DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo práctico 3

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Sebastián Fernández Ledesma	392/06	sfernandezledesma@gmail.com
Fernando Gasperi Jabalera	56/09	fgasperijabalera@gmail.com
Maximiliano Wortman	892/10	maxifwortman@gmail.com
Santiago Camacho	110/09	santicamacho90@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

Introducción

Objetivos

El objetivo de este TP es analizar el problema de grafos conocido como K-PMP y algunas técnicas algorítmicas que existen para resolver bien un conjunto de instancias o el problema en general. A lo largo de este TP, vamos a analizar primero que es el problema enunciado y luego que formas encontramos para aproximarnos a una solución para ese problema.

Descripción del problema

El enunciado del problema lo describe así:

Dado un grafo G = (V,E), una función F:G.V $\to \mathbb{R} \geq 0$, función de peso asociados a sus ejes y dada una particion P del conjunto de nodos, se define las aristas intrapartición como el conjunto de aristas que tienen ambos extremos en un mismo conjunto de alguno de los elementos de P. Una k-partición es un partición de los nodos de G, que tiene exactamente k subconjuntos, los cuales podrian ser vacios. Luego el peso de un k-partición es la suma de los pesos de las aristas intrapartición. El problema k-PMP pide encontrar un k-partición en G de peso mínimo.

Formalización del problema a resolver

Dado un grafo G=(V,E)

Dada una función F:G.E$\to \mathbb{R} \geq 0$

Dado un k:Nat, k ≥ 1.

Se define una particion S:Conj(Conj(Nodo)) en k-conjuntos de nodos de G como:

$$KP(G,k,S) == TRUE \implies |S| = k \wedge (\forall v \in G.V)(\exists S' \in S)(v \in S') \wedge$$
$$(\forall p, p' : Conj(Nodo))(p \neq p' \wedge p \in S \wedge p' \in S \implies p \cap p' = \emptyset \wedge p \subset G.V \wedge p' \subset G.V)$$

Se definen el conjunto de aristas intrapartición C de un un k-partición S de G como:

$$AI(S) == C \wedge (\forall e = (v1, v2), \{v1, v2\} \subseteq G.E)(e \in C \implies \exists s' \in S / (\{v1, v2\} \subseteq s')$$

Luego el peso P de una k-partición S es la sumatoria de los pesos del conjunto C de aristas intraparticion.

$$AI(S) = C \implies P(C) = \sum_{i}^{|C|} F(C_i)$$

Se solicita hallar S, una k-partición de G tal que sea de peso mínimo:

$$(\forall S', KP(G,k,S') \implies P(AI(S')) \geq P(AI(S)))$$

1. Desarrollo de los puntos

1.1. Biohazard

En primer lugar es necesario recordar brevemente de qué trataba el problema *Biohazard*:

> Una empresa de logística necesita transportar en camiones n productos químicos desde una fábrica hacia un depósito seguro. Cualquier producto puede ser transportado en cualquiera de los camiones y para cada par de productos p_i y p_j se conoce de antemano el coeficiente de peligrosidad $h_{i,j}$ que conlleva transportar dichos productos en el mismo camión. Los camiones tiene un umbral de peligrosidad que no puede ser superado. Respetando esta restricción se desea averigüar la mínima cantidad de camiones necesaria para transportar los productos.

Ambos problemas pertenecen a una clase de problemas conocida como optimización combinatoria. La misma puede describirse rápidamente como la búsqueda de un objeto óptimo dentro de un conjunto finito de objetos, donde el criterio de optimalidad está determinado por la maximización o la minimización de una función específica. Una forma clara de describir un problema en particular de optimización combinatoria es definiendo sus partes:

- El espacio de soluciones X.
- El predicado de factibilidad P.
- El conjunto de soluciones factibles Y.
- f la función objetivo que queremos maximizar o minimizar.

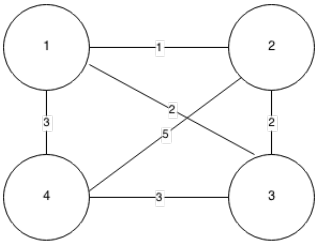
Comparemos las diferentes partes de los dos problemas:

Parte	Biohazard	k-PMP
X	Tomando a los productos como elementos de un conjunto C el espacio de soluciones consta de todas las particiones del conjunto C.	El espacio de soluciones cuenta con todas las particiones de V en un número de k o menos conjuntos ya que puede haber particiones de nodos vacías.
P	La suma de las peligrosidades de cada una de las particiones no supera el umbral de peligrosidad de la instancia.	En este caso el predicado factibilidad es siempre verdadero ya que cualquier partición en k o menos subconjuntos puede ser la solución.
Y	Todos los elementos de X que cumplan P. Todas las particiones que no tengan un camión con una peligrosidad mayor al umbral permitidio.	En este caso X es igual a Y porque el predicado es siempre verdadero.
f	$f(S) = \#S$ y se la quiere minimizar.	$f(S) = \sum_{e \in E_{intraparticion}} w(e)$ y se la quiere minimizar

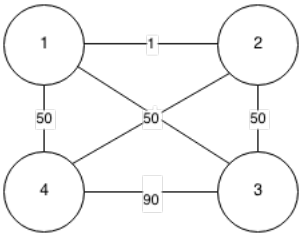
Vemos que en ambos problemas el conjunto de soluciones factibles se corresponde con un subconjunto de las particiones de los nodos en k-PMP y de los camiones en *Biohazard*. Además, en los dos problemas buscamos minimizar la función objetivo.

Si modelamos el problema de los camiones con grafos una posibilidad es tomar a los productos químicos como nodos y asociar la peligrosidad de dos productos como el peso de la arista que los une. De esta forma, al modelar una instancia del problema de los camiones obtendremos un grafo completo con $n = |productos|$ y cada arista con un peso igual a la peligrosidad entre los productos correspondientes a esos dos nodos. Veamos como quedaría una instancia de ejemplo:

	P_1	P_2	P_3	P_4
P_1	0	1	2	3
P_2	1	0	2	5
P_3	2	1	0	3
P_4	3	5	3	0



Si resolvemos k-PMP en el grafo resultante obtendremos la forma de transportar los productos con tan sólo k camiones de manera tal que la peligrosidad total sea mínima. Si resolviésemos k-PMP en el grafo variando k entre 1 y n podríamos quedarnos con la primer k que cumpla que cada una de las particiones no supera el umbral de peligrosidad. Sin embargo, no podemos asegurar que k sea efectivamente la mínima cantidad de camiones. Veamos un ejemplo para ilustrar mejor esto:



Supongamos que el umbral de peligrosidad es $\mu = 50$. Se ve claramente que la partición que devolvería k-PMP para $k = 2$ ó $k = 3$ ($k = 1$ y $k = 4$ son únicas por lo cual k-PMP y *Biohazard* devolverían la mismo) es $S = \{\{1, 2\}, \{3, 4\}\}$. El peso total es $\omega(S) = 1 + 90 = 91$. Por lo tanto, el mínimo k para el cual todas las particiones tienen un peso menor o igual a μ es $k = 4$. Sin embargo, tomando $S' = \{\{1, 4\}, \{3, 2\}\}$ tenemos que $\omega(S) < \omega(S') = 100$ aunque se cumple que $(\forall s'_i \in S' | \omega(s'_i) < \mu)$.

Veamos qué pasa si hacemos el camino inverso modelando k-PMP como *Biohazard*. Utilizando la misma equivalencia que antes obtenemos un producto por nodo y la peligrosidad entre dos productos se obtiene como el peso de la arista que une a los nodos correspondientes a esos productos. Sin embargo, ahora debemos considerar un caso que en primera instancia quedaría afuera del modelo: cuando dos nodos no están unidos por ninguna arista en el grafo. Lo que haremos para que nuestro modelo también incluya a este caso será asignarles una peligrosidad igual a 0 a dos productos cuyos nodos correspondiente no son adyacentes en el grafo. De esta forma podemos asignarles una peligrosidad pero la misma no cambia la instancia del problema ya que no aporta peligrosidad. Por último, resta decidir qué umbral de peligrosidad utilizar. En un primer momento, parece que el problema no tiene sentido

plantearlo como *Biohazard* porque en él buscamos la mínima cantidad de particiones y en k-PMP ese valor ya lo conocemos, es k. Una posibilidad es variar el umbral de peligrosidad de 0 a el peso total del grafo y notar los umbrales en los cuales pasa a necesitar un camión más. Podríamos pensar que si necesita i camiones para que niguno supere una peligrosidad de μ y precisa $i+1$ para que ninguno supere una peligrosidad de $\mu+1$ entonces la partición obtenida al utilizar un umbral igual a $\mu+1$ se corresponde con la partición de $(i+1)$-PMP. No obstante esto es falso. Un coontraejemplo es la (TODO insertar cita a la figura) FIGURA 2 en la que con $\mu = 49$ necesitamos 3 camiones pero para $\mu = 50$ nos basta con 2 camiones pero la partición correspondiente no es 2-PMP como se explicó antes.

1.2. Colores, colores y colores

Se llama coloreo válido de un grafo a una asignación $f : V \to C$ tal que:

$$f(v) \neq f(u) \quad \forall (u, v) \in E$$

donde los elementos de C son llamados colores. Además se denomina k-coloreo de un grafo G a un coloreo válido de un grafo que usa exactamente k colores, es decir $|C| = k$. El problema de coloreo de un grafo consta de encontrar su coloreo mínimo, un coloreo válido que utilice la mínima cantidad de colores. Podemos modelar el problema de coloreo de forma tal que si resolvemos k-PMP resolvemos coloreo de un grafo. Sea $G = (V, E)$ el grafo al cual queremos encontrarle el coloreo mínimo. La única transformación que le haremos al grafo será asignarle a sus aristas peso 1:

$$\omega(v) = 1 \quad \forall v \in V$$

El procedimiento que realizaremos para obtener el coloreo mínimo de G es resolver k-PMP para $k = 1, ..., n$ y quedarnos con el mínimo k_{min} tal que el peso total de la partición es igual a 0:

$$k_{min} = \min_{1 \leq i \leq n} \omega(S_{i-PMP})$$

con S_{i-PMP} igual a la partición correspondiente a i-PMP. Veamos que efectivamente con la partición correspondiente a k_{min}-PMP podemos obtener el coloreo mínimo de G. Tenemos que comprobar dos puntos:

1. La partición representa un coloreo válido.
2. El coloreo representado es mínimo.

1) Si a cada uno de los k_{min} subconjuntos los pintamos de un color distinto tenemos todo el grafo coloreado porque todos los nodos pertenecen a algun subconjunto de la partición. Como el peso total de la partición es igual a 0, $\omega(S_{k_{min}}) = 0$, no puede ser que haya dos nodos adyacentes en la misma partición porque les habíamos asignado a todas las aristas un peso de 1. Si hubiera dos nodos u y v en una misma partición tal que $(u, v) \in E$ como $\omega((u, v)) = 1$ y es una arista intrapartición porque estamos suponiendo que u y v pertenecen a la misma: $\omega(S_{k_{min}}) \geq 1$. Lo cual es absurdo porque sabemos que es igual a 0. Por lo tanto, tenemos un coloreo válido porque les asignamos un color a cada nodo y nodos con un mismo color no son adyacentes.

2) Demostraremos por absurdo que es mínimo. Supongamos que existe un coloreo válido C de G que utiliza $c < k_{min}$ colores. Si tomamos la partición inducida por el coloreo C, es decir tomamos la partición de V que resulta de colocar a los nodos de colores iguales en el mismo subconjunto, obtenemos una partición con $c < k_{min}$ subconjuntos que tiene un peso total de 0 ya que los nodos en cada una de las particiones no son adyacentes entre sí. Esto es absurdo porque partimos asumiendo que k_{min} era la mínima cantidad de subconjuntos necesaria para que el peso total de la partición sea 0.

Así queda demostrado que podemos tomar cualquier grafo G y obtener su coloreo resolviendo k-PMP a lo sumo n veces, ya que no puedo tener una partición con más n subconjuntos, en el grafo obtenido luego de asignarle un peso de 1 a cada una de sus aristas.

Veamos qué sucede si resolvemos coloreo en un grafo para el cual necesitábamos obtener su k-PMP. Resulta que el número cromático de G $\chi(G)$ se corresponde con k_{min}, es decir el mínimo k para el cual el peso total de k-PMP es 0. Esto se puede demostrar con un razonamiento muy parecido al que realizamos para demostrar que el coloreo representado por la partición correspondiente a k_{min}-PMP es mínimio. En primer lugar, construimos la partición correspondiente a algún $\chi(G)$-coloreo ubicando los nodos de cada color en subconjuntos distintos. Como los nodos de un subconjunto s_i son todos del mismo color y pertenecen a un coloreo válido entonces no son adyacentes. Por lo tanto, no existen aristas intrapartición. Lo cual implica que el peso total de la partición obtenida es igual a 0. Desmostramos que es mínimo por absurdo, suponemos:

$$\exists \quad k < \chi(G) \quad | \quad \omega(S_k) = 0$$

pero sabemos que a partir de una partición con peso total igual a 0 podemos obtener un coloreo válido porque las aristas tienen peso estrictamente mayor a 0:

$$\omega(e) > 0 \quad \forall e \in E$$

pero entonces obtendríamos un coloreo válido que utiliza $k < \chi(G)$ colores ya que la cantidad de colores es igual a la cantidad de subconjuntos de la partición. Lo cual es absurdo porque por definición $\chi(G)$ es mínimo. No sólo resolviendo coloreo en un grafo G obtenemos k_{min} y su partición correspondiente, además obtenemos k-PMP para todo $k > k_{min}$ ya que como no podemos obtener un partición con peso estrictamente menor que 0 porque las aristas tiene todas peso positivo necesariamente la partición correspondiente a k_{min}-PMP es mínima para cualquier k mayor. k_{min}-PMP es una partición válida para k-PMP con $k > k_{min}$ porque asumimos que tenemos $(k - k_{min})$ subconjuntos vacíos para completar los k necesarios.

1.3. k-PMP en la vida real

Como vimos antes k-PMP es un problema de optimización combinatoria y cumple con la división de partes definida en el primer punto. Para cada ejemplo definiremos las partes pertinentes:

- *Pabellones de un Hospital*: Se quiere construir un hospital con a lo sumo k pabellones y hay que especializar cada pabellon en áreas de enfermedades lo más compatibles posible. Definimos Las enfermedades como nodos y se analiza un coeficiente de compatibilidad o parentezco entre ellas.

  X: Todos los conjuntos k-particiones (se aceptan particiones vacias) son los pabellones en los cuales divido el hospital.

P e Y: Todas las divisiones posibles de menos de k pabellones forman una solución factible.

f: Que la suma de los coeficientes de parentezco entre enfermedades sea mínima.

- *Union Deportiva Nacional*: En un país hay que crear a lo sumo k distintas uniones/-federaciones de un deporte y quiero que las distancias (aristas) entre las institucione-s/entidades (nodos) que conforman la Unión sea mínima.

  X: Todos los conjuntos k-particiones (se aceptan particiones vacias) son las Uniones en las cuales divido al pais.

  P e Y: Todas las divisiones posibles de menos de k Uniones forman una solución factible.

  f: Que la suma de las distancias entre clubes de las uniones sea mínima.

- *Impresora*: Una impresora tiene que elegir k colores para imprimir y agrupa los colores más parecidos para usar una tinta de color k.

  X: Todos los conjuntos k-particiones (se aceptan particiones vacias) son los colores disponibles en la impresora.

  P e Y: Todas las divisiones posibles de menos de k colores son una solución posible.

  f: Que los colores que se imprimen con el mismo color sean lo mas parecidos posibles.

2. Algoritmo exacto

2.1. Descripción del algoritmo

El algoritmo utilizado para obtener la k-PMP de un grafo G consiste en construir todas las posibles soluciones de manera ordenada para mediante backtracking. Veamos qué contiene el conjunto de soluciones posibles S asociado a un grafo $G = (V, E)$. Sea P_V el conjunto con todas las particiones posibles del conjunto V:

$$S = \{p \quad | \quad p \in P_V \wedge |p| \leq k\}$$

Todas las particiones de un conjunto de n elementos se pueden obtener recursivamente. Si contamos con todas las particiones de i elementos, denominaremos S_i al conjunto que contiene a todas las particiones de i elementos, entonces podemos generar todas las particiones de $i+1$ elementos simplemente tomando cada una de las particiones pertenecientes a S_i y agregando el elemento $i + 1$ a cada uno de sus subconjuntos. Describimos este procedimiento con un pequeño pseudocódigo:

Algorithm 1 Particiones

```
S_{i+1} ← ∅
while S_i ≠ ∅ do
  s ← sacarUno(S_i)
  for j ← 0...|s| do
    nuevoSubconjunto ← dameIesimo(j, s)
    nuevoSubconjunto ∪ e_{i+1}
    S_{i+1} ∪ {s \ dameIesimo(j, s) ∪ nuevoSubconjunto}
  end for
end while
```

Se puede ver fácilmente que con este procedimiento efectivamente consideramos todas las particiones posibles que se pueden obtener con los n vértices de V. Sin embargo, estamos considerando muchas particiones que no es necesario tener en cuenta. Por ejemplo, aquellas que tengan una cantidad de subconjuntos mayor a k.

2.2. Podas y estrategias

Las podas que le agregamos a nuestro backtracking para reducir el árbol de soluciones son:

1. no consideraremos particiones que tengan más de k subconjuntos. Una vez que tenemos k subconjuntos en la solución actual no agregamos más ya que la solución no puede tener más de k subconjuntos. El costo de esta poda es $O(1)$ porque sólo requiere una comparación de tipos primitivos.

2. no considerar particiones que utilicen menos de k subconjuntos. Si estamos buscando la k-PMP de G, llamémosla P_{min} y la misma utiliza menos de k subconjuntos $|P_{min}| < k$ entonces su peso total necesariamente es nulo $\omega(P_{min}) = 0$ ya que si hubiera algún subconjunto s con peso mayor 0 eso indicaría que existe alguna arista $e = (v, u)$ intra-partición en ese subconjunto con peso mayor a 0. Por lo tanto, si tomamos a v uno de los extremos de esa arista y lo pasamos a un subconjunto s' vacío entonces obtendríamos una nueva partición con un peso estrictamente menor al de la mínima lo cual es absurdo. En el algoritmo esto se ve reflejado cuando la cantidad de subconjuntos no utilizados es igual a la cantidad de nodos que nos quedan por ubicar $|P_{actual}| - k = |nodosRestantes|$. En ese caso lo que hacemos es agregar los $|nodosRestantes|$ a cada uno de los subconjuntos todavía no utilizados ya que cualquier partición P' que agregue alguno de los $nodosRestantes$ a uno de los subconjuntos ya utilizados necesariamente cumple que $\omega(P') \geq \omega(P_{actual})$ porque P_{actual} no va a tener más aristas intra-partición y agregando nodos a conjuntos ya utilizados puede que agreguemos aristas intra-partición o no. Esta poda también tiene costo $O(1)$ porque sólo necesita que se comparen la cantidad de subconjuntos vacíos con la cantidad de nodos restantes. Ambos valores son conocidos en todo momento del algoritmo.

3. inicializamos $pesoMinimo = +\infty$ y cada vez que encontramos una nueva partición que incluye a todos los nodos comparamos su peso con el de $pesoMinimo$ si su peso es menor entonces actualizamos $pesoMinimo$ y la partición mínima encontrada hasta el momento. De esta forma siempre sabemos cuál es el peso de la mejor solución obtenida hasta el momento. Entonces, cada vez que vamos a agregar un nodo a un subconjunto comparamos cuál sería el peso total de la nueva partición generada con el de la mínima actual. Si resulta ser mayor o igual no hacemos la llamada recursiva porque sabemos que agregar nodos a la partición sólo puede incrementar el peso de total de la partición porque todas las aristas tienen pesos positivos. Esta poda tiene costo $O(1)$ ya que también implica la comparación de dos valores ya conocidos: el peso total de la partición actual y el peso de la mínima obtenida hasta el momento.

4. si ya existen elementos en los k subconjuntos, es decir no restan subconjuntos vacíos, calculamos el peso que aportaría a la partición agregar cada uno de los nodos restantes al subconjunto que menos peso agregue. Es decir, calculo el mínimo peso que pueden llegar a sumar a la partición actual los i nodos restantes. Si la suma de todos los pesos más el peso actual es mayor al mínimo obtenido hasta el momento dejo de recorrer esa

rama. El costo de esta poda es $O(n^2)$ porque por cada nodo restante $(n-i)$ tengo que recorrer i. El máximo posible es $\frac{n^2}{4}$ que es del orden de n^2.

2.3. Complejidad temporal

Para analizar la complejidad temporal del algoritmo veremos por un lado el costo que pagamos en cada llamada recursiva y por otro la cantidad de llamadas recursivas que realizamos. La función auxiliar utilizada en cada llamada recursiva es $calcularPesoEnSubconjunto$. Esta función es llamada una vez por cada subconjunto, por lo tanto, cuando estamos agregando el nodo v_i tiene una complejidad amortizada de $O(i)$ ya que la cantidad total de nodos en todos los subconjuntos es i y no recorremos subconjuntos vacíos. La cantidad de llamadas recursivas es igual a la cantidad de particiones de i elementos en j subconjuntos con $1 \leq i \leq n$ y $1 \leq j \leq k$. Ya que como vimos antes generamos todas las particiones de los n nodos en k subconjuntos diferentes de forma recursiva utilizando en el paso i las particiones de $i-1$ elementos y agregamos el elemento i en cada uno de los subconjuntos de cada una de las particiones. Este objeto combinatorio, cantidad de formas de ubicar n objetos en k subconjuntos sin dejar subconjuntos vacíos, está representado por los números de Stirling de segunda especie:

$$S(n,k) = \left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{j=0}^{k} (-1)^j \binom{k}{j} (k-j)^n$$

aquí presentamos su fórmula exacta la cual se vale del principio de inclusión-exclusión para resolver el conteo. Una cota superior conocida para la función $S(n,k)$ es:

$$S(n,k) \leq \frac{1}{2} \binom{n}{k} k^{n-k}$$

Sin embargo, este valor sólo representa las hojas de nuestro árbol de backtracking ya que al nosotros generarlas recursivamente primero generamos $S(i,j)$ con $1 \leq i \leq n$ y $1 \leq j \leq k$. Por lo tanto las llamadas recursivas totales son:

$$\sum_{i=0}^{n} \sum_{j=0}^{k} S(i,j) \leq \sum_{i=0}^{n} \sum_{j=0}^{k} \frac{1}{2} \binom{i}{j} j^{i-j}$$

La cota de complejidad resulta ser:

$$O(\sum_{i=0}^{n} \sum_{j=0}^{k} \frac{1}{2} \binom{i}{j} j^{i-j} i)$$

porque le agregamos el costo de calcular el peso en cada uno de los subconjuntos. Ésta cota se corresponde con la complejidad del algoritmo utilizando las podas 1 y 2 porque no consideramos subconjuntos vacíos y tampoco consideramos más de k subconjuntos. La complejidad temporal del algoritmo sin podas es equivalente a la de *Biohazard* ya que consideramos todas las particiones de un conjunto de n elementos, en este caso los nodos y recordamos que era:

$$O(\sum_{i=0}^{n} i!)$$

2.4. Experimentación

El objetivo de esta experimentación es revisar las cotas de complejidad obtenidas a través del análisis teórico a la luz de resultados empíricos y comparar las diferentes podas para ver cuáles son más eficientes y si resulta beneficioso componerlas. Por composición de podas nos referimos a aplicar más de una, lo cual no necesariamente es mejor que aplicar sólo una ya que quizás realizan podas parecidas duplicando el trabajo para podar lo mismo. En primer lugar, ejecutamos el algoritmo sin podas para instancias de hasta 11 nodos, ya que con más nodos tomaba demasiado tiempo, y dividimos el tiempo por $\sum_{i=0}^{n} i!$ que era la cota que habíamos obtenido para el algoritmo sin podas. Esperamos que la curva se asemeje a una recta constante. Cada instancia la corrimos 10 veces y nos quedamos con el tiempo mínimo de las 10 corridas. Para cada $1 \leq n \leq 12$ corrimos 100 instancias generadas pseudoaleatoriamente y nos quedamos con el promedio de las 100:

n	sin podas	k subconjuntos	mínimo peso	hay mejora	compuestas
3	86	74	70	74	64
4	199	148	125	130	118
5	525	276	204	203	183
6	1760	557	372	359	324
7	6906	1096	651	561	516
8	30305	2177	1175	936	886
9	149877	21016	3240	5034	2133
10	819763	62888	7192	10064	4026
11	4561726	160775	13322	17324	6527

Donde:

sin podas backtracking sin podas.

k **subconjuntos** aplicamos simultáneamente las podas de no seguir agregando nuevos subconjuntos si ya tenemos k y no generar particiones con menos de k subconjuntos.

hay mejora se refiere a la poda en la que evaluamos si colocando a todos los nodos que restan en sus respectivos subconjuntos en los que aportan el mínimo peso posible se mejoraría el mínimo peso obtenido hasta el momento.

compuestas se efectuaron las 4 podas simultáneamente.

En esta primer tabla presentamos los tiempos obtenidos en microsegundos (μs). Decidimos presentar estos resultados en forma de tabla y no como un gráfico ya que al sólo poder comparar con pocos valores de n, por las restricciones de tiempo impuestas por la variante del backtracking *sin podas*, los gráficos reflejan de forma pobre las relaciones entre las diferentes podas. Ordenándolas por su performance en estas instancias obtenemos:

$$compuestas < minimo\ peso < hay\ mejora << k\ subconjuntos << sin\ podas$$

Los mayores saltos los encontramos entre *sin podas* y el resto de las podas y entre la poda k *subconjuntos* y el resto. Una posible explicación al segundo salto considerado es que

la restricción de sólo contar con k subconjutos queda implícita en el resto de las podas ya que, como vimos en la sección en la que analizamos cada una por separado, necesariamente una solución en la mayoría de los casos cuenta con exactamente k subconjuntos y no con menos. Además, podemos ver que la composición de podas obtuvo la mejor performance, este es un buen indicio de que las podas mejoran su performance si se ejecutan en simultáneo. Sin embargo, es necesario realizar comparaciones de a pares para ver si todas realmente mejoran entre sí. Por ejemplo, tomamos *mínimo peso* y *hay mejora*, las corremos en simultáneo y por separado. Luego, comparamos los tiempos para ver si efectivamente se mejoran entre sí. Este procedimiento se debería realizar para todos los pares posibles para extraer de alguna forma las podas que de alguna forma podan de manera más disjunta posible, es decir que las ramas que podan prácticamente no se superponen.

n	sin podas/ $\sum_{i=0}^{n} i!$	k subconjuntos (%)	compuestas (%)
3	9	86	74
4	8	74	59
5	3	52	34
6	2	31	18
7	1.16	15	7
8	0.6	7	3
9	0.3	14	1
10	0.2	7	0.5
11	0.1	3	0.1

En esta segunda tabla, en la primer columna tenemos los tiempos de *sin podas* divididos por la cota de complejidad obtenida. Vemos que los valores decrecen pero de forma muy lenta. Con tan pocos valores no estamos seguros si esto se debe a que la cota de complejidad es demasiado holgada o simplemente es un comportamiento que sólo se observa en estos primeros valores de n y luego se mantiene constante. Para obtener más información es necesario correr el algoritmo para más valores de n. En la segunda y tercer columna tenemos el porcentaje del tiempo entre el tomado por una cota con respecto al sin podas:

$$\frac{t(con\ poda) \times 100}{t(sin\ podas)}$$

el objetivo de presentar esta información es doble. Por un lado, queríamos ver la gran mejora que puede representar aplicar aunque sea sólo una cota con respecto a correr el algoritmo sin ellas y por otro ver más claramente como diferentes podas tiene mejor performance que otras. En este caso en particular elegimos comparar *compuestas* porque nos parece que lo más interesante es generar más podas e ir comparandolas con las ya obtenidas de a pares como ya explicamos. Si la comparación es favorable querrá decir que agregarla a nuestro set de podas redundará en una mejora de la performance y por lo tanto habremos enriquecido nuestra poda compuesta.

3. Heurística constructiva golosa

Definimos el peso de un nodo como la suma de los pesos de las aristas que inciden sobre él. La idea de la heurística golosa constructiva es entonces “aislar” a los nodos más pesados, bajo la premisa de que éstos son los más problemáticos y deben ser los primeros en procesarse.

Lo que se hace en un principio es agarrar el nodo más pesado, y agregarlo a un conjunto. Después, se busca el nodo más pesado de los no asignados, y se pone en el mejor conjunto disponible. El criterio para elegir a un conjunto es el peso del nodo en ese conjunto. Y así siguiendo hasta que no haya mas nodos. El pseudocódigo es:

Algorithm 2 HeuristicaGolosaConstructiva(Grafo G, nat k)

```
1: Vector<Conjunto<Nat>> conjuntos(k, vacío)
2: if k ≥ n then
3:    CompletarConSolucionTrivial(conjuntos, n) // conjuntos[i] = i ∀ nodo i ∈ G
4: else
5:    Vector<Nat> nodosOrdenados ← OrdenarNodosPorPesoMayorAMenor(G)
6:    for each vértice de nodosOrdenados do
7:       pesoMinimo ← +∞
8:       mejorConjunto ← 0
9:       for i = 0 to k - 1 do
10:         pesoEnConjunto ← PesoVerticeEnConjunto(vértice, conjuntos[i])
11:         if pesoEnConjunto < pesoMinimo then
12:            pesoMinimo ← pesoEnConjunto
13:            mejorConjunto ← i
14:         end if
15:      end for
16:      conjuntos[mejorConjunto].insertar(vértice)
17:   end for
18: end if
19: return conjuntos
```

Por ejemplo, en el grafo de la Figura 10 (para cada nodo se aclara entre paréntesis su peso) para k = 2 –es decir, la solución son los conjuntos $\{S_1, S_2\}$ siendo ambos vacíos al inicio–. El algoritmo hace lo siguiente: Supongamos que el orden de los nodos por peso dió $[F, C, A, E, B, D]$.Toma el nodo F porque es el más pesado (tiene peso 11) y lo agrega en S_1. Después toma C que tiene peso 9, y lo pone en S_2, que sigue vacío (el peso de C en $S_1 = \{\text{F}\}$ es 3 porque C y F son adyacentes con una arista de peso 3). Hasta ahora tenemos la partición $\{\{F\}, \{C\}\}$. Los nodos A y E tienen ambos peso 8, por el orden ya predefinido toma el vértice A y lo pone en el mejor conjunto, que es S_2 –ahí tiene peso 2– mientras que en S_1 tiene peso 3. Tenemos entonces $\{\{F\}, \{C, A\}\}$. El nodo E tiene peso 5 en S_1 por su arista con F y peso 2 en S_2 a pesar de ser adyacentes a ambos nodos del conjunto, por lo tanto se pone ahí. Siguiendo el mismo razonamiento, el algoritmo pone a B en S_1 y luego a D en S_1. Finalmente, queda lo que muestra la Figura 2. Notar como los nodos más pesados tienden a estar más aislados, en particular el nodo F lo está completamente. Notar también que las aristas más pesadas (de peso ≥ 3) no forman parte de ningún conjunto de la partición calculada por la heurística.

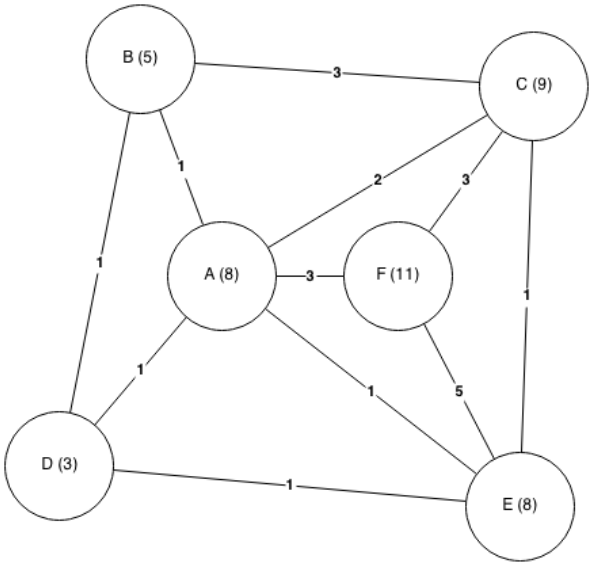


Figura 1: Grafo de entrada

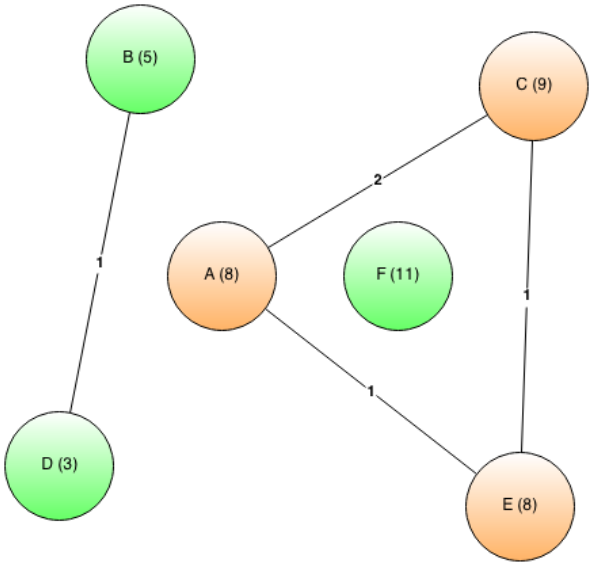


Figura 2: La solución dada por la heurística, diferenciada con colores

El problema de la heurística es que solamente compara el peso de un nodo con los conjuntos, sin tener en cuenta el resto de los nodos aún no asignados. Esto puede dar soluciones tan alejadas de la óptima como queramos. A saber:

Sea G un grafo cuyas aristas tienen peso 1, y k el parámetro de k-PMP. Es decir, el grado de un vértice es igual a su peso. G tiene dos tipos diferentes de nodos:

- Los nodos $P_1, ..., P_k$ forman un subgrafo completo. Entonces, $d(P_i) = k - 1$ en ese subgrafo.
- Los nodos $L_1, ..., L_N$ verifican que cada L_i es solamente adyacente a todos los nodos P_i. Es decir, $d(L_i) = k$ para todo $i = 1, ..., N$.

Por lo anterior, los nodos P_i tienen grado $N + k - 1$. Estos nodos van a ser más pesados que los L_i cuando $N > 1$, pues

$$N + k - 1 > k \iff N > 1$$

Supongamos entonces que hay más de un nodo de tipo L. Notemos además que el conjunto $\{L_1, ..., L_N\}$ tiene peso cero, ya que ningún par de nodos es adyacente entre sí. Como el algoritmo ordena por peso de mayor a menor, el orden será del tipo

$$[P_{i_1}, ..., P_{i_k}, L_{j_1}, ..., L_{j_N}]$$

Entonces, si N > 1, el algoritmo va a elegir un nodo P hasta que se terminen. A P_{i_1} lo va a poner en S_1. A P_{i_2} no lo puede poner en S_1 porque tiene peso 1 ahí (es adyacente a P_{i_1}, por lo tanto lo pone en S_2 que está vacío. A P_{i_3} no puede ponerlo ni en S_1 ni en S_2 por la misma razón, entonces lo pone S_3. Así, una vez terminados todos los nodos P, la partición es

$$\{\{P_{i_1}\}, \{P_{i_2}\}, ..., \{P_{i_k}\}\}$$

Queda insertar los N nodos restantes, que son los $L_{j_1}, ..., L_{j_N}$. Pero para cualquiera de estos nodos, su peso en S_x es 1 porque es adyacente a P_{i_x}. Como el algoritmo elige al primer

conjunto y después chequea si eligiendo los demás mejora, y no lo hace en este caso, todos van a ir a S_1. Por lo tanto, la solución dada por el algoritmo será

$$\{\{P_{i_1}, L_1, L_2, ..., L_N\}, \{P_{i_2}\}, ..., \{P_{i_k}\}\}$$

El peso total de esta solución es N. Pero si la solución fuera

$$\{\{L_1, L_2, ..., L_N\}, \{P_{i_1}, P_{i_2}\}, ..., \{P_{i_k}\}\}$$

el peso total sería 1. Es decir, la solución dada por la heurística golosa puede ser tan mala como queramos. Veamos gráficamente para k = 3 la diferencia:

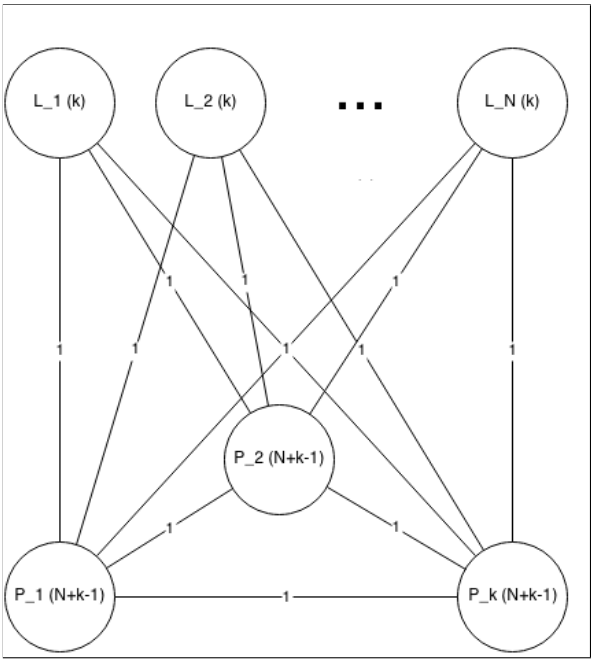


Figura 3: Grafo de entrada

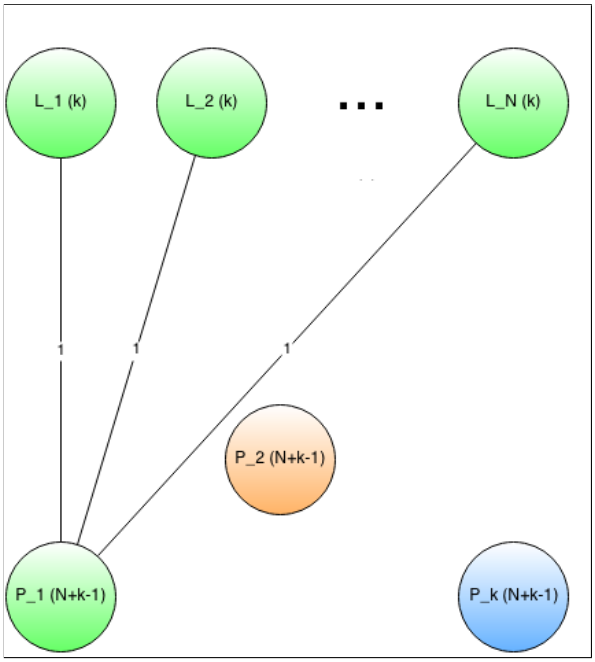


Figura 4: Solución golosa, con peso N.

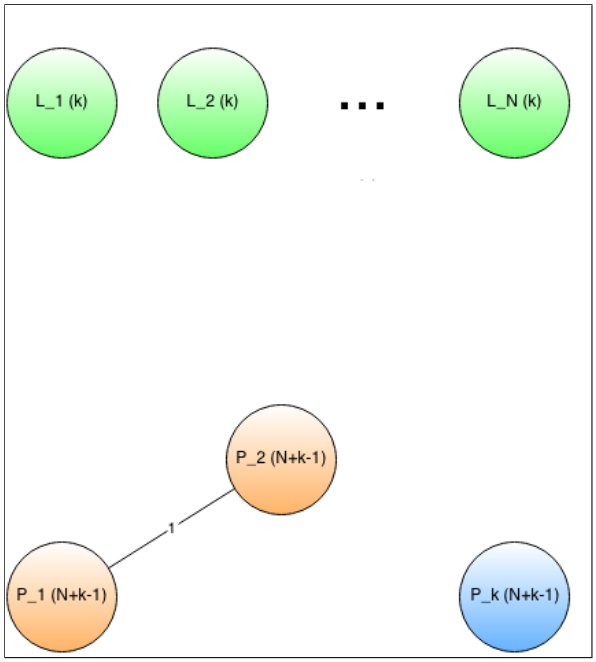


Figura 5: Una solución óptima, con peso 1.

3.1. Análisis de complejidad temporal

Sea n la cantidad de vértices del grafo de entrada, y k el parámetro de k-PMP. Vamos a usar el pseudocódigo ya introducido para el cálculo de complejidad:

1. En la primera línea se crean k conjuntos vacíos. Esto cuesta $O(k)$. Llamaremos C_i al i-ésimo conjunto. El caso poco interesante es que sea $k \geq n$. Se toma como solución (que de hecho es óptima) la partición

   $$\{\{v_1\}, \{v_2\}, ..., \{v_n\}, \{\}, ..., \{\}\}$$

   que tiene peso total cero. Insertar los nodos cuesta $O(n)$, y ahí terminaría el algoritmo, cuya complejidad temporal total sería $O(k)$. Pasemos al caso relevante, que es $k < n$.

2. Se crea un vector de tamaño n, con los vértices ordenados por peso en el grafo, de mayor a menor. Preguntarle a un vértice su peso en el grafo cuesta $O(n)$ porque representamos al grafo con una matriz de adyacencia, entonces en total es $O(n^2)$ saber el peso de todos los nodos. Finalmente, ordenar las tuplas $< nodo, peso(nodo) >$ cuesta $O(n \log n)$. Por lo tanto, todo esto cuesta $O(n^2)$. Al nodo i-ésimo de este nuevo ordenamiento lo llamaremos u_i.

3. Ahora para cada u_i se hace lo siguiente:

   a) Inicializar las variables $pesoMinimo$ y $mejorConjunto$ cuesta $O(1)$.

   b) Para cada conjunto C_i de la partición, se calcula el peso de u_i en él y se chequea si es mejor para cambiar el $mejorConjunto$. El peso en un C_i se calcula simplemente sumando los pesos de la aristas adyacentes entre u_i y los vértices del conjunto, lo cual lleva $O(|C_i|)$. Sabemos que

   $$\sum_{i=1}^{k} |C_i| = i - 1$$

   (pues sólo se asignaron $i-1$ nodos en la iteración i-ésima). Por otro lado, estamos iterando sobre k conjuntos, entonces el costo de este paso es $O(k) + O(i-1)$.

   c) Insertar el nodo en $mejorConjunto$ cuesta $O(\log |mejorConjunto|)$ que podemos acotar por $O(\log(i-1))$.

   En total, todo este paso cuesta

   $$\sum_{i=1}^{n} (O(k) + O(i-1) + O(\log(i-1))) = O(nk) + O(n^2) + O(\log n!)$$

   Por la aproximación de Stirling del factorial ($\log n! = n \log(n) - n + O(\log n)$), tenemos que $O(\log n!) = O(n \log n)$, entonces la complejidad total de este paso es $O(nk) + O(n^2)$.

Tenemos entonces que la complejidad temporal de la heurística golosa es

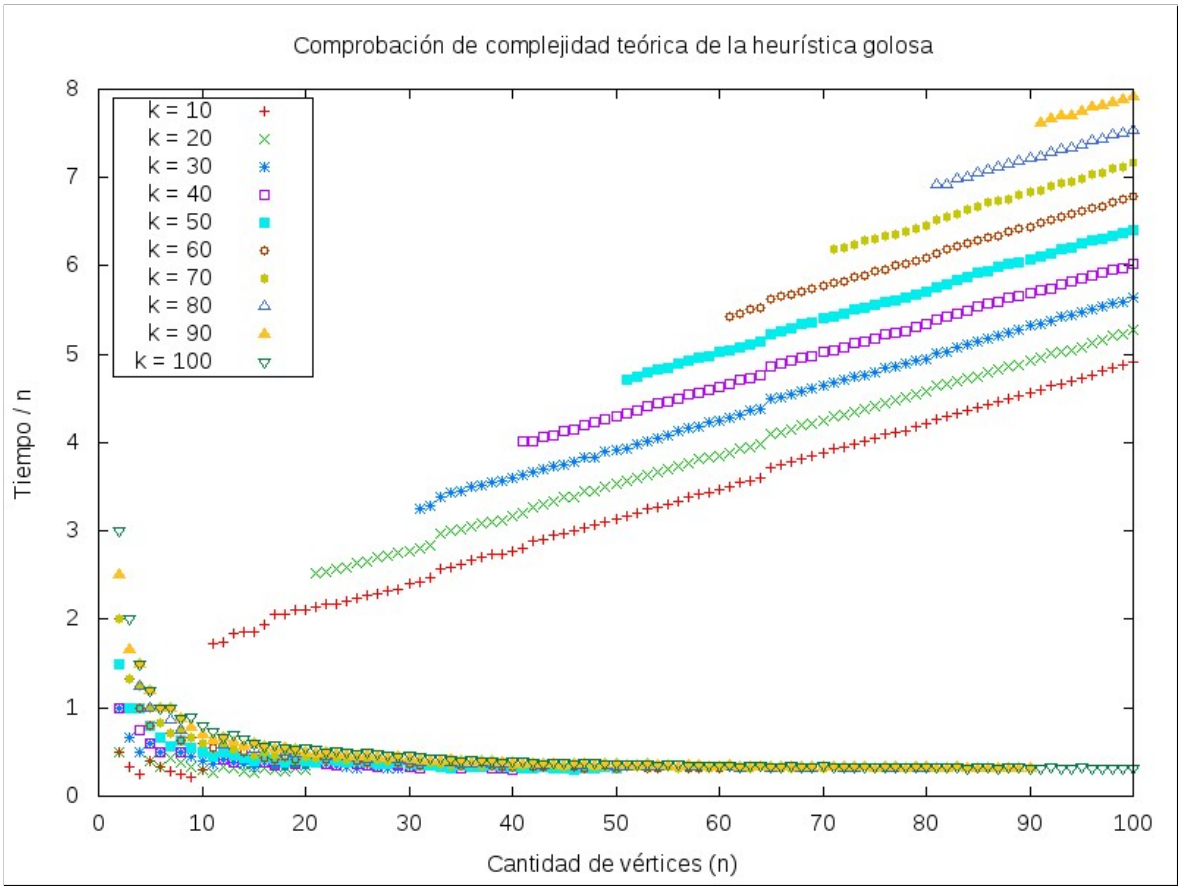
$$T(n,k) = O(k) + O(n^2) + O(nk) + O(n^2)$$
$$T(n,k) = O(nk) + O(n^2)$$

Como tomar valores $k \geq n$ no tiene sentido, ya que una solución óptima es trivial, entonces siendo $k < n$ vale que $O(nk) = O(n^2)$. Por lo tanto, la complejidad temporal de peor caso de la heurística para un tamaño de entrada de n vértices es

$$T(n) = O(n^2)$$

3.2. Test de complejidad

Construimos 100 instancias aleatorias de grafos de n vértices, para cada $n = 1, ..., 100$. Para cada instancia, se calculó el tiempo de ejecución de la heurística con parámetro $k = 10, 20, ..., 100$ cinco veces (tomando el mínimo), para luego calcular el promedio para cada n separando por k. Así, para cada n tenemos el tiempo de ejecución promedio de cada k. Como esperamos que la complejidad temporal sea $O(n^2)$ se dividió la muestra por n, esperando ver una recta, que es lo que efectivamente ocurre. Veamos el gráfico de los resultados del test:



Podemos observar que para cada $n \leq k$, el tiempo dividido por n es una constante, lo cual tiene sentido porque habíamos acotado por $O(k)$ construir la solución trivial, y $O(n) \subseteq O(k)$ porque $n \leq k$. Recién a partir de $n = k + 1$ el tiempo dividido n se vuelve apreciable, y como esperábamos es una recta, lo que indica que es $O(n^2)$ como era nuestra hipótesis.

4. Heurística de búsqueda local

4.1. Descripción de los algoritmos

El procedimiento de una heurística de busqueda local consiste en tomar una solución inicial s cualquiera e iterativamente mejorarla remplazandola por una solución mejor del conjunto de soluciones vecinas ($N(s)$ definidas particularmente en cada algoritmo), hasta llegar a un óptimo local.

Llamamos heurística de busqueda local al método de selección que usamos para elegir o descartar una vecindad. Los algoritmos implementados a continuación usan el mismo criterio de heurística que consiste en iterar sobre las vecindades y elegir la primera que mejora la solución. Usamos este criterio por que mejora el tiempo de ejecución ya que cuando encuentra una mejor la elige y por que no encontramos otra considerablemente mejor. Por ejemplo se podría haber propuesto una heurística que recorra todas las vecindades y elija la mejor opción entre todas, pero aun en el mejor caso deberiamos recorrer todas las soluciones de $N(s)$ y esto incrementa el costo temporal.

Implementamos dos tipos de vecindades: **(A)** toma un nodo de alguno de los k subconjuntos y prueba si cambiando este nodo de subconjunto mejora el valor de la solución, siendo la vecindad todas las soluciones que difieren de la actual con un solo nodo en otro subconjunto k. **(B)** consiste en intercambiar dos nodos cualesquiera de distintos subconjuntos y ver si el valor mejora, es decir que la vecindad son aquellas soluciones que difieren solamente con un intercambio de nodos en dos distintos subconjuntos.

Consideramos que la implementación que tiene la vecindad de tipo **(B)** es más limitada que la **(A)** porque la **(B)** mantiene la cardinalidad en los subconjuntos de la solución (ya que solo hace swap entre nodos de distintos subconjuntos) y esto hace que el algoritmo este más condicionado por la solución de partida.

(A):

Algorithm 3 HeuristicaBusquedaLocal(Grafo G, nat k)

```
1:  Vector<Conjunto<Nat>> conjuntos(k, vacío)
2:  solucionInicial(G,k)
3:  Bool hayMejora ← true
4:  while hayMejora do
5:     hayMejora ← false
6:     for each nodos do
7:        Bool swappeado ← false
8:        Int subset ← 0
9:        while !swappeado y subset < k do
10:          if subset != subcjtoDel(nodo) y pesoDelNodoEnSubcjto(actual) > pesoDelNodoEnSubcjto(subset) then
11:             borroNodoDeSubcjto(nodo,actual)
12:             agregoNodoASubcjto(nodo,subset)
13:             hayMejora ← true
14:             swappeado ← true
15:          end if
16:          subset++
17:       end while
18:    end for
19: end while
20: return conjuntos
```

(B):

Algorithm 4 HeuristicaBusquedaLocalConSwap(Grafo G, nat k)

```
1: Vector<Conjunto<Nat>> conjuntos(k, vacío)
2: solucionInicial(G,k)
3: Bool hayMejora ← true
4: while hayMejora do
5:    hayMejora ← false
6:    for each nodos do
7:       Bool swappeado ← false
8:       nodoSwap ← nodo+1
9:       while !swappeado y nodoSwap < n do
10:         if subcjtoDel(nodo) != subcjtoDel(nodoSwap) then
11:            pesoNodoCambiado ← pesoDelNodoEnSubcjto(nodo,subcjtoDel(nodoSwap))
12:            pesoNodoSwapCambiado ← pesoDelNodoEnSubcjto(nodoSwap,subcjtoDel(nodo))
13:            if peso(nodo)+peso(nodoSwap) > pesoNodoCambiado+pesoNodoSwapCambiado
               then
14:               borroNodoDeSubcjto(nodo,subcjtoDel(nodo))
15:               agregoNodoASubcjto(nodo,subcjtoDel(nodoSwap))
16:               borroNodoDeSubcjto(nodoSwap,subcjtoDel(nodoSwap))
17:               agregoNodoASubcjto(nodoSwap,subcjtoDel(nodo))
18:               hayMejora ← true
19:               swappeado ← true
20:            end if
21:         end if
22:         nodoSwap++
23:      end while
24:   end for
25: end while
26: return conjuntos
```

Ejemplo de ejecución de los algoritmos:

Tenemos el siguiente grafo de entrada, y nos piden correr los algoritmos para $k = 3$

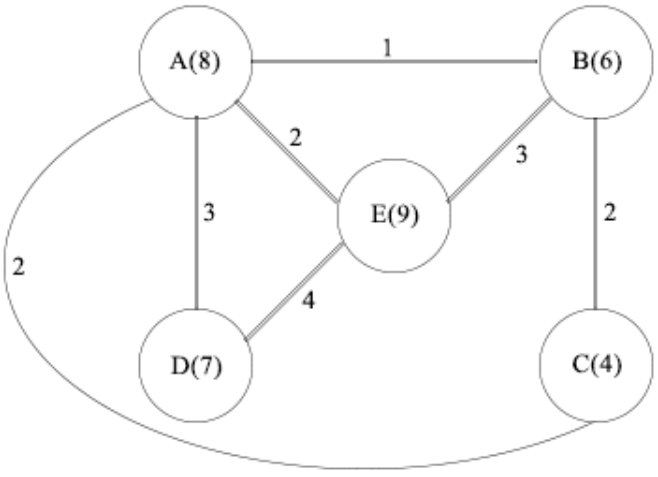


Figura 6: Grafo de entrada

Contamos con la siguiente solución inicial (entre parentesis el peso de cada nodo):

S_1	S_2	S_3
A(3)	E(0)	D(0)
B(4)		
C(3)		

Costo total: 5

(A)

Tomo A:
→ Lo pruebo en S_2: A(2) ✓Mejora

S_1	S_2	S_3
B(2)	E(2)	D(0)
C(2)	A(2)	

Costo total: 4

Tomo B:
→ Lo pruebo en S_2: B(4) No mejora
→ Lo pruebo en S_3: B(0) ✓Mejora

S_1	S_2	S_3
C(0)	E(2)	D(0)
	A(2)	B(0)

Costo total: 2

Tomo C:
→ Lo pruebo en todos pero no mejora en ninguno por que tiene peso 0.
Tomo D:
→ Lo pruebo en todos pero no mejora en ninguno por que tiene peso 0.
Tomo E:
→ Lo pruebo en S_1: E(0) ✓Mejora

S_1	S_2	S_3
C(0)	A(0)	D(0)
E(0)		B(0)

Costo total: 0

Hace una iteración mas buscando mejora pero no la encuentra, entonces termina.

(B)

Tomo A y B:
→ estan en el mismo subconjunto (no hago nada).
Tomo A y C:
→ estan en el mismo subconjunto.
Tomo A y D:
→ como peso(A en S_3 sin D)+peso(D en S_1 sin A) < peso(A actual)+peso(D actual)

S_1	S_2	S_3
B(2)	E(0)	A(0)
C(2)		
D(0)		

Costo total: 2

Tomo B y A:
→ no mejora.
Tomo B y C:
→ estan en el mismo subconjunto.
Tomo B y D:
→ estan en el mismo subconjunto.
Tomo B y E:
→ no mejora.
Tomo C y A:
→ no mejora.
Tomo C y B:
→ estan en el mismo subconjunto.
Tomo C y D:
→ estan en el mismo subconjunto.

...

No mejora en ninguna de las combinaciones siguientes. Termina.

4.2. Análisis de complejidad temporal

Sea n la cantidad de vértices del grafo de entrada, y k el parámetro de k-PMP. Vamos a usar el pseudocódigo ya introducido para el cálculo de complejidad de ambos algoritmos. Hay que mencionar que ésta se calcula para el peor caso de una iteración del ciclo más exterior de los algoritmos (*while(hayMejora)*) ya que no podemos determinar cuantas veces se itera sobre este mismo. Lo que si se afirma es que termina cuando encuentra un óptimo local, es decir ninguno de sus vecinos tiene una mejor solución. Se omite la complejidad de la obtención de la solución inicial y se asume como entrada.

(A):

- Linea 5: seteamos la variable *hayMejora* en $O(1)$
- Linea 6: iniciamos un ciclo que se repite n veces.
  - Lineas 7 y 8: seteamos variables en $O(1)$ y calculamos el peso del nodo en cuestión en el subconjunto esto cuesta $O(n)$ (si bien no está explícito en el pseudocódigo, realizamos este preproceso para no realizarlo dentro del ciclo siguiente y así aumentar su complejidad innecesariamente.)
  - Linea 9: Se inicia otro ciclo sobre los subconjuntos que como peor caso se prueba cambiar el nodo a todos los subconjuntos y recorremos los k.
    - Linea 10: Checkeo booleano del if en $O(1)$
    - Lineas 11, 12, 13 y 14: borrar e insertar nodos de un conjunto cuesta $O(\log n)$ y las asignaciones de variables $O(1)$.
      Entonces tenemos $2O(\log n) + 2O(1)$
      Cómo peor caso esto ocurre una vez en las k iteraciones ya que hace que salgamos de él.
    - Linea 16: aumentamos una variable en $O(1)$

Entonces tenemos:

$$Complejidad(A) = O(1) + n(O(1) + O(n) + O(k) + 2O(\log n) + 2O(1) + O(1))$$

$$Complejidad(A) = n(max\{O(n), O(k), O(\log n)\})$$

$$Complejidad(A) = max\{O(n^2), O(nk), O(n \log n)\}$$

$$Complejidad(A) = O(n^2)$$

(B):

- Linea 5: seteamos la variable *hayMejora* en $O(1)$
- Linea 6: iniciamos un ciclo que se repite n veces.
  - Lineas 7 y 8: seteamos variables en $O(1)$ y calculamos los pesos de ambos nodos en los dos subconjuntos, esto cuesta $4O(n)$ (esto lo hacemos para no calcularlos dentro del siguiente ciclo.)
  - Linea 9: Se inicia otro ciclo sobre los nodos (para checkear con quien swappeo) en peor caso se repita n veces.
    - Linea 10: Checkea booleano del if en $O(1)$
    - Lineas 11 y 12: Calcula el valor del peso del nodo en un subconjunto, como peor caso esto toma $O(n)$
    - Linea 13: Checkea booleano del if en $O(1)$
    - Lineas 14 a 19: Borra e inserta nodos de un subconjunto a otro, cuesta $O(\log n)$ y las asignaciones de variables $O(1)$.
      Entonces tenemos $4O(\log n) + 2O(1)$
      Cómo peor caso esto ocurre una vez en las n iteraciones ya que hace que salgamos de él.

- Linea 16: aumentamos una variable en $O(1)$

Entonces tenemos:

$$Complejidad(B) = O(1) + n(O(1) + 4O(n) + n(O(n) + 4O(\log n) + 2O(1)) + O(1))$$

$$Complejidad(B) = n(O(n) + O(n^2))$$

$$Complejidad(B) = O(n^3)$$

4.3. Experimentación de calidad

Con esta experimentación nos interesan ver dos cosas:

1. cuál de las dos vecindades obtiene soluciones de mejor calidad.
2. cuál de las dos vecindades *gana* más veces que la otra. Por ganar nos referimos a que obtiene una partición de menor peso.

Para evaluarlo generamos un set de instancias que se compone de 100 instancias por n, siendo n la cantidad de nodos de la instancia, con $3 \leq n \leq 22$. La limitante para aumentar más el n es que el tiempo del exacto, incluso con todas las podas, crece demasiado rápido y para poder saber cuán lejos estaban los pesos obtenidos por las heurísticas de búsqueda local era necesario correr el algoritmo exacto. En todas las instancias las búsquedas para las dos vecindades parten de la misma partición inicial que es la generada por el goloso sin aleatorización. Creemos que empezar desde la solución obtenida por la golosa se acerca un más al uso real que va a tener en comparación con empezar con una partición completamente aleatoria, por lo cual los resultados que obtengamos serán más representativos del comportamiento que vayan a tener en su contexto de uso. Los valores graficados se corresponden con los promedios de pesos por instancia. Es necesario aclarar que limitamos el número de iteraciones de las dos búsquedas locales a 100. Creemos que esto es importante ya que no sólo nos interesa controlar que la calidad de las soluciones que estamos comparando hayan sido obtenidas dentro de el mismo número de iteraciones. De no haber hecho esto una de las dos podría haber superado a la otra pero realizando muchas más iteraciones. Sumamos los pesos obtenidos para las 100 instancias de cada n y luego tomamos el promedio. El promedio no es un buen indicador en muchos casos ya que no nos habla en general de la dispersión de los resultados. Para darnos una idea de cuán estable son los resultados que obtienen cada una de las vecindades necesitaríamos obtener también el desvío estándar. Sin embargo, los promedios de las dos vecindades han quedado con error relativo pequeño con respecto al peso obtenido por el algoritmo exacto por lo cual el desvío estándar debe ser pequeño ya que no pueden haber valores por debajo del exacto para compensar otros muy altos.

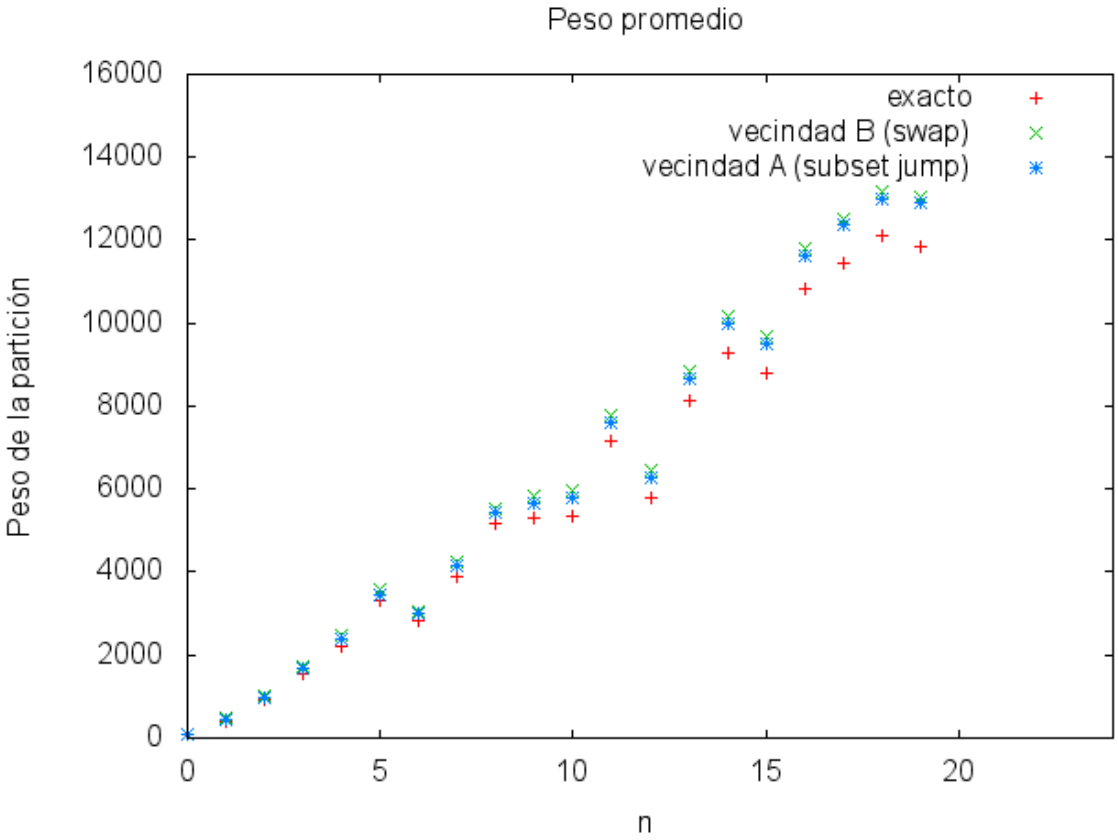


Figura 7: Pesos promedio.

La distancia entre los pesos obtenidos por las vecindades y el exacto se mantiene parece estar creciendo. No obstante, la distancia entre las dos vecindades se mantiene, a simple vista, constante. La vecindad A, que corresponde a la que cambia de subconjunto a un nodo, siempre se ubica por debajo de la vecindad B, correspondiente a la de swap, pero por un margen muy pequeño. Sobre todo para las instancias donde el n es más grande, superior a 15, los resultados dejan de seguir una curva suave lo cual interpretamos que se debe a que 100 instancias a medida que el n aumenta deja de ser un valor representativo. Además, realizamos un histograma que muestra para cada n qué vecindad obtuvo el menor peso de las dos para cada una de las 100 instancias. De esta forma podemos ver más claramente cuál ganó más veces, otro indicador de cuál podría llegar a ser superior.

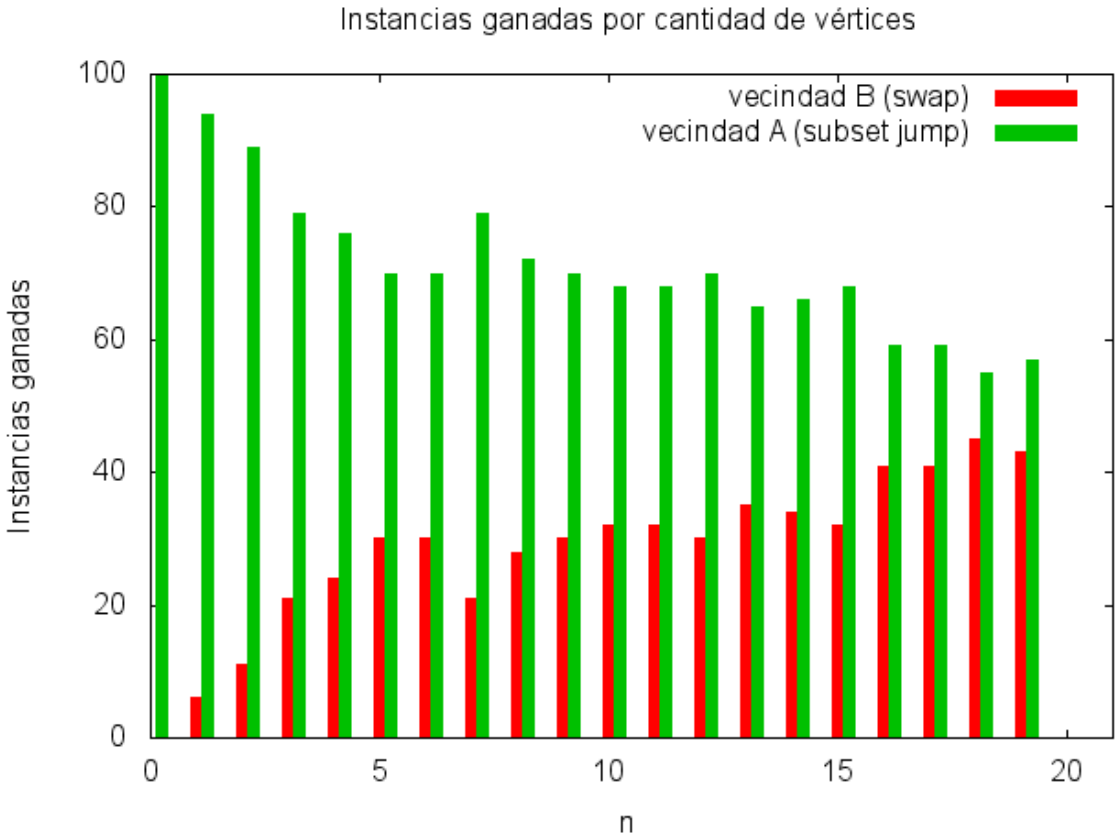


Figura 8: Instancias ganadas.

Vemos que la vecindad A comienza con una ampli ventaja sobre la vecindad B pero a medida que el n aumenta se va emparejando cada vez más. Ésta es una tendendia que no habíamos podido apreciar en el gráfico de pesos. No estamos seguros si se debe a como ya mencionamos la falta de representatividad de las 100 instancias para los n más grandes o a que efectivamente la vecindad A obtiene soluciones de mejor calidad en la mayoría de los casos.

4.4. Experimentación de complejidad

En esta experimentación simplemente queremos ganar confianza en las cotas de complejidad calculadas teóricamente. Recordamos que para la vecindad A (cambio de subconjunto) la cota obtenida había sido $O(n^2)$. Lo que hicimos fue correr el mismo conjunto de instancias que para la experimentación de calidad pero aumentando el rango de valores de que toma n ya que en este caso no es necesario correr el algoritmo exacto que nos limitaba anteriormente. Los tiempos están divididos por n.

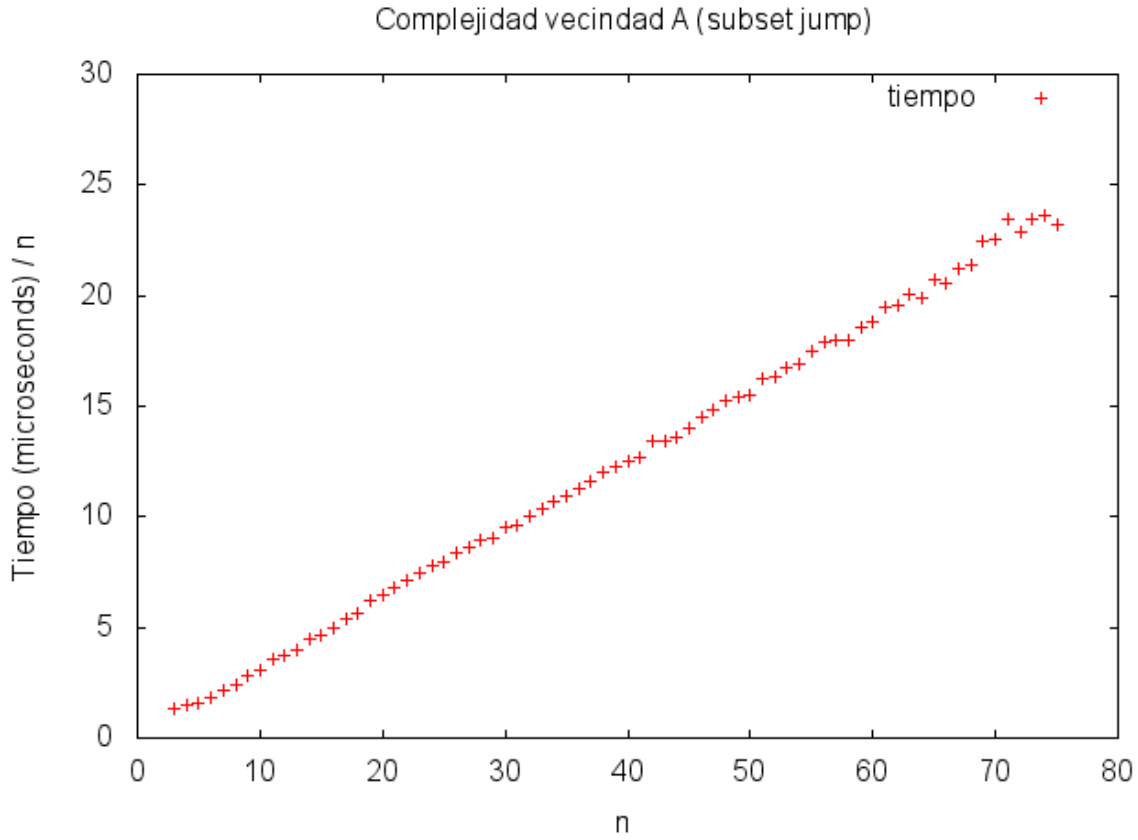


Figura 9: Instancias ganadas.

Los tiempos parecen presentar un comportamiento bastante definido incluso para valores de n más grandes por lo cual creemos que la cota de complejidad se acerca lo suficiente a la real. Realizamos lo mismo para la vecindad B (swap). Recordamos que la cota obtenida para ésta vecindad era $O(n^3)$ por lo cual dividimos los tiempos por n^2.

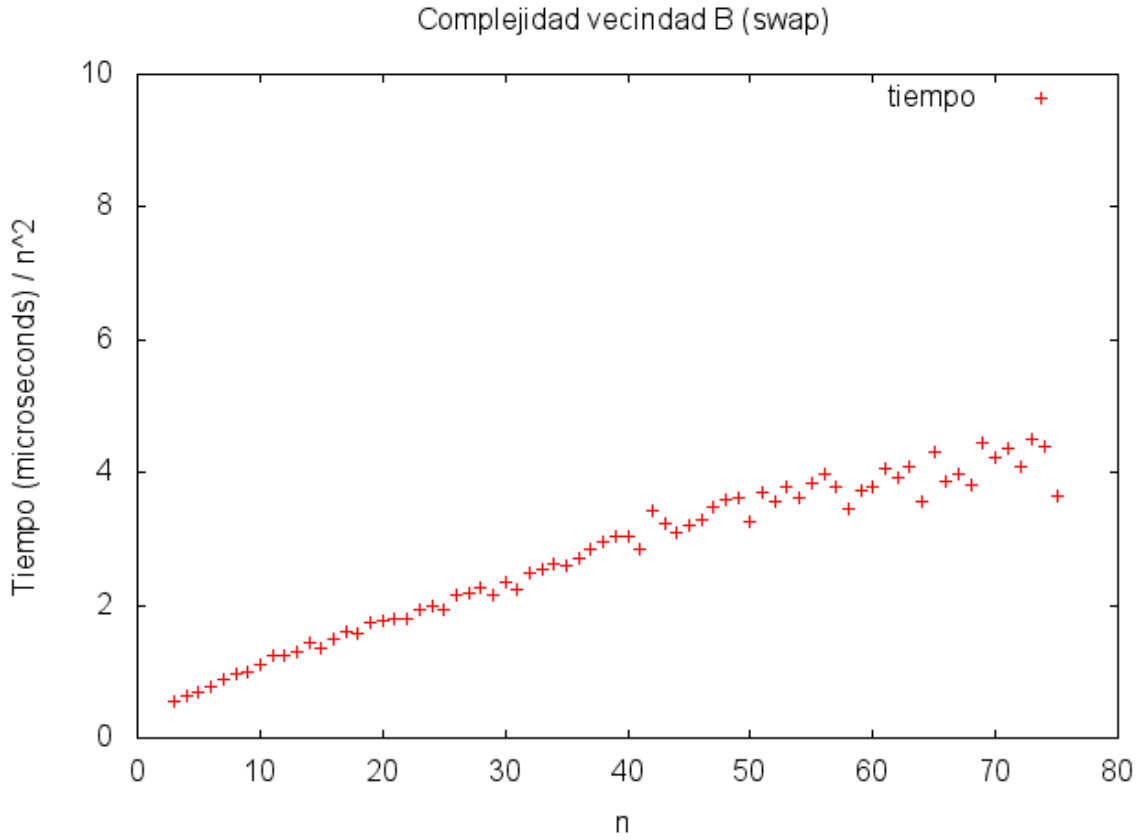


Figura 10: Instancias ganadas.

En este caso vemos que si bien parece que los valores exhiben un crecimiento lineal, a partir de $n = 40$ esta tendencia se vuelve cada vez más endeble ya que los valores empiezan a presentar turbulencia. Creemos que una vez más esto puede deberse a la falta de representatividad de un valor de sólo 100 instancias para un número de nodos tan grandes, lo cual podría resultar en instancias muy densas y otras muy simples. Concluimos que la vecindad A por tener una complejidad menor y mostrarse levemente superior en cuanto a calidad de las soluciones obtenidas con respecto a la vecindad B será una mejor elección para nuestra implementación de la metaheurística GRASP.

5. Metaheurística GRASP

5.1. Descripción del algoritmo

Una heuristica GRASP es un proceso iterativo, en el cual cada iteración consiste en dos fases, una constructiva, en la cual una posible solución es producida y otra fase con una busqueda local en la cual se encuentra un óptimo local en una vecindad definida por la solución construida en la primera fase. La mejor solución obtenida se devuelve como resultado. El siguiente pseudocodigo describe el procedimiento de GRASP.

5.1.1. Pseudocódigo de GRASP

Algorithm 5 GRASP(Grafo G, Nat k, Nat maxIter)

```
1: mejorSolucion ← ∞
2: while criterioDeParada(maxIter) do
3:    solucionGolosa ← HeuristicaGolosaAleatorizada(G,k, etc.)
4:    solucionLocal ← busquedaLocal(G,k,solucionGolosa,etc.)
5:    if solucionLocal < mejorSolucion then
6:       mejorSolucion ← solucionLocal
7:    end if
8: end while
9: return mejorSolucion
```

criterioDeParada es una función que devuelve cuando para el algoritmo. Las utilizamos para evitar que corra infinitamente y para garantizar una solución en tiempo razonable.

Implementamos dos criterios:

1. *Cantidad Máxima de Iteraciones*: el ciclo termina cuando llega a hacer una prefijada cantidad de iteraciones
2. *Iteraciones sin Mejora*: el ciclo termina cuando hace una prefijada cantidad de iteraciones sin mejorar la *mejorSolucion* hasta el momento.

También se puede implementar una combinación de ambas, entonces el ciclo termina cuando ocurre alguna de las dos cosas. (estos criterios son analizados en mayor profundidad en la sección de *Testing*).

GRASP También hace la función *HeuristicaGolosaAleatorizada* para obtener una solución inicial para la posterior optimización de la busqueda local. Esta es una modificación de la Heuristica Golosa presentada anteriormente, con ciertas decisiones aleatorizadas.

Presentamos el pseudocódigo de la misma:

5.1.2. Pseudocódigo de la heurística golosa aleatorizada

Algorithm 6 HeuristicaGolosaAleatorizada(Grafo G, Nat k, Nat profundidadVertice, Nat profundidadConjunto)

```
1: Vector<Conjunto<Nat>> conjuntos(k, vacío)
2: Vector<Nat> nodosOrdenados ← OrdenarNodosPorPesoMayorAMenor(G)
3: Nat cantidadConjuntosCandidatos ← minimo(k, profundidadConjunto)
4: while not nodosOrdenados.vacio() do
5:     Nat cantidadVerticesCandidatos ← minimo(nodosOrdenados.size(), profundidad-
       Vertice)
6:     verticeNuevo ← Elegir de manera aleatoria un nodo de nodosOrdenados entre las
       primeras cantidadVerticesCandidatos posiciones
7:     Borrar a verticeNuevo de nodosOrdenados
8:     Ordenar conjuntos por el peso del verticeNuevo en ellos de menor a mayor
9:     Elegir de manera aleatoria un conjunto de conjuntos entre los primeras
       cantidadConjuntosCandidatos posiciones
10:    Insertar nuevoVertice en conjunto
11: end while
12: return conjuntos
```

Los puntos claves del algoritmo anterior son en la randomización de elecciones:

1. *Profundidad de la elección de nodos*: el algoritmo goloso sin randomización ordena decrecientemente los nodos por peso y luego elige siempre el primero para colocarlo en algún subconjunto, pero aquí determinamos un entero v que va a servir como parámetro para elegir aleatoriamente entre los v primeros nodos.
2. *Profundidad de la elección de conjuntos*: en la versión sin aleatoriedad siempre se elige colocar el nodo en cuestión dentro del subconjunto en el cual tenga peso mínimo, en cambio en este se determina un entero s que sirve como parámentro para elegir aleatoriamente entre los s subconjuntos donde el nodo tenga el menor peso.

Ademas de que uno elige sobre nodos y el otro sobre conjuntos, la diferencia entre los dos puntos anteriores es que el orden del primer punto se determina al principio del algoritmo y no cambia a lo largo de la ejecución, a diferencia del segundo punto que su orden varia en tiempo de ejecución. Estos items son profundizados en la siguiente sección de *testing*.

5.2. Tests

Mostraremos los resultados de varios tests realizados a la GRASP, los cuales introducimos brevemente:

- **Test de configuración**: Dado un conjunto de instancias, busca la mejor configuración de la GRASP, variando criterios de parada y de selección de la lista de candidatos (RCL) de la heurística golosa aleatorizada.
- **Test de calidad**: Para un conjunto reducido de instancias, se compara cuánto más pesada es la solución de la GRASP en relación a la solución óptima, usando la configuración ganadora del test anterior.

- **Test de tiempo de ejecución**: Dado un conjunto de instancias, se calculan los tiempos de ejecución de la GRASP para distintas configuraciones.

El conjunto de instancias utilizado está compuesto por 100 instancias para cada $n = 3, ..., 100$ y tiene las siguientes características:

1. Los pesos de las aristas de los grafos pertenecen al intervalo cerrado $[0.0001, 1000]$.
2. Sea $G = (V, X)$ con $n = |V|$ y $m = |X|$ un grafo cualquiera del conjunto. Sea $m_{max} = \frac{n(n-1)}{2}$. Entonces,

$$0.7 \cdot m_{max} \leq m \leq m_{max}$$

   Esto lo hicimos para evitar tener instancias con grafos fáciles de resolver incluso para la heurística golosa. Esto es claro en el caso extremo de que el grafo no tenga aristas. Teniendo esto en cuenta, pedimos que al menos tengan 70 % de la máxima cantidad de aristas posibles para cada grafo.

3. Sea $G = (V, X)$ con $n = |V|$ un grafo cualquiera del conjunto. Entonces,

$$2 \leq k \leq max\left(2, \frac{n}{3}\right)$$

   La razón de esto es la misma que en el punto anterior. En el caso extremo, si tuviéramos un grafo con $k \geq n$, alcanzaría con poner un vértice en cada conjunto para obtener una solución óptima (en el otro extremo, $k = 1$ sólo admite una solución, que obviamente es óptima). A mayor k, más márgen de error se le da a la heurística para equivocarse porque tiene más conjuntos donde colocar vértices. Por este motivo limitamos a $\frac{n}{3}$ la cantidad de conjuntos que puede tener una partición.

Las instancias fueron generadas aleatorizando cada una de las variables mencionadas. El código del generador se encuentra en el Apéndice.

5.2.1. Test de configuración

Lo que primero necesitamos es tener una idea de cuál configuración usar en la GRASP. Para ello, testearemos con todas las combinaciones de configuraciones para un set acotado de valores, y discutiremos al respecto sobre cómo interpretar los resultados obtenidos, y así decidir una configuración.

Tenemos dos criterios de parada: parar por un límite α de iteraciones máximo, o parar si una cierta cantidad β de instancias pasaron sin haber mejora en la solución. Por el el lado de la heurística aleatorizada, es decir, para la selección de candidatos, tenemos dos variables independientes: la profundidad de la elección del próximo vértice a insertar, y la profundidad de la elección del conjunto en que va a ser insertado el vértice.

Parar por un máximo de iteraciones α es el criterio más sencillo, pero tiene inconvenientes. Por un lado, podemos estar haciendo muchas iteraciones de más ya que rápidamente encontramos la solución óptima; esto ocurre siempre con grafos con pocos nodos, que son fáciles de resolver. Por otro lado, puede pasar que la cantidad fija de iteraciones que fue seteada no sea suficiente, y que obtengamos soluciones subóptimas incluso para lo que puede dar GRASP.

Parar por iteraciones sin mejora es más flexible porque si se encuentra rápidamente la solución óptima, no va a haber mejora en las próximas β iteraciones, por lo cual el algoritmo va a terminar mucho más rápido para este tipo de casos. Para el caso en que sí haya mejoras todo el tiempo, incluso aunque $\beta < \alpha$ el algoritmo seguirá ejecutando hasta que pasen β iteraciones sin mejorar, lo cual puede hacer que el total de iteraciones sea mucho mayor a α, pero consiguiendo una solución con mejor calidad que parar por máximo de iteraciones. De todas formas, podría pasar que las soluciones obtenidas de la golosa y mejoradas con la búsqueda local, sean peores que la mejor hasta el momento durante β iteraciones, y que si $\beta < \alpha$, se termine haciendo menos iteraciones que α, por lo cual no hay garantía de que parar por iteraciones sin mejora en el caso $\beta < \alpha$ vaya a devolver mejores soluciones que parar en α iteraciones.

Si tomamos $\beta < \alpha$, cabe preguntarse qué valor de β hace falta para que parar por iteraciones sin mejora sea mejor que parar por máximo de iteraciones. Fijamos $\alpha = 100$, y testeamos con $\beta = 10, 35, 50, 70$.

Pero todavía falta la selección de candidatos: plantearemos que ambas profundidades puedan tomar los valores 1, 2 ó 4. Como notación, cuando decimos profundidad (x_1, x_2) significa profundidad de elección de vértice x_1 y profundidad de elección de conjunto x_2. Notar que la profundidad $(1, 1)$ es equivalente a la golosa pura, sin aleatoriedad. Dejamos esa opción para verificar que la aleatoriedad es efectivamente necesaria para obtener mejores soluciones. Con respecto a esto, conjeturamos que lo mejor es aleatorizar lo máximo posible, entonces esperamos ver que $(4, 4)$ sea la mejor configuración de selección.

Dado un n, para cada instancia que tenga un grafo de n nodos, se va ejecutar GRASP con cada configuración por separado y se van a acumular los resultados. Luego, se busca cuál es la mejor configuración para este n hallando el mínimo de las sumas de pesos para cada configuración. El mínimo se calcula de la siguiente manera:

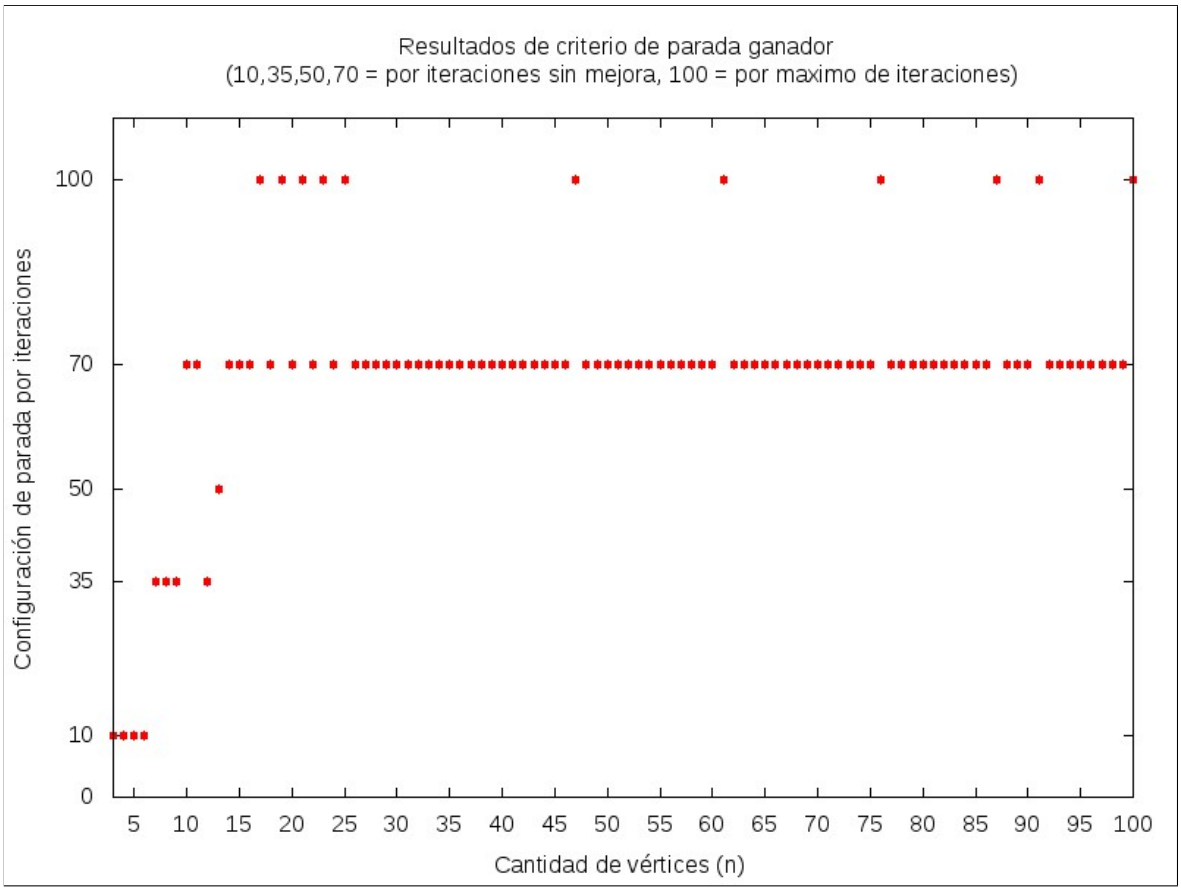
Algorithm 7 Cálculo del mínimo peso de las configuraciones para un n

```
mejorPesoAcumulado ← +∞
for valor de iteraciones sin mejora (10, 35, 50, 70) do
    for cada valor de profundidad de vértice (4, 2, 1) do
        for cada valor de profundidad de conjunto (4, 2, 1) do
            if Peso acumulado de esta configuración < mejorPesoAcumulado then
                Actualizar mejorPesoAcumulado
                Poner la actual como mejor configuración
            end if
        end for
    end for
end for
for valor de máximo de iteraciones (100) do
    for cada valor de profundidad de vértice (4, 2, 1) do
        for cada valor de profundidad de conjunto (4, 2, 1) do
            if Peso acumulado de esta configuración < mejorPesoAcumulado then
                Actualizar mejorPesoAcumulado
                Poner la actual como mejor configuración
            end if
        end for
    end for
end for
```

Empezamos buscando el mínimo parando por iteraciones sin mejora, y después de encontrarlo, vemos si por máximo de iteraciones es mejor para ese n. Dentro de cada configuración de parada, calculamos para cada profundidad (recorriéndolas de esta manera: $(4,4)$, $(4,2)$, $(4,1)$, $(2,4)$, etc). Si como conjeturamos, $(4,4)$ es la mejor profundidad, las demás no deberían dar pesos acumulados menores, y debería ganar siempre $(4,4)$. Lo mismo para el criterio de parada, si parar por 10 iteraciones consigue la solución óptima, y las demás configuraciones no la mejoran, ésa es la elegimos como mejor porque hizo menos iteraciones que las demás.

Por otro lado, para el total del conjunto de instancias hacemos esta misma acumulación de pesos también separando por configuración, y buscamos de esta manera obtenemos la configuración de mínimo peso en las 10.000 instancias.

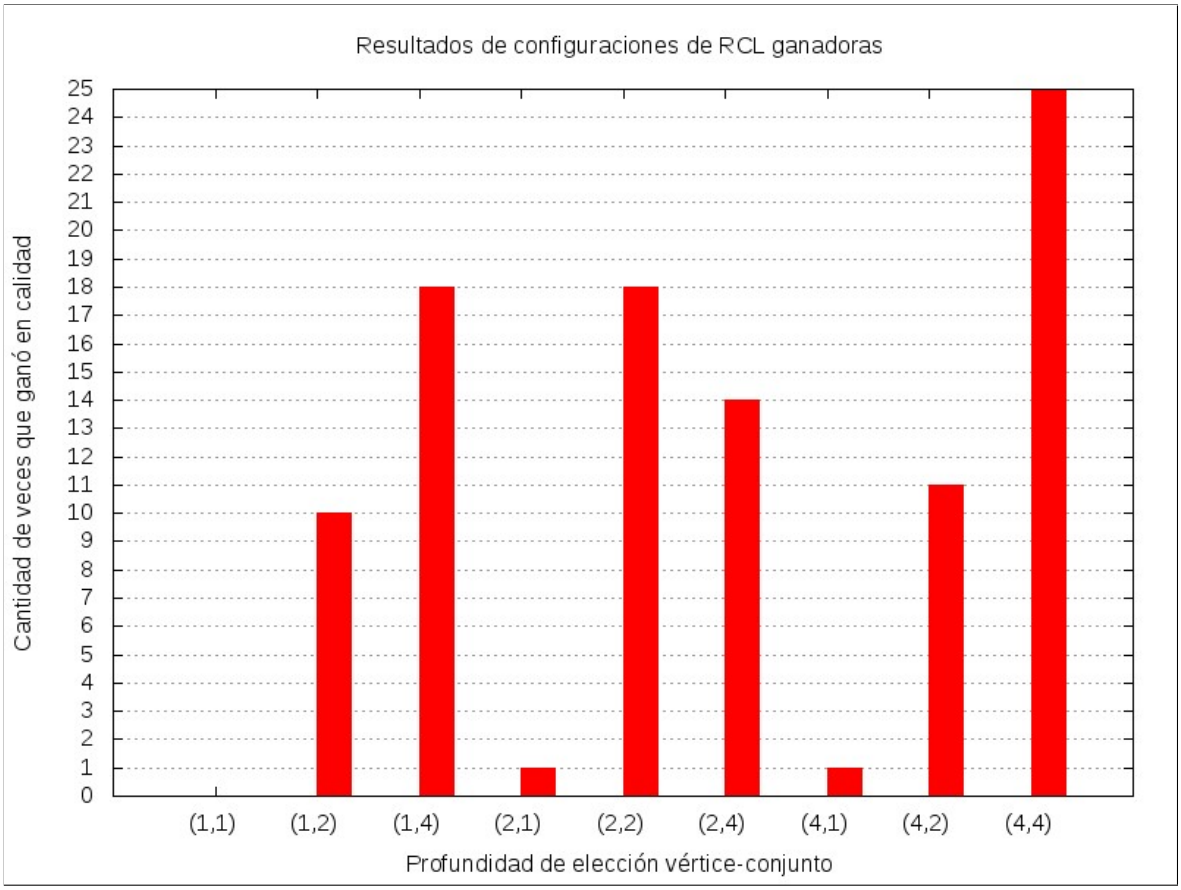
Veamos primero cuál de los dos criterios de parada resultó ganador para cada n. El siguiente gráfico se interpreta de la siguiente manera: Si y(n) es 10, 35, 50, ó 70, entonces ganó iteraciones sin mejora con valor y(n). Si es 100, entonces ganó máximo de iteraciones, con ese valor (que es el único).



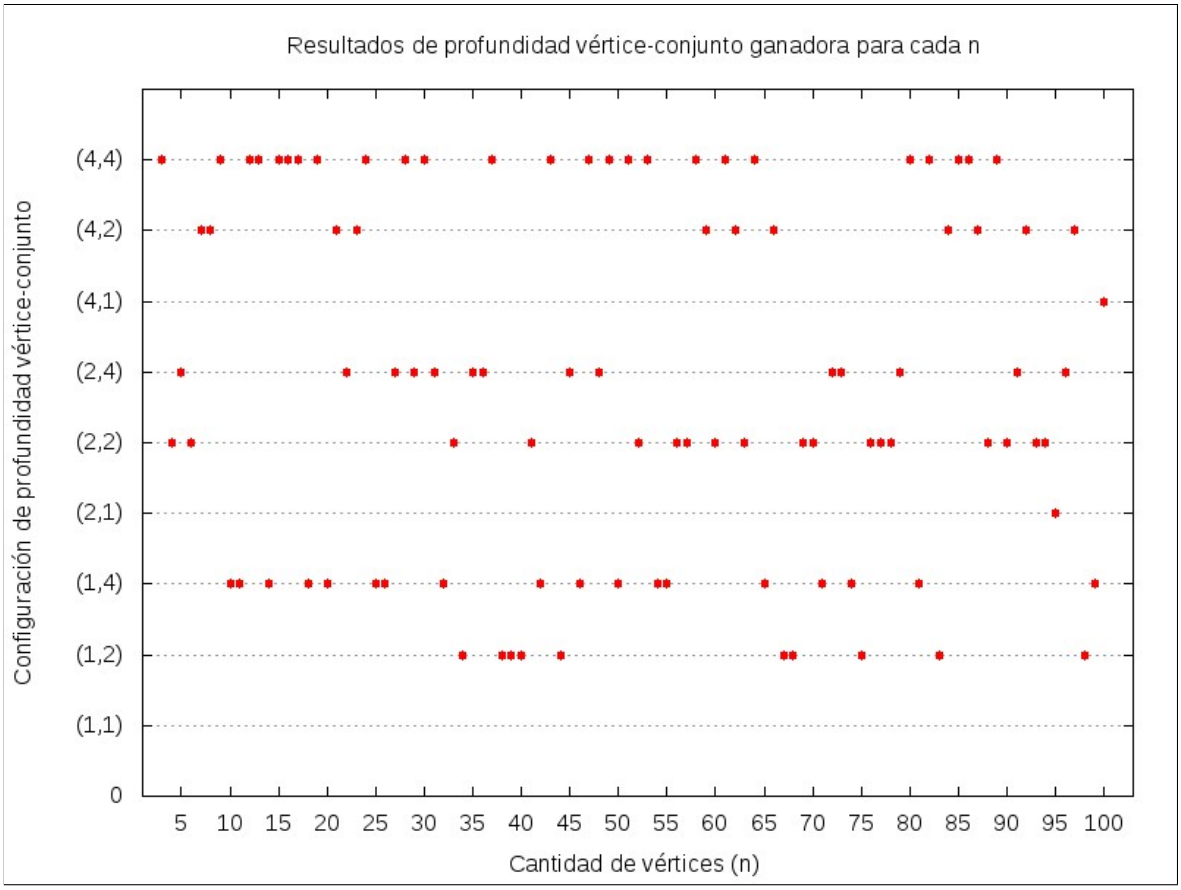
Podemos observar que para valores de n menores a 15, muchas veces alcanza con 10, 35 ó 50 iteraciones sin mejora para ganar; es decir, en ningún caso usar las 100 iteraciones del otro criterio de parada mejora la solución, como preveíamos. Pero al aumentar el n, aunque claramente parar por 70 iteraciones sin mejora gana más veces, observamos que a veces resulta mejor el criterio de máximo de iteraciones, lo cual muestra falta de garantía de parar por iteraciones sin mejora.

Independientemente de qué criterio de parada sea mejor para cada n, veamos qué profun-

didad resultó ganadora. Veamos primero para cuantos n resultó ganadora cada una:



La primera observación es que usar de profundidad $(x, 1)$ no es conveniente. En particular, $(1, 1)$, que es equivalente a que la heurística golosa no tenga aleatoriedad, no resultó ganadora para ningún caso, como esperábamos. Usar $(4, 4)$ gana más veces (25), y usar $(x, 4)$ gana 57 veces, contra 39 ganadas de $(x, 2)$. Fijando los conjuntos, $(1, x)$ gana 28 veces, $(2, x)$ gana 33, y $(4, x)$ gana 37 veces. Hasta ahora pareciera que $(4, 4)$ es la mejor profundidad, pero tenemos que ver más detalladamente qué ocurre para cada n, porque si por ejemplo $(4, 4)$ sólo ganara para las primeros 25 cantidades de nodos, pero fuera superada para $n \geq 27$, claramente sería una profundidad que no funciona para grafos con una cantidad elevada de nodos. Veamos entonces cuál profundidad gana para cada n:



Podemos ver en el gráfico que $(4, 4)$ gana de manera consistente, excepto para los n más altos, para los cuales no parece haber claramente un ganador, ya que ganan casi todas las configuraciones. Hay un problema más: no sabemos exactamente por cuánto gana una profundidad. Vamos a usar el cálculo paralelo que hicimos, la suma de los pesos de cada configuración para las 10.000 instancias, para ver qué tan alejadas están verdaderamente las configuraciones de profundidad. Veamos los pesos acumulados de iteraciones sin mejora con valor 70 para cada profundidad (x, y):

(x,y)	1	2	4
1	459037888	432225472	432218464
2	433368064	432117408	432198464
4	432523232	432218784	432113312

Cuadro 1: Peso acumulado para cada configuración

(x,y)	1	2	4
1	6.2309 %	0.0259 %	0.0243 %
2	0.2903 %	0.0009 %	0.0197 %
4	0.0948 %	0.0244 %	0.0000 %

Cuadro 2: Error relativo de cada configuración contra el peso de la configuración $(4, 4)$

$(4, 4)$ logra el menor peso, y $(1, 1)$ el peor (siendo 6,23 % más pesada). En particular, la primera columna, es decir, las profundidades $(x, 1)$ son las tres peores. Pero el resto de las configuraciones no son mucho más pesadas que $(4, 4)$ (la más pesada de éstas, $(1, 2)$, sólo es un 0,026 % más pesada. En particular, $(2, 2)$ es la más cercana siendo sólamente 0,0009479 % más pesada. Si tuviéramos que elegir entre fijar alguna profundidad en 1, y poder variar la otra, claramente podemos sacar como conclusión que aleatorizar la elección del conjunto es más importante que aleatorizar la elección del vértice a insertar. Aunque por otro lado, vemos que aumentar la profundidad de la elección de vértices también mejora las soluciones.

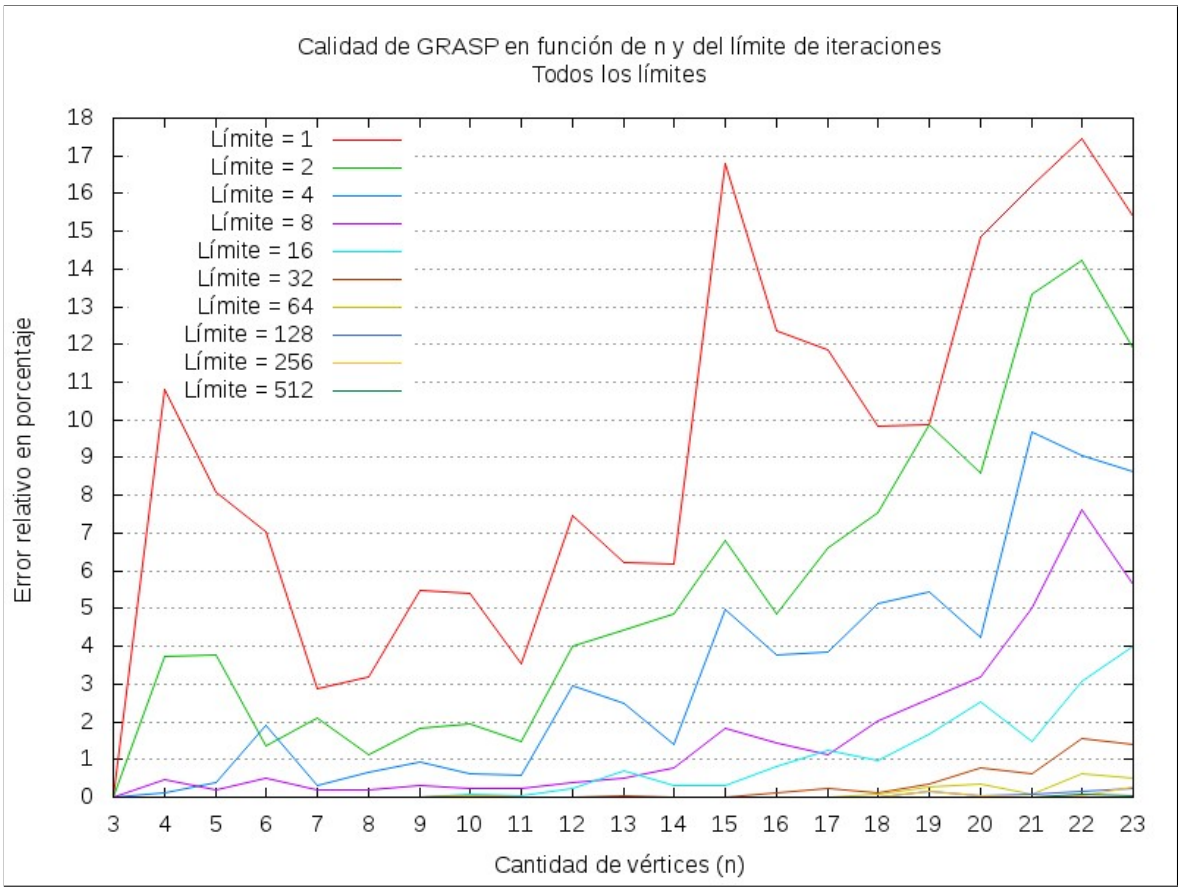
Por todo lo anterior, decidimos usar la configuración siguiente:

- Criterio de parada: iteraciones sin mejora.
- Profundidad elección vértice-conjunto: $(4, 4)$.

5.2.2. Test de calidad

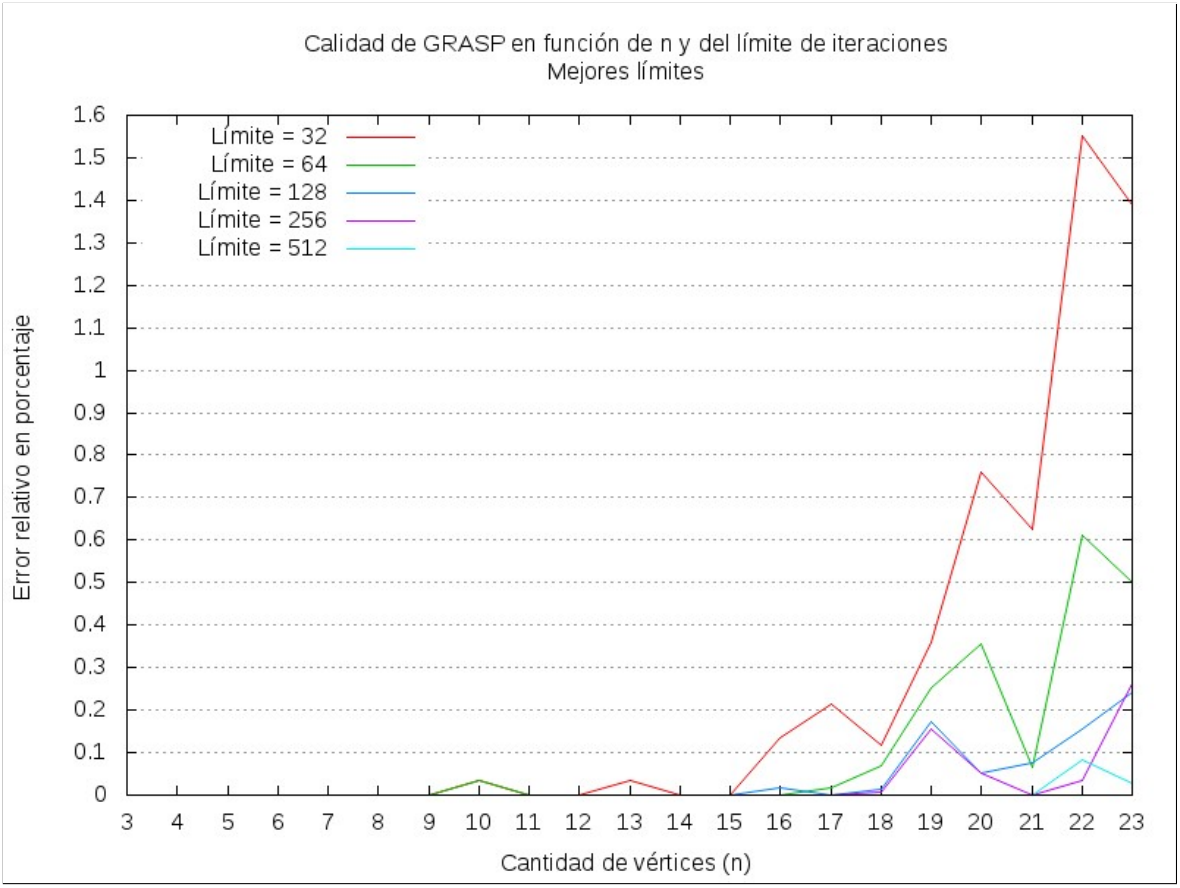
En este test comparamos los pesos de las soluciones dadas por la GRASP con la configuración que elegimos (iteraciones sin mejora y profundidad $(4, 4)$), contra los pesos de las soluciones óptimas obtenidas del algoritmo exacto. Lo hicimos hasta $n = 23$ por la complejidad temporal no polinomial del algoritmo exacto.

Variamos el límite de iteraciones en potencias de 2: tomamos los valores 2^i con $i = \{0, 1, ..., 9\}$. Para cada uno, calculamos el promedio de los errores relativos (en porcentaje) de las intancias de n nodos, para $n = \{3, 4, ..., 23\}$. Veamos los resultados:



Como era de esperar, a mayor límite de iteraciones, mejor solución devuelve la GRASP. Tomar como límite una sola iteración tiene la peor performance, superando en el peor caso el 17 %. Además, podemos ver que el error relativo para cualquier valor de iteraciones aumenta con el n, lo cual sugiere que no alcanza con fijar un cierto límite si no se sabe la máxima cantidad de vértices que van a tener los grafos de las instancias de entrada.

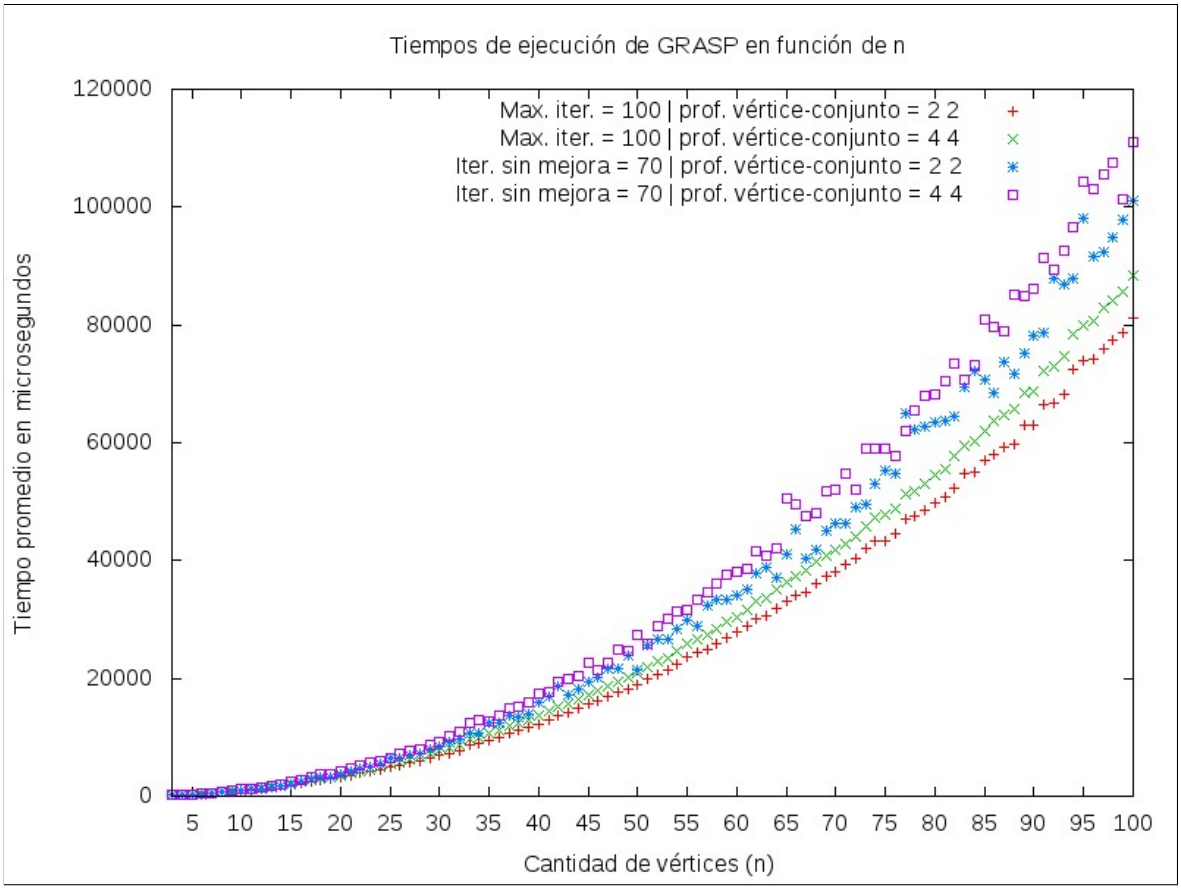
Veamos más en detalle lo que ocurre con los límites de iteraciones más altos, que dan los mejores resultados:



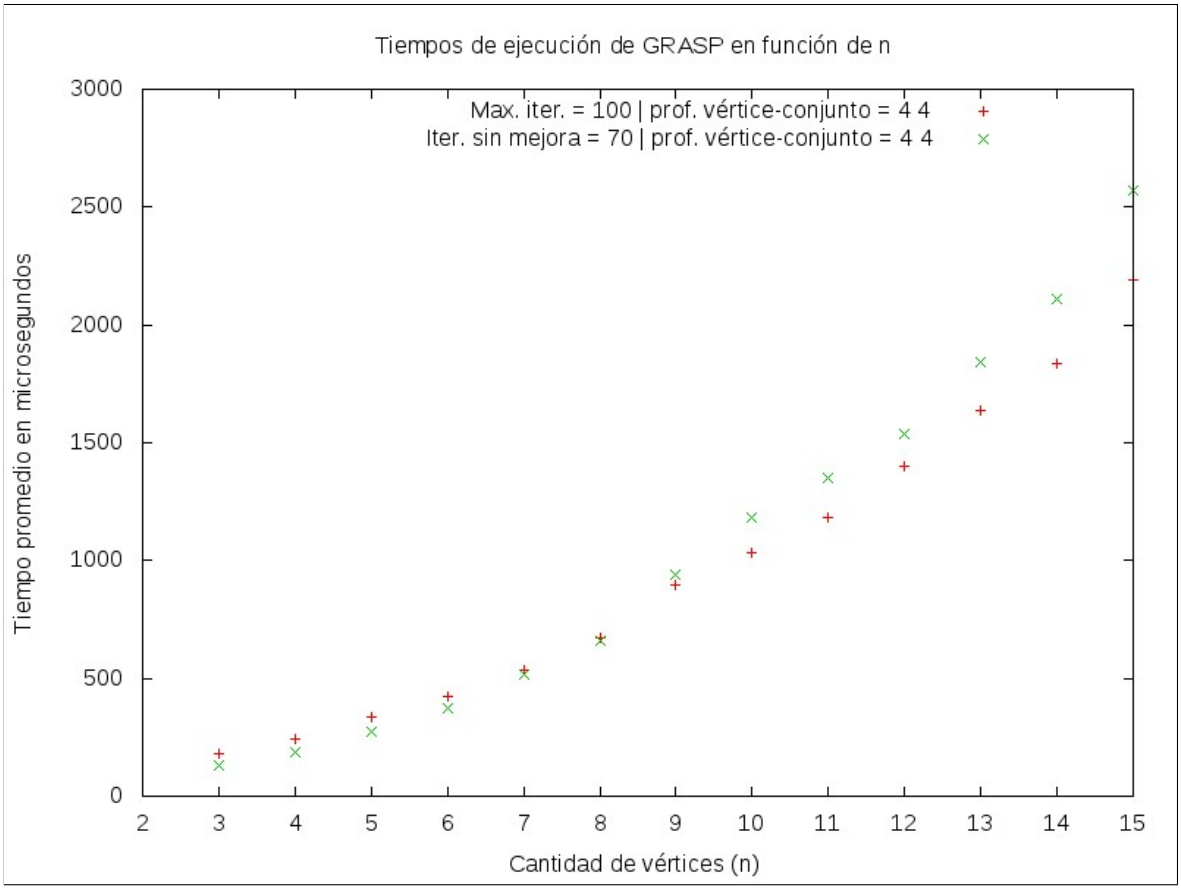
Nuevamente notamos que sigue valiendo la observación anterior para los mejores límites, esto es, tienden a crecer con n, incluso lo más altos. Solamente usar 512 iteraciones se mantiene por debajo del 0,1 %, pero conjeturamos que al aumentar n, también crecerá y deberá usarse más de 512 iteraciones de límite para obtener una calidad menor a esa cota.

5.2.3. Test de tiempo de ejecución

Calculamos los promedios de los tiempos de ejecución para las instancias de cada n, con $n = \{3, 4, ..., 100\}$. Lo hicimos para los dos criterios de parada que usamos en el test de configuración, es decir, usamos máximo de iteraciones con límite 100, e iteraciones sin mejora con límite 70; y para cada uno, usamos dos profundidades: $(2, 2)$ y $(4, 4)$. Con esto queremos ver si como supusimos, usar el criterio de iteraciones sin mejora efectivamente hace menos iteraciones para valores bajos n, y hace más para valores altos. Además queremos testear si aumentar la profundidad también aumenta los tiempos.



Vemos por un lado que a mayor n, el criterio de iteraciones sin mejora con ambas profundidades toma más tiempo que el criterio de iteraciones máximas, y que para cada una de ellas, usar más profundidad lleva también más tiempo. Lo primero lo explicamos con nuestra hipótesis: iteraciones sin mejora hace más iteraciones que el máximo a mayor n, y esta es una de las razones por las que termina dando mejores soluciones. Por otro lado, usar más profundidad hace que la golosa aleatorizada provea soluciones más distantes, y que haya más probabilidad de que la búsqueda local las mejore para superar a la mejor partición hasta el momento. Esto sugiere que usar más profundidad genera mejores soluciones. Falta ver qué ocurre para los primeros n:



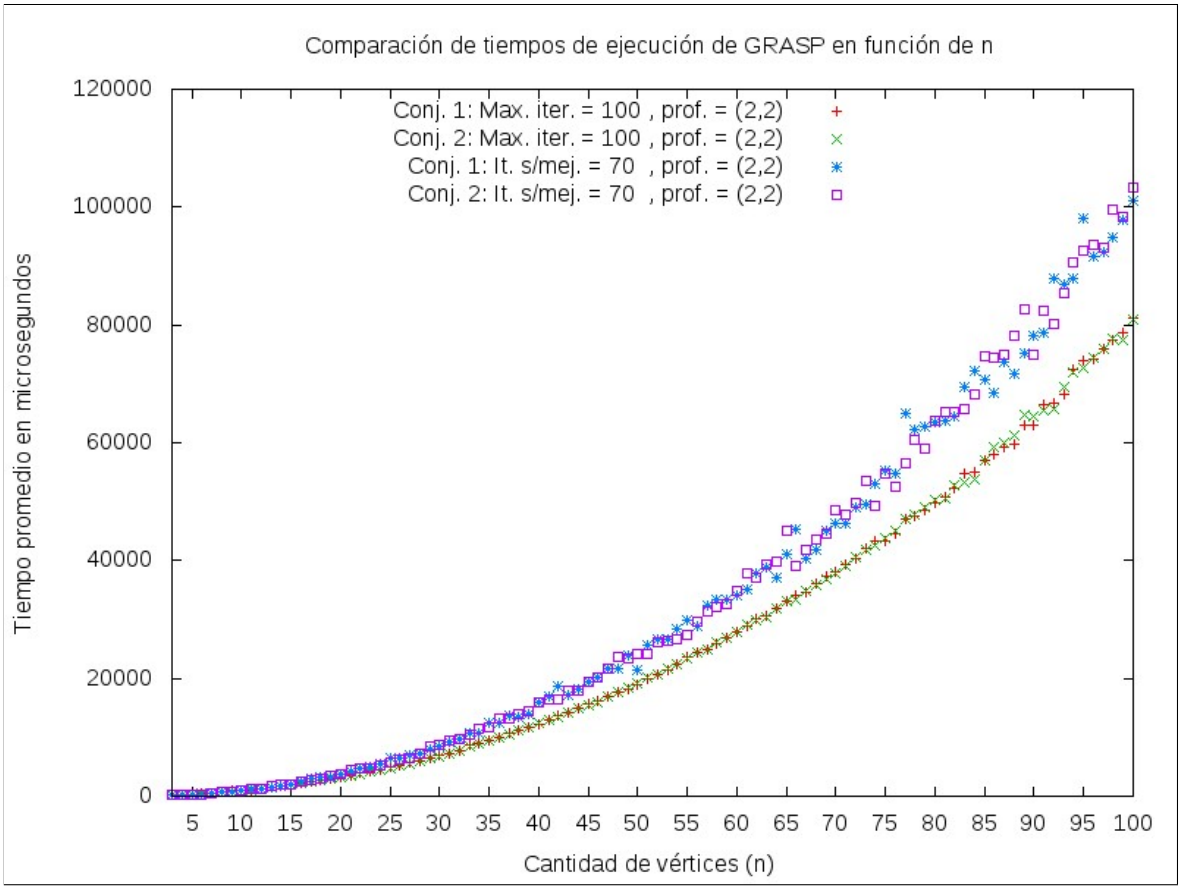
Como preveíamos, para grafos pequeños termina antes usar iteraciones sin mejora, aunque rápidamente iteraciones sin mejora comienza a superar las 100 iteraciones, y por esto es mayor para $n \geq 9$. La razón por la cual la diferencia es poco apreciable es que estamos usando como límite de iteraciones sin mejora a 70, pero ya vimos en el test de configuración que para valores menores a 15 podríamos usar 10, 35 ó 50 y así —sin perder calidad— obtener mayores diferencias en el tiempo de ejecución.

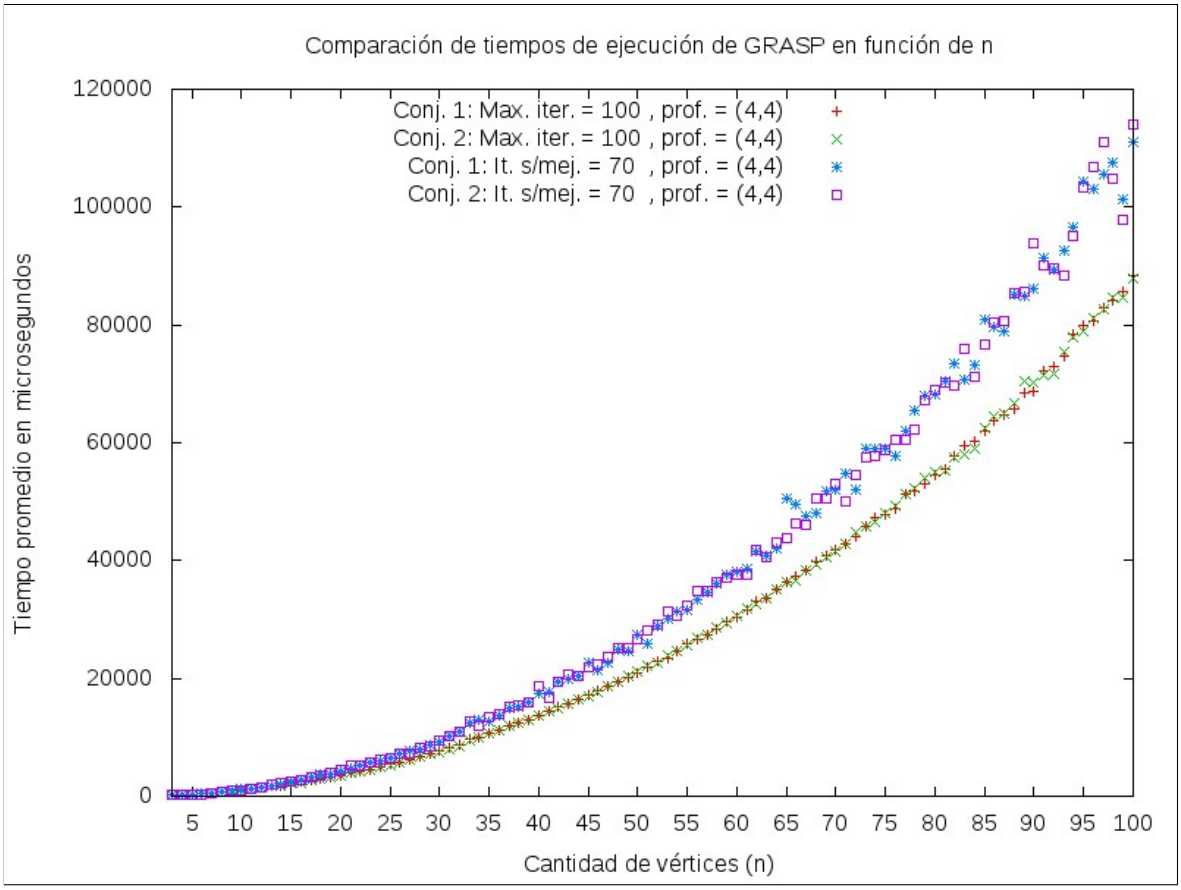
6. Comparación con nuevo conjunto de instancias y conclusiones

Para un nuevo conjunto de instancias, generado de la misma manera que el que usamos para elegir la mejor configuración, corrimos los tests de tiempos de ejecución y de calidad y comparamos con la primera tanda de instancias.

6.1. Comparación de tiempos de ejecución

Esperábamos que esto no se vea afectado por el nuevo conjunto de instancias, que es lo que efectivamente ocurre. Por claridad, separamos los gráficos por profundidad:


Comparación de tiempos de ejecución de GRASP en función de n
Conj. 1: Max. iter. = 100 , prof. = (2,2)
Conj. 2: Max. iter. = 100 , prof. = (2,2)
Conj. 1: It. s/mej. = 70 , prof. = (2,2)
Conj. 2: It. s/mej. = 70 , prof. = (2,2)
Tiempo promedio en microsegundos
Cantidad de vértices (n)
0
20000
40000
60000
80000
100000
120000
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100



6.2. Comparación de calidades

Este test sí puede dar lugar a diferencias apreciables, ya que fijamos una configuración que no tenemos garantía de que sea óptima. Veamos primero la forma que tienen las calidades solamente del segundo conjunto:

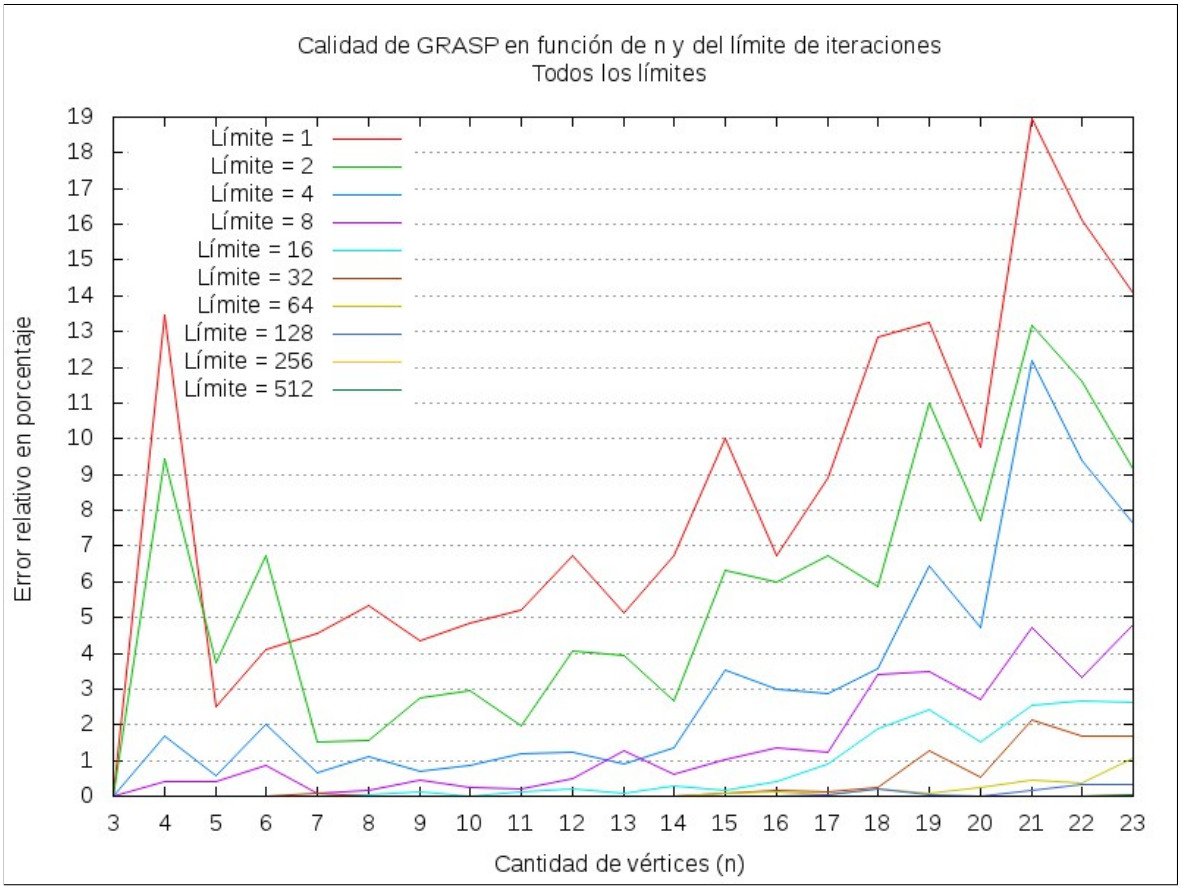


Figura 11: Calidad del segundo conjunto de instancias

A simple vista parece ser similar al gráfico obtenido para el primer conjunto, con los errores relativos creciendo con el n. Hagamos una comparación de las calidades de los mejores límites de iteraciones de un conjunto, contra el nuevo:

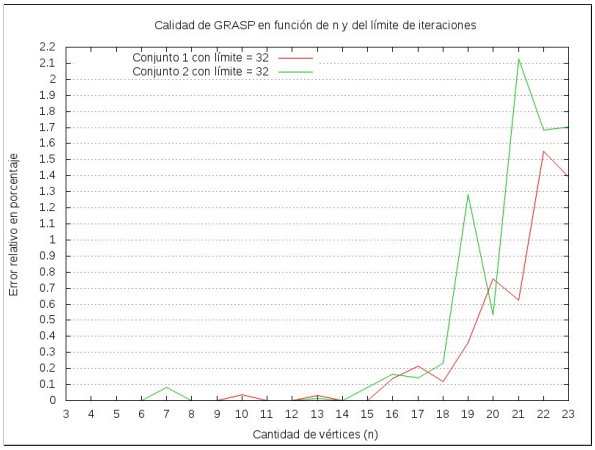


Figura 12: Comparación de calidad para límite de iteraciones = 32

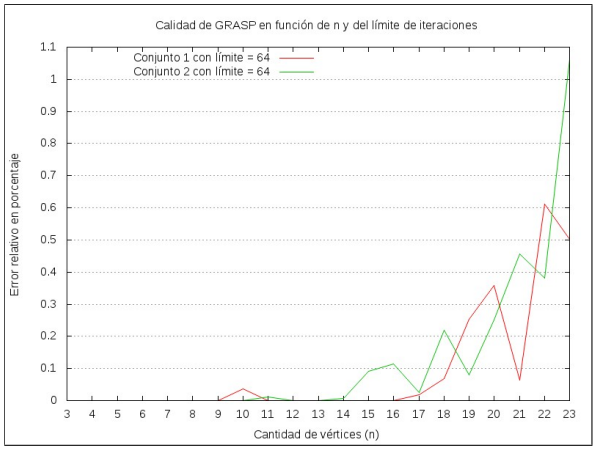


Figura 13: Comparación de calidad para límite de iteraciones = 64

Para 32 iteraciones, el nuevo conjunto de instancias devuelve soluciones con una calidad peor para la mayoría de los casos, llegando a tener en los peores casos el doble de error relativo

que el primer conjunto. Para 64, es menos claro, aunque para 14 y 15 nodos ni siquiera se llega a una solución óptima en el segundo conjunto pero sí en el primero, y para $n = 23$ se duplica el error relativo con respecto al primer conjunto.

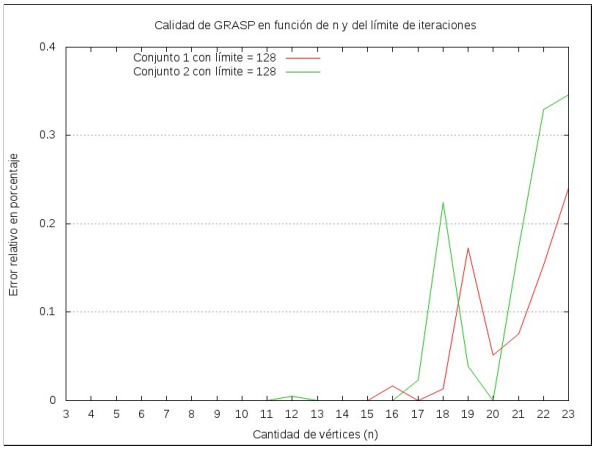


Figura 14: Comparación de calidad para límite de iteraciones = 128

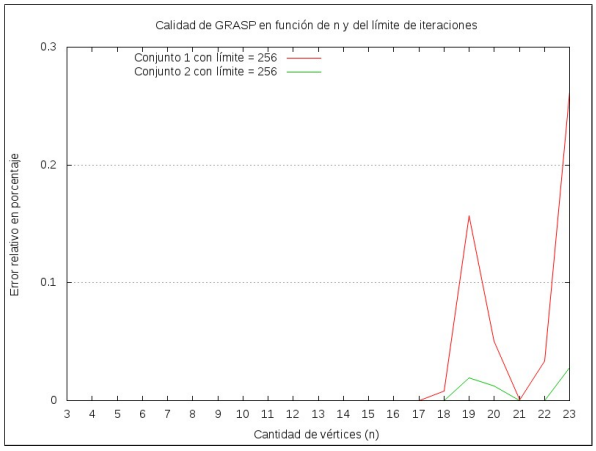


Figura 15: Comparación de calidad para límite de iteraciones = 256

Para el límite de 128 iteraciones ocurre algo similar que para 64, aunque para 256 el que obtiene claramente la peor calidad es el primer conjunto, mientras el segundo se mantiene incluso por debajo de la barrera de 0,05 %.

6.3. Conclusiones

Discutimos sobre criterios de parada y sobre la selección de candidatos de la golosa aleatorizada, y elegimos una configuración en base a los resultados obtenidos para un primer conjunto de instancias.

Testeamos sobre dos criterios de parada y comentamos sus beneficios y problemas. En una aplicación real, sería conveniente por lo menos usar una combinación de ambos criterios, es decir, iteraciones sin mejora pero con un límite máximo de iteraciones global. Por otro lado, para distintos valores de n, vimos que es conveniente variar los límites para reducir el tiempo de ejecución, por lo cual sería interesante tener límites dinámicos según la cantidad de vértices del grafo de entrada.

Para la selección de candidatos de la golosa aleatorizada, testeamos varios niveles de profundidad de elección de vértice y de elección de conjunto. No fue claro en particular que la mejor profundidad sea $(4, 4)$, ya que aunque resultó ganadora, los errores relativos eran tan cercanos a cero que bien podría tratarse de una acumulación de los errores de cálculo inherentes a la aritmética finita. Sí es más claro que es más importante aleatorizar la elección del conjunto destino.

Elegimos la configuración de iteraciones sin mejora y profundidad de elección de vértice-conjunto $(4, 4)$ y comparamos los resultados del primer conjunto con un nuevo conjunto de iteraciones. Vimos que aunque los tiempos de ejecución se mantuvieron intactos, la calidad no fue la esperada, teniendo en algunos casos el doble de error relativo para el segundo conjunto. Esto sugiere que no hay garantía de que una configuración funcione relativamente bien siempre, la única garantía de calidad parece ser la cantidad de iteraciones que realiza la GRASP.

7. Apéndice: Códigos fuente

7.1. Algoritmo exacto

```
1  void kpmp(vector<vector<double> > &adym, Partition &partition, Partition
       &min_partition, int node, map<string, bool> &options)
2  {
3    // check if has added all nodes
4    if (node == adym.size()) {
5      if (partition.weight < min_partition.weight &&
6          partition.subsets_used <= partition.allowed_subsets) {
7        min_partition = partition;
8      }
9      return;
10   }
11
12   // poda k_subsets
13   if (options["k_subsets"]) {
14     int subsets_left = partition.allowed_subsets - partition.subsets_used;
15     if (nodes_left(adym.size(), node) == subsets_left) {
16       partition.push_back_to_subset(partition.subsets_used, node);
17       kpmp(adym, partition, min_partition, node+1, options);
18       partition.pop_back_from_subset(partition.subsets_used-1);
19     }
20   }
21
22   // poda pesada
23   if (options["can_improve"]) {
24     if (partition.subsets_used == partition.allowed_subsets) {
25       double min_weight_of_adding_rest = 0;
26       for (int current_node = node; current_node < adym.size();
             current_node++) {
27         double min_weight_in_subset = numeric_limits<double>::max();
28         for (int subset = 0; subset < partition.subsets_used; ++subset) {
29           double current_weight = partition.weight_in_subset(adym,
                 current_node, subset);
30           if (current_weight < min_weight_in_subset) {
31             min_weight_in_subset = current_weight;
32           }
33         }
34         min_weight_of_adding_rest += min_weight_in_subset;
35       }
36
37       if (min_weight_of_adding_rest + partition.weight >=
             min_partition.weight) {
38         return;
39       }
40     }
41   }
42
43   // try adding it to every non-empty subset
44   for (int subset = 0; subset < partition.subsets_used; ++subset) {
45     double added_weight = partition.weight_in_subset(adym, node, subset);
46     // poda min_weight
47     if (options["min_weight"]) {
48       if (partition.weight+added_weight < min_partition.weight) {
49         partition.push_back_to_subset(subset, node);
50         partition.weight += added_weight;
51         kpmp(adym, partition, min_partition, node+1, options);
52         partition.pop_back_from_subset(subset);
53         partition.weight -= added_weight;
54       }
```

```
55      } else {
56        partition.push_back_to_subset(subset, node);
57        partition.weight += added_weight;
58        kpmp(adym, partition, min_partition, node+1, options);
59        partition.pop_back_from_subset(subset);
60        partition.weight -= added_weight;
61      }
62    }
63
64    if (options["k_subsets"]) {
65      if (partition.subsets_used < partition.allowed_subsets) {
66        // try adding new partition
67        partition.push_back_to_subset(partition.subsets_used, node);
68        kpmp(adym, partition, min_partition, node+1, options);
69        partition.pop_back_from_subset(partition.subsets_used-1);
70      }
71    } else {
72      if (partition.subsets_used < partition.total_subsets()) {
73        partition.push_back_to_subset(partition.subsets_used, node);
74        kpmp(adym, partition, min_partition, node+1, options);
75        partition.pop_back_from_subset(partition.subsets_used-1);
76      }
77    }
78  }
```

7.2. Heurística golosa constructiva (pura)

```
std::vector<std::set<int>> Heuristica::resolverGolosoPuro() {
    std::vector<std::set<int>> res(k_);
    int n = grafo_.getCantidadVertices();
    if (n <= k_) {
        for (int i = 0; i < n; i++) {
            res[i].insert(i);
        }
    } else {
        std::vector<int> verticesOrdenadosPorPeso = ordenarPorPesoEnGrafo();
        for (auto & v : verticesOrdenadosPorPeso) {
            double mejorPeso = pesoEnSubconjunto(v, res[0]);
            int mejorSubconjunto = 0;
            for (int i = 1; i < k_; i++) {
                int pesoEnSubconj = pesoEnSubconjunto(v, res[i]);
                if ( pesoEnSubconj < mejorPeso ) {
                    mejorPeso = pesoEnSubconj;
                    mejorSubconjunto = i;
                }
            }
            res[mejorSubconjunto].insert(v);
        }
    }
    return res;
}
```

7.3. Heurística golosa constructiva aleatorizada

```
1  std::vector<std::set<int>> Heuristica::resolver() {
2      std::vector<std::set<int>> res(k_);
3      std::vector<int> verticesOrdenadosPorPeso = ordenarPorPesoEnGrafo();
4      int cantidadConjuntosCandidatos = std::min(k_,
           profundidadEleccionConjunto_);
5      while (!verticesOrdenadosPorPeso.empty()) {
6          int cantidadVerticesCandidatos =
               std::min((int)verticesOrdenadosPorPeso.size(),
               profundidadEleccionVertice_);
7          int indicePorRemover = rand() % cantidadVerticesCandidatos;
8          int verticeNuevo = verticesOrdenadosPorPeso[indicePorRemover];
9          verticesOrdenadosPorPeso.erase(verticesOrdenadosPorPeso.begin() +
               indicePorRemover);
10         std::vector<std::pair<int,double>> infoConjuntos; // Esto guarda las
               tuplas <conjunto i, peso del vertice en conjunto i>
11         for (int i = 0; i < k_; i++) {
12             infoConjuntos.push_back(std::make_pair(i,
                   pesoEnSubconjunto(verticeNuevo, res[i])));
13         }
14         std::sort(infoConjuntos.begin(), infoConjuntos.end(), [] (const
               std::pair<int,double> & a, const std::pair<int,double> & b) {
               return a.second < b.second; });
15         res[infoConjuntos[rand() %
               cantidadConjuntosCandidatos].first].insert(verticeNuevo);
16     }
17     return res;
18 }
```

7.4. Heurística de búsqueda local con vecindad (A)

```
int main(int argc, char *argv[])
{
  int n, m, k;
  cin >> n >> m >> k;

  // having w((u, v)) = 0 || (u, v) not in E is the same
  vector<vector<double> > adym(n, vector<double> (n, 0));
  int u, v;
  double w;
  double total_weight = 0;
  for(int i = 0; i < m; i++) {
    cin >> u >> v >> w;
    adym[u][v] = w;
    adym[v][u] = w;
    total_weight += w;
  }

  vector<set<int> > partition(k, set<int>());
  for(int i = 0; i < n; ++i) {
    partition[0].insert(i);
  }

  cout << "Starting␣with␣a␣total␣weight␣of␣" << total_weight << endl;
  vector<int> node_indexed_partition(n, 0);
  bool has_improved = true;
  while (has_improved) {
    has_improved = false;
    for (int i = 0; i < n; ++i) {
      double node_weight_in_current_subset =
        node_weight_in_subset(i, node_indexed_partition[i], partition, adym);
      bool swapped = false;
      int subset = 0;
      while (!swapped && subset < k) {
        if (subset != node_indexed_partition[i]) {
          cout << "Considering␣swapping␣node␣" << i << "␣from␣subset␣" <<
              node_indexed_partition[i] << "␣to␣subset␣" << subset << endl;
          double node_weight_in_subset_j = node_weight_in_subset(i, subset,
              partition, adym);
          if (node_weight_in_current_subset > node_weight_in_subset_j) {
            cout << "FOUND␣IMPROVEMENT!" << endl;
            cout << "␣Swapping␣node␣" << i << "␣from␣subset␣" <<
                node_indexed_partition[i] << "␣to␣subset␣" << subset << endl;
            partition[node_indexed_partition[i]].erase(i);
            partition[subset].insert(i);
            node_indexed_partition[i] = subset;
            total_weight = total_weight - node_weight_in_current_subset +
                node_weight_in_subset_j;
            cout << "New␣total␣weight␣is:␣" << total_weight << endl;
            has_improved = true;
            swapped = true;
          }
        }
        ++subset;
      }
    }
  }

  cout << "Minimum␣weight␣reached:␣" << total_weight << endl;
  cout << "PARTITION" << endl;
  for (int i = 0; i < k; ++i) {
    cout << "Partition␣" << i << ":␣";
```

```
58      for(set<int>::iterator it = partition[i].begin(); it !=
            partition[i].end(); ++it) {
59        cout << *it << ' ';
60      }
61      cout << endl;
62    }
63
64    return 0;
65  }
```

7.5. Heurística de búsqueda local con vecindad (B)

```
1  int main(int argc, char *argv[])
2  {
3    int n, m, k;
4    cin >> n >> m >> k;
5
6    // having w((u, v)) = 0 || (u, v) not in E is the same
7    vector<vector<double> > adym(n, vector<double> (n, 0));
8    int u, v;
9    double w;
10   double total_weight = 0;
11   for(int i = 0; i < m; i++) {
12     cin >> u >> v >> w;
13     adym[u][v] = w;
14     adym[v][u] = w;
15     total_weight += w;
16   }
17
18   vector<set<int> > partition(k, set<int>());
19   for(int i = 0; i < n; ++i) {
20     partition[0].insert(i);
21   }
22
23   cout << "Starting␣with␣a␣total␣weight␣of␣" << total_weight << endl;
24   vector<int> node_indexed_partition(n, 0);
25   bool has_improved = true;
26   while (has_improved) {
27     has_improved = false;
28     for (int first_node = 0; first_node < n; ++first_node) {
29       double first_node_current_weight = node_weight_in_subset(first_node,
             node_indexed_partition[first_node], partition, adym, -1);
30       bool swapped = false;
31       int second_node = first_node+1;
32       while (!swapped && second_node < n) {
33         if (node_indexed_partition[second_node] !=
               node_indexed_partition[first_node]) {
34           cout << "Considering␣swapping␣node␣" << first_node << "␣from␣
                 subset␣" << node_indexed_partition[first_node] <<
35             "␣for␣node" << second_node << "␣on␣subset␣" <<
                   node_indexed_partition[second_node] << endl;
36           double second_node_current_weight =
                 node_weight_in_subset(second_node,
                 node_indexed_partition[second_node], partition, adym, -1);
37           double first_node_new_weight = node_weight_in_subset(first_node,
                 node_indexed_partition[second_node], partition, adym,
                 second_node);
38           double second_node_new_weight = node_weight_in_subset(second_node,
                 node_indexed_partition[first_node], partition, adym,
                 first_node);
39           // if what I remove is greater than what I'm adding
40           bool swap_improves =
                 (first_node_current_weight+second_node_current_weight) >
                 (first_node_new_weight+second_node_new_weight);
41           if (swap_improves) {
42             cout << "FOUND␣IMPROVEMENT!" << endl;
43             cout << "␣Swapping␣node␣" << first_node << "␣from␣subset␣" <<
                   node_indexed_partition[first_node] << "for␣node␣" <<
44               second_node << "␣on␣subset␣" <<
                     node_indexed_partition[second_node] << endl;
45             partition[node_indexed_partition[first_node]].erase(first_node);
46             partition[node_indexed_partition[second_node]].erase(second_node);
47             partition[node_indexed_partition[first_node]].insert(second_node);
```

```
48              partition[node_indexed_partition[second_node]].insert(first_node);
49              int temp = node_indexed_partition[second_node];
50              node_indexed_partition[second_node] =
                    node_indexed_partition[first_node];
51              node_indexed_partition[first_node] = temp;
52              total_weight = total_weight -
                    (first_node_current_weight+second_node_current_weight) +
                    (first_node_new_weight+second_node_new_weight);
53              cout << "New total weight is: " << total_weight << endl;
54              has_improved = true;
55              swapped = true;
56            }
57          }
58          ++second_node;
59        }
60      }
61    }
62
63    cout << "Minimum weight reached: " << total_weight << endl;
64    cout << "PARTITION" << endl;
65    for (int i = 0; i < k; ++i) {
66      cout << "Partition " << i << ": ";
67      for(set<int>::iterator it = partition[i].begin(); it !=
            partition[i].end(); ++it) {
68        cout << *it << ' ';
69      }
70      cout << endl;
71    }
72
73    return 0;
74  }
```

7.6. Metaheurística GRASP

```
1  double Grasp::ejecutar(int criterioParada) { // Ejecuta GRASP hasta que se
       llegue al criterio de parada y devuelve el minimo peso
2      cout.precision(4);
3      iteracionActual_ = 0;
4      ultimaIteracionConMejora_ = 0;
5      while( ! parar(criterioParada) ) {
6          iteracionActual_++;
7          particionActual_ = h_.resolver();
8          pesoParticionActual_ = pesoParticion(particionActual_);
9          busquedaLocalUnNodo();
10         if (pesoParticionActual_ < pesoMejorParticion_) {
11             mejorParticion_ = particionActual_;
12             pesoMejorParticion_ = pesoParticionActual_;
13             ultimaIteracionConMejora_ = iteracionActual_;
14         }
15     }
16     return pesoMejorParticion_;
17 }
```

7.7. Generador de instancias

```cpp
typedef int Vertice;

int CANT_INSTANCIAS;
int MIN_VERTICES;
int MAX_VERTICES;
double MAX_COSTO_ARISTA;

Vertice seleccionarVerticeRandom(const set<Vertice> & conjunto) {
    int i = rand() % conjunto.size();
    auto it = conjunto.begin();
    for(int j = 0; j < i; j++) it++;
    return *it;
}

int main(int argc, const char* argv[]) {
    ifstream archivoConfiguracion("configuracionGeneracionInstancias.txt");
    archivoConfiguracion >> CANT_INSTANCIAS >> MIN_VERTICES >> MAX_VERTICES
        >> MAX_COSTO_ARISTA;
    archivoConfiguracion.close();
    srand(time(NULL) + getpid()); // Seedeo
    cout.precision(4);
    for (int n = MIN_VERTICES; n <= MAX_VERTICES; n++) {
        const int m_maximo = n * (n - 1) / 2; // Cantidad de aristas del
            grafo completo de n nodos
        const int m_minimo = 0.7 * m_maximo; // Minima cantidad de aristas
            de los grafos generados
        const int k_minimo = 2; // Minimo k de las instancias generadas
        const int k_maximo = max(k_minimo, n / 3); // Maximo k de las
            instancias generadas
        for (int i = 1; i <= CANT_INSTANCIAS; i++) {
            int m = m_minimo + rand() % (m_maximo - m_minimo + 1); // m es
                un valor aleatorio entre m_minimo y m_maximo
            int k = k_minimo + rand() % (k_maximo - k_minimo + 1); // k es
                un valor aleatorio entre k_minimo y k_maximo
            cout << n << "␣" << m << "␣" << k << endl;
            set<Vertice> vertices; // Vértices que todavía tienen al menos
                una arista disponible
            for (int i = 0; i < n; i++) {
                vertices.insert(i); // Los vertices van a ser enteros entre
                    0 y n-1, tengo que sumarles uno al imprimir
            }
            vector<set<Vertice>> vecinosPosibles(n);
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    if (i != j) {
                        vecinosPosibles[i].insert(j);
                    }
                }
            }
            while (m > 0) {
                Vertice v = seleccionarVerticeRandom(vertices);
                if (vecinosPosibles[v].size() == 0) {
                    vertices.erase(v);
                } else {
                    m--;
                    Vertice w = seleccionarVerticeRandom(vecinosPosibles[v]);
                    vecinosPosibles[v].erase(w);
                    vecinosPosibles[w].erase(v);
                    double weight = MAX_COSTO_ARISTA *
                        (static_cast<double>(rand()) /
                        static_cast<double>(RAND_MAX));
```

```
52                      if (weight == 0.f) { // 0.f es equivalente a no tener
                            arista, hay que arreglarlo
53                          weight += 0.0001; // Esto sólo tiene sentido para
                                cout.precision(r) con r >= 4
54                      }
55                      cout << v + 1 << "␣" << w + 1 << "␣" << fixed << weight
                            << endl;
56                  }
57              }
58          }
59      }
60      return 0;
61  }
```