucionPrevia
3: Para todo nodo i de 1 a n del grafo $\mathcal{O}(n)$
4: Miro que peso agrega el nodo i en el conjunto asignado por SolucionPrevia $\mathcal{O}(n)$
5: Miro que peso agrega el nodo i quitandolo del conjunto asignado y poniendolo en los demas $\mathcal{O}(n)$
6: Si algun conjunto M obtengo un peso menor, modifico SolucionPrevia $\mathcal{O}(n^2)$
7: Asigno i al conjunto M $\mathcal{O}(1)$
8: Itero

En el paso "Miro qué peso agrega el nodo i quitándolo del conjunto asignado y poniéndolo en los demás", que dado que todo nodo puede estar sólo en un conjunto, iterando sobre todos los elementos de los conjuntos, al menos voy a iterar n veces, de ahí la complejidad $\mathcal{O}(n)$.

Dada nuestra implementación de conjunto con listas simplemente enlazadas, encontrar un elemento y borrarlo cuesta $\mathcal{O}(n^2)$ por esta razón modificar solución previa tendrá esta complejidad. Utilizando listas doblemente enlazadas esto se hubiera podido reducir a $\mathcal{O}(n)$ pero por falta de tiempo no se implementó.

Algorithm 10: Busqueda2(SoluciónInicial)

Guardo SoluciónInicial en SolucionPrevia
Mientras en el paso anterior se haya mejorado SolucionPrevia
Asigno $j = 1$
  Tomo el j-esimo nodo mas pesado de la SolucionPrevia, al que llamo i $\mathcal{O}(1)$
  Miro que peso agrega el nodo i en el conjunto asignado por SolucionPrevia $\mathcal{O}(n)$
  Miro que peso agrega el nodo i quitandolo del conjunto asignado y poniendolo en los demas $\mathcal{O}(n)$
    Si En algun conjunto M obtengo un peso menor
      modifico SolucionPrevia y asigno i al conjunto M $\mathcal{O}(n^2)$
      Itero
    Si no, sumo 1 a j
    Si $j == n$
      Devuelvo SolucionPrevia
    Si no
      Itero

Algorithm 11: Busqueda3(SoluciónInicial)

Guardo SoluciónInicial en SolucionPrevia
Mientras en el paso anterior se haya mejorado SolucionPrevia
  Para todo nodo i de 1 a n del grafo $\mathcal{O}(n)$
    Tomo otro nodo k del mismo conjunto de i $\mathcal{O}(1)$
    Miro que peso agrega el nodo i y k en el conjunto asignado por SolucionPrevia $\mathcal{O}(n)$
    Miro que peso agrega el nodo i y k quitandolo del conjunto asignado y poniendolo en los demas $\mathcal{O}(n)$
      Si algun conjunto M obtengo un peso menor, modifico SolucionPrevia $\mathcal{O}(n^2)$
      Asigno i y k al conjunto M $\mathcal{O}(1)$
      Itero

6. Análisis de Complejidades

Aquí analizaremos las complejidades de los diferentes algoritmos. Para cada uno de ellos elegimos una implementación con matriz de adyacencias para modelar el peso de las aristas y un vector de longitud variable para implementar los diferentes conjuntos.

Para el primero, cada paso de búsqueda local tendrá una complejidad de peor caso de $O(n(n+n+n^2))$. Ahora si k es mayor a n quiere decir que hay más subconjuntos disponibles que nodos, lo que llevaría a una solución trivial donde cada nodo va en un subconjunto diferente y la solución para $k - PMP$ sería 0. Luego podemos acotar a k por n con lo que se obtendría una complejidad igual a $O(n^3)$

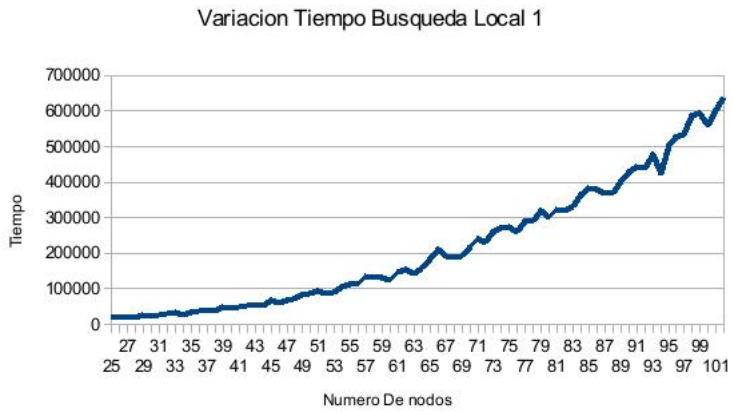
Para el segundo algoritmo, por cada iteración del algoritmo la complejidad será $O(n^2)$ para calcular el nodo con mayor peso del grafo. $O(n)$ para determinar si existe una mejor partición donde este nodo pueda estar y, en el caso de que exista, $O(n^2)$ para quitarlo de la partición anterior y agregarlo a la nueva. Luego, nuevamente acotando k por n, se obtiene una complejidad para cada paso de la iteración de $O(n^2)$.

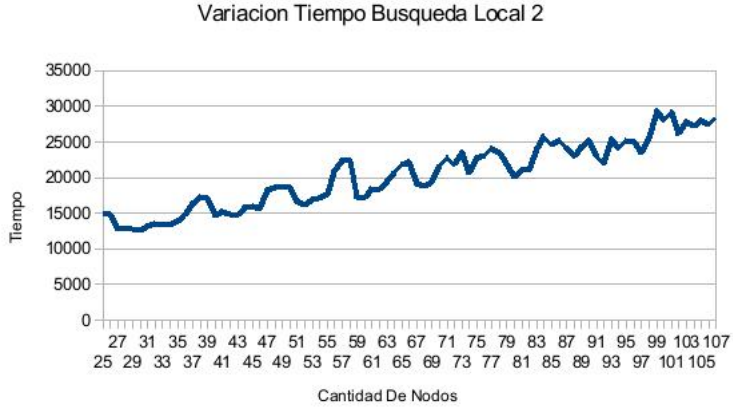
El tercer algoritmo es simplemente el primer algoritmo pero esta vez para cada nodo además tomo un vecino, y realizo el mismo procedimiento que antes, esto para cada vecino, suponiendo que en el peor caso, para un nodo todos los otros nodos esten en el mismo conjunto, se tendrá que realizar el procedimiento de antes la misma cantidad de veces solo que ahora para dos nodos distintos, luego la complejidad continúa siendo $O(n^2(n + nk + n^2))$. Acotando nuevamente k, obtenemos que el algoritmo es: $O(n^3)$

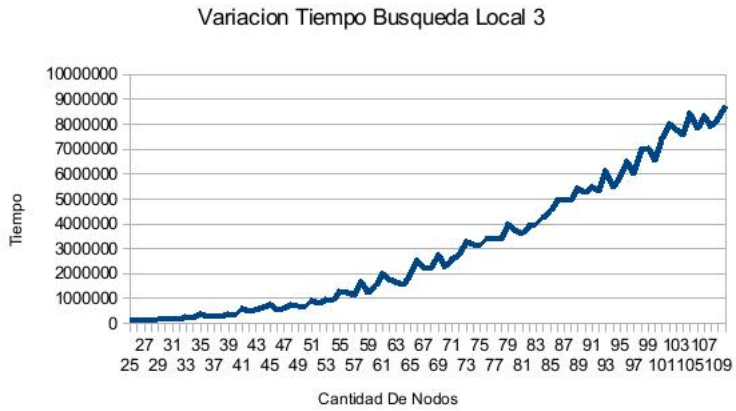
7. Testing

En esta sección comprobaremos de manera empírica las complejidades antes calculadas y luego una comparación entre las tres diferentes heurísticas.

Para ello, al igual que para el algoritmo goloso, tomamos grafos completos con 25 a 100 nodos, y comparamos los diferentes tiempos obtenidos:

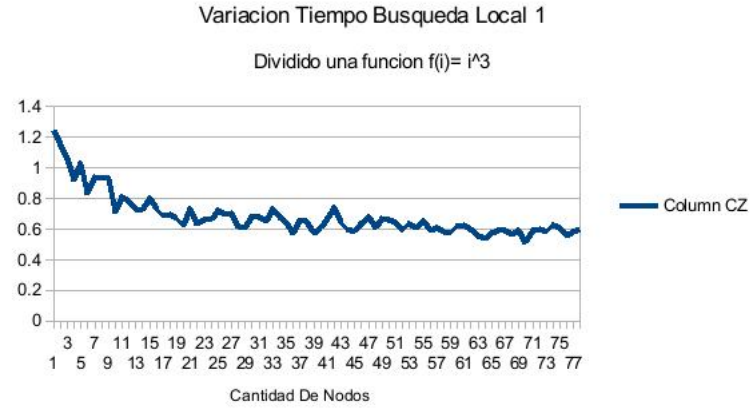


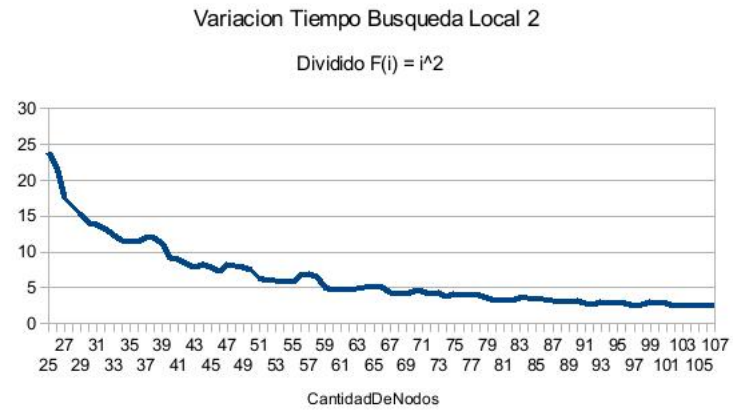


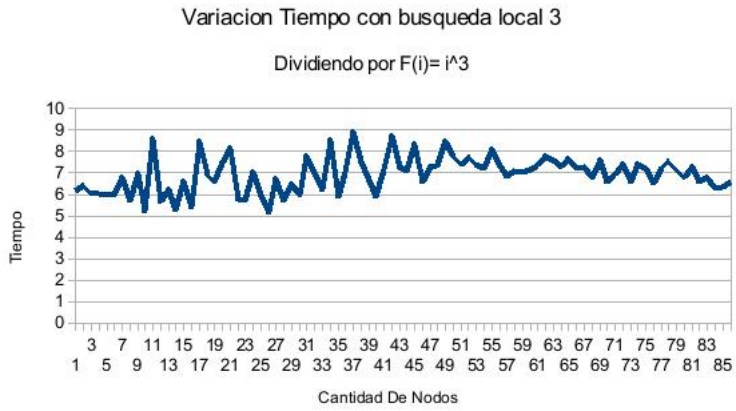


Puede verse que si bien existe cierto ruido en la toma de muestras, causado muy posiblemente por el hecho de que para cada grafo, el número de veces que es posible continuar mejorando la solución sea variable y poco controlable. Existe una tendencia muy marcada en los tres algoritmos.

Para poner más en evidencia esto, dividiremos por las complejidades teóricas para así hacer más evidente este patrón:

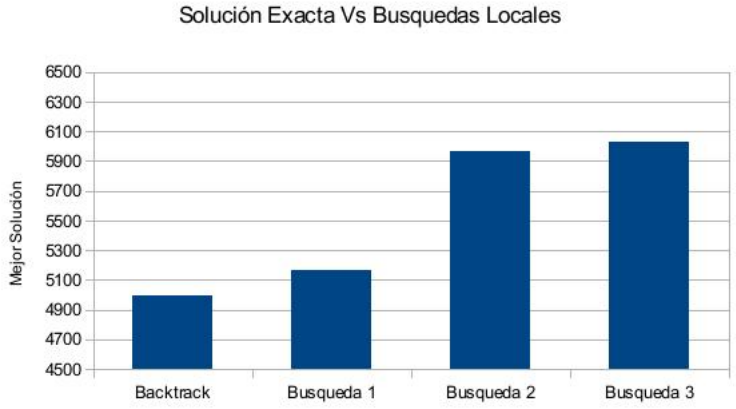






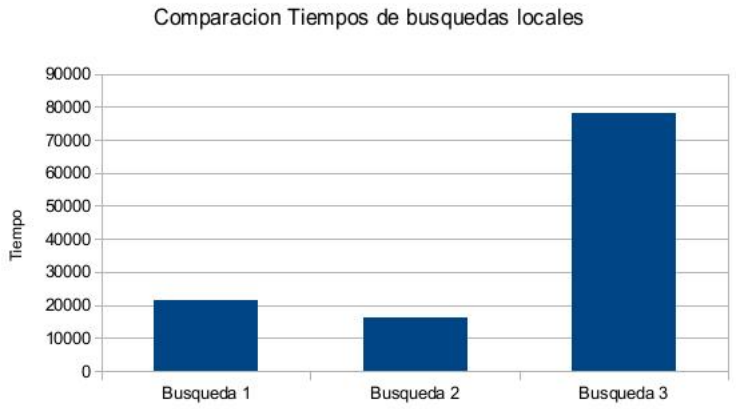
Como puede verse, en los tres casos las heurísticas están acotadas por las complejidades calculadas en el apartado anterior.

Veamos como se comportan las busquedas locales contra el algoritmo exactos para K_23 con peso en las aristas entre 1 a 100 elegidos de forma uniforme, tomaremos 100 muestras y compararemos los resultados contra el algoritmo de backtracking.



Puede verse que el que mejor aproxima a las soluciones reales es la primera búsqueda local. Y no solo eso, de los 100 casos tomados, la búsqueda local 1 logra encontrar la solución exacta a 13 de las instancias! mientras que las otras dos heurísticas en ninguno de los casos logran encontrar la solución exacta.

Además, para estas instancias se realiza a modo comparativo un promedio de los tiempos que tardan en encontrar la solución, esto es lo que se obtiene:



Por lo que en primera instancia, el algoritmo de búsqueda local 1 es bastantemente superior en todo sentido a los otros dos, ya que encuentra soluciones exactas y obtiene tiempos empíricos mucho mejores que los otros dos algoritmos.

8. GRASP

9. Idea

Para la metaheurística de GRASP, tomaremos el algorimto goloso previamente implementado y lo combinaremos con las diferentes búsquedas locales tambien previamente implementadas.

La idea es la siguiente, para cada paso del algoritmo, corremos primero la heurística golosa con un orden aleatorio en sus nodos. Este orden aleatorio, como ya hemos demostrado en el apartado del algoritmo goloso, podrá ir produciendo diferentes respuestas, cada una con una mejor o peor aproximación a la respuesta exacta.

Luego a la solución obtenida por el algoritmo goloso se le aplicarán una o varias de las búsquedas locales implementadas en un intento de aproximar más aún la respuesta al óptimo.

Como primer criterio de corte el algoritmo se correrá un número finito de veces, que será determinado en el momento de experimentación.

Como segundo criterio de corte se realizarán estos mismos pasos hasta que luego de un número x a determinar de intentos no haya sido posible mejorar la solución. En este punto se entrega la mejor respuesta obtenida hasta el momento.

Tras realizar diversas experimentaciones nos quedamos con dos versiones de GRASP, una que en el paso de búsqueda local usa la búsqueda local 2, y otra que en el paso de búsqueda local primero utiliza una búsqueda local 1 y luego una búsqueda local 3.

A continuación se formaliza de manera más precisa el algoritmo:

Algorithm 12: GRASP1(SoluciónInicial)

```
1: while(1)
2:     Genero una solucion incial a partir del Algoritmo Goloso
3:     Utilizo busqueda local 2 para mejorar la solución incial
4:     Si conseguí una mejor solucion que antes, la guardo
5:     Si tras 1000 iteraciones no se pudo consegir una mejor solución
6:         Devuelvo la solución
```

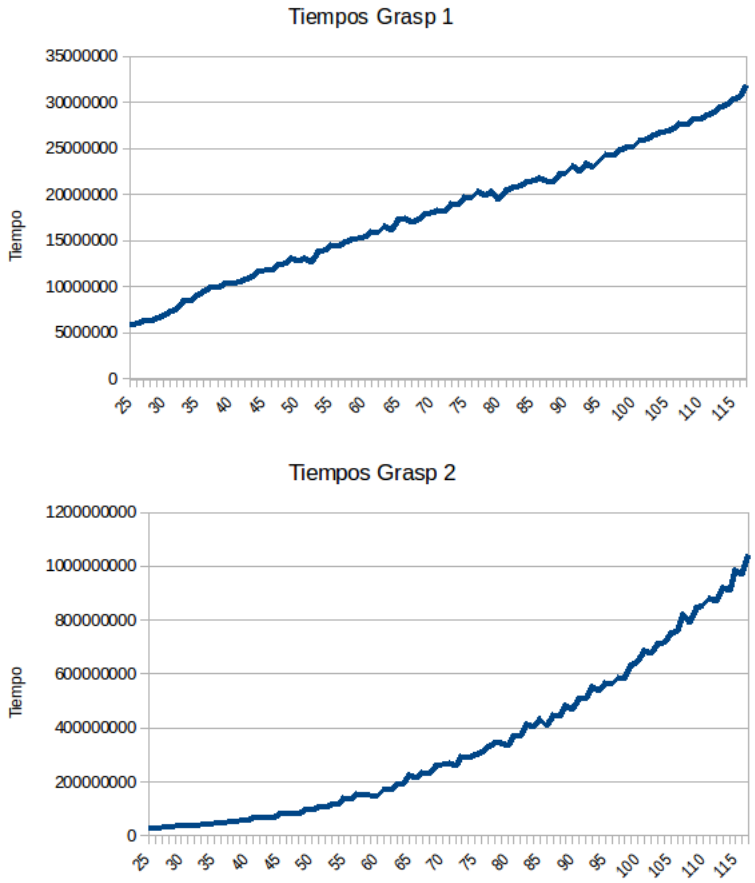
Algorithm 13: GRASP2(SoluciónInicial)

```
1: while(1)
2:     Genero una solucion incial a partir del Algoritmo Goloso
3:     Utilizo busqueda local 1 para mejorar la solución
4:     Utilizo busqueda local 3 para mejorar la solución
5:     Si conseguí una mejor solucion que antes, la guardo
6:     Si tras 1000 iteraciones no se pudo consegir una mejor solución
7:         Devuelvo la mejor solución
```

10. Experimentación

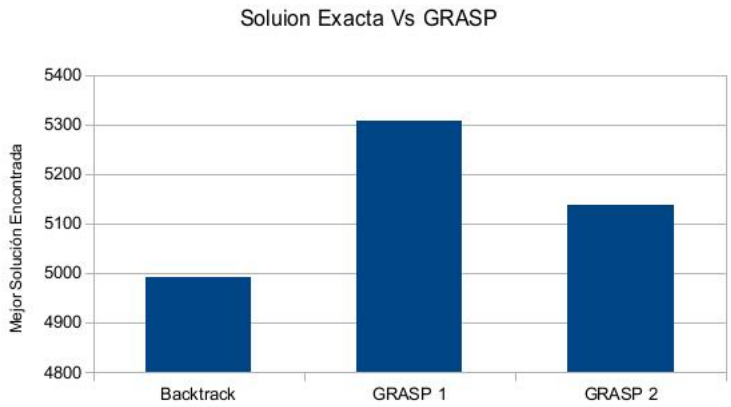
Ahora haremos una experimentación donde compararemos los tiempos de ejecución y los resultados del algoritmo de GRASP contra la solución exacta obtenida por backtracking.

Primero tomamos grafos de 25 a 100 nodos completos con aristas uniformemente distribuidas de 1 a 100 y analizamos los tiempos que dan los algoritmos:



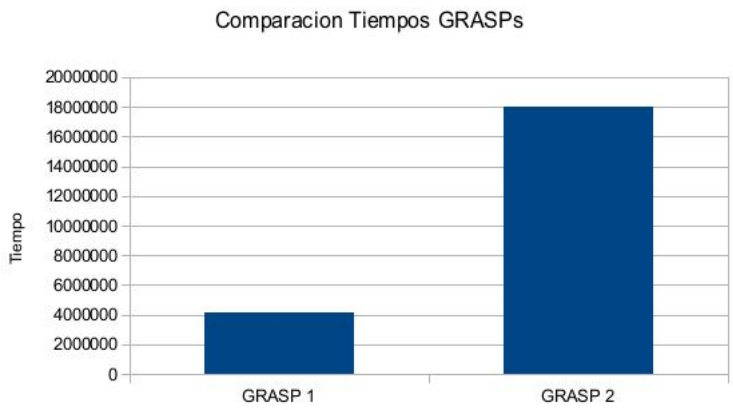
Si bien se puede ver que no es exponencial, estas metaheurísticas dependen tanto de factores inherentes al algoritmo que se vuelve dificil de encontrar una complejidad de peor caso.

Además, se comparan como en los casos anteriores, las soluciones dadas por las dos heurísticas contra la solución exacta. La experimentación para este caso es igual a la realizada para el algoritmo goloso y para las búsquedas locales:



Puede verse aquí que la mejor de las heurísticas es la del GRASP 2, que no sólo encuentra soluciones más cercanas a la original, sino que en 17 de los 100 casos, logra encontrar la solución exacta. Esto, seguramente debido a que usa la búsqueda local 1, que como ya habíamos constatado en su experimentación, tambien lograba alcanzar soluciones exactas.

Finalmente, para los mismos casos de prueba antes vistos, también se miden los tiempos que tardan en encontrar la respuesta, obteniéndose los siguientes resultados:



Puede verse que el hecho de encontrar soluciones exactas tiene su contrapartida en el hecho de que la metaheurística tarde significativamente más que la que no los encuentra.

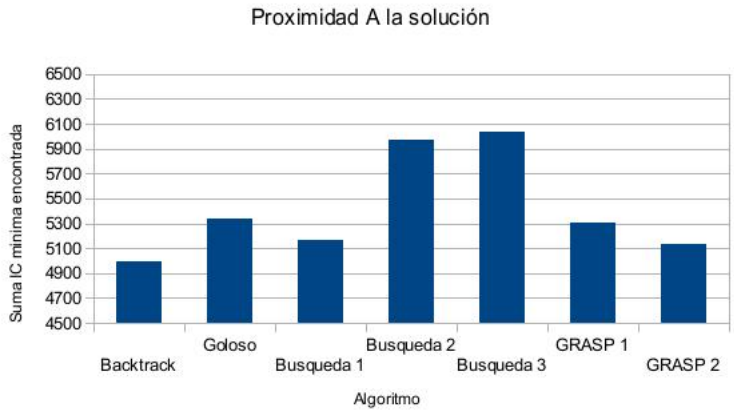
11. Concluciones Finales

12. Experimentación Final

Para concluir este trabajo, tomamos todos los algoritmos antes vistos y realizamos una experimentación conjunta entre todos para mostrar como se comparan entre sí.

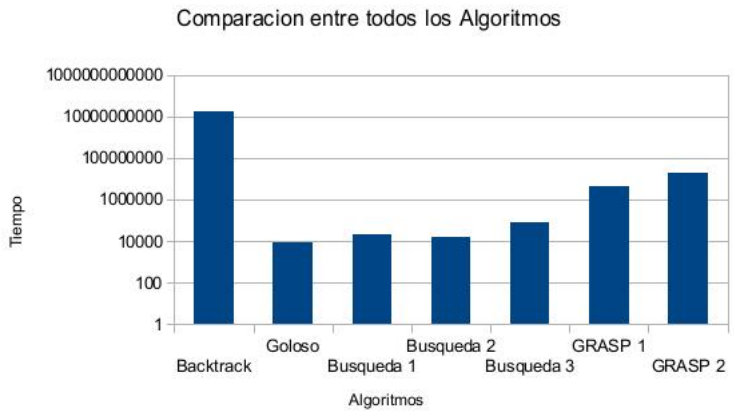
La primera de las experimentaciones se realizará con grafos completos de 23 vertices con pesos de las aristas elegidas al azar de forma del 1 al 100. Sobre cada una de estas instancias se correrán todos los algoritmos antes descriptos, se tomarán cual es la mejor solucion encontrada por el mismo y cuanto tarda en encontrarla y en base a los resultados obtenidos se graficarán en una tabla comparativa donde puedan verse los resultados.

De esta experimentación, se obitiene que:



Nuevamente puede observarse que las mejores heuristicas son las de busqueda local 1 y GRASP 2.

Y los tiempos obtenidos son:



13. Conclusión

Concluimos que dado un problema como $k - PMP$ el cual no conocemos algoritmos polinomiales para resolverlo de forma exacta, podemos utilizar las distintas técnicas algorítmicas para dar una solución en un tiempo razonable con la contrapartida de relajar los requerimientos de la misma.

Vimos que la técnica de Metaheurística GRASP es una combinación válida entre distintos esquemas de algoritmos y trata de combinarlas para explorar el espacio de soluciones de una manera eficiente. En nuestro caso particular vimos que la implementación de GRASP 2 es la que mejores soluciones encontró.

Como trabajo adicional habria que hacer pruebas más extensas en distintas familias de grafos conocidas para ver si nuestro algoritmo tiene casos patológicos aun no descubiertos los cuales su solución es tan mala como uno quisiera.

14. Aclaraciones

14.1. Medicion de los tiempos

Para este tp como trabajamos bajo el lenguaje de programacion C++, decidimos calcular los tiempos utilizando 'chrono' de la libreria standard de c++ (chrono.h) que nos permite calcular el tiempo al principio del algoritmo y al final, y devolver la resta en la unidad de tiempo que deseamos.