DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Prctico 2

Informe y análisis de resultados.

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Catriel Omar D'Ela	964/11	catriel.delia@gmail.com
Ignacio Niesz	722/10	ignacio.niesz@gmail.com
Julin Osas Jamardo	769/08	jamardo.julian@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

Índice

1. Introducción

En el presente trabajo intentaremos resolver mediante distintas técnics algorítmicas el problema de la *k-Partición de Mínimo Peso* (**k-PMP**).

Primero es necesario introducir la idea de partición de conjuntos. Una partición de un conjunto A consta de varios de subconjuntos no vacíos de A disjuntos, tales que la unión de todos ellos es A. Es decir, sea $P = \{A_i : iI\}$, se cumple:

- Para cada $i \in I$, $A_i \subseteq A$ y $A_i \neq \emptyset$
- Para cada par $i \neq j, A_i \cap A_j = \emptyset$
- $\bigcup_{i \in I} A_i = A$

Además, una *k-partición*, es una partición con k subconjuntos.

Esta noción se puede trasladar a grafos. Un grafo G se representa con dos conjuntos V y W tales que V, corresponde a los vértices y W a las aristas, una partición de G consiste en particionar el conjunto de vértices V, bajo la definición de partición de conjuntos, por ejemplo, $G = (V, W)$ con $V = \{1, 2, 3\}$ es decir, el grafo con los vértices 1 2 y 3, las posibles particiones de G son $\{\{1\}, \{2, 3\}\}, \{\{1, 2\}, \{3\}\}, \{\{1, 3\}, \{2\}\}, \{\{1\}, \{2\}, \{3\}\}$ y todo el conjunto $\{1, 2, 3\}$. Cabe aclarar que en el desarrollo del trabajo, tomamos al conjunto vacío como un posible subconjunto perteneciente a una partición, por ejemplo $\{1, 2, 3\}, \{\}$ es válida como partición de G, por lo que la única propiedad de la definición de Partición de conjuntos que no se cumple es la primera.

Al particionar un grafo, podemos identificar dos tipos de ejes. Las *aristas intrapartición* que son aquellas tal que ambos extremos pertenecen a un mismo subconjunto de una partición o dicho formalmente, sea $V_1, V_2, ..., V_k$ una partición de V una arista $uv \in W$ es intrapartición si existe un $i \in \{1, ..., k\}$, tal que $u, v \in V_i$. De la misma manera, podemos diferenciar las aristas que no corresponden a ninguna partición como aquellas tales que sus extremos pertenecen a subconjuntos distintos.

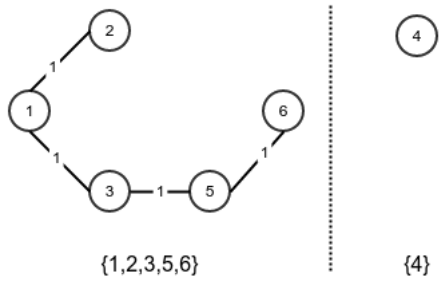
Para ejemplificar el problema gráficamente, la figura **Figura 1b** es una *2-partición* del grafo de la **Figura 1a**. Las aristas $(1, 2)$ y $(3, 4)$ no son intrapartición ya que los vértices de sus extremos quedaron en distintas particiones, mientras que las aristas $(1, 3)$, $(3, 5)$, $(5, 6)$ y $(2, 4)$ si lo son.

(a) Grafo original

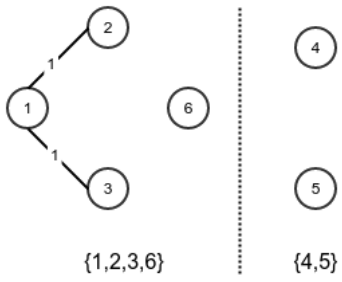
(b) Partición

Figura 1

Volviendo al problema **k-PMP**, dada una función de peso $\omega : E \to R+$, definida sobre las aristas de G, el peso de una k-partición es la suma de los pesos de las aristas intrapartición, por lo que el objetivo es hallar la de mínimo peso con respecto a cualquier otra k-partición. Por ejemplo en la **Figura 1b** la suma de las aristas intrapartición es 103 mientras que si particionamos dejando los extremos de la arista con peso 100 en distintas particiones como muestra la **Figura 2a** logramos un peso total de 4 y más aún, si particionamos como en la **Figura 2b** logramos un peso total de 2.



(a) Partición con peso total 4



(b) Partición con peso total 2

Figura 2

Es importante notar que la partición de la **Figura 2b** es una k-PMP, pero no es la única, ya que existen otras como la $\{2, 3, 5, 6\}, \{4, 1\}$ también con peso 2.

El problema *k-Partición de Mínimo Peso* o *Minimum K-Cut* (en Inglés) es un problema **NP-Completo** [1] por lo que no se conoce un algoritmo polinomial que lo resuelva, es por esto que a lo largo del trabajo, vamos a dar una algoritmo basado en la técnica de **backtracking** para obtener una de las soluciones "exactas pero de gran complejidad computacional, para luego explorar la **Heurística Golosa**, la **Búsqueda Local** y por último **GRASP**, que nos ayudaran a acercarnos a la solución lo más posible con ordenes de complejidad más bajos.

2. Problema k-PMP

El problema k-PMP se reduce en encontrar todas las posibles k-Particiones del conjunto de vértices de un grafo y de ellas, obtener las de menor suma total de aristas intrapartición. Hallar las particiones de un conjunto ya fue resuelto previamente en otro trabajo práctico de la cátedra y se puede relacionar con otros problemas de igual complejidad.

2.1. k-PMP y el problema 3 del TP1

El ejercicio 3 del TP1, consistía en ubicar productos en camiones, teniendo en cuenta cierta peligrosidad entre cada par de productos y un umbral de peligrosidad propio de cada camión. El objetivo era minimizar la cantidad de camiones, por lo que era necesario poner los productos de manera que las peligrosidades de a pares fueran mínimas y de esa forma, colocar la mayor cantidad de productos en cada camión.

Para resolverlo se utilizó la técnica de **Backtracking** y la idea básicamente era que el primer producto se podía ubicar en el primer camíon, el segundo, en el primer camión o en su defecto crear uno nuevo, el tercero, en los camiones ya utilizados, (uno o dos) o colocar uno nuevo, etc. Se demostró que para n productos, cada hoja del árbol de backtracking era una posible partición de un conjunto de n elementos, pues nos podmeos abstraer de los productos y los camiones, y hablar de elementos y conjuntos, entonces el razonamiento se convierte en ubicar el primer elemento en un subconjunto nuevo, el segundo en el subconjunto que ya teníamos o en un segundo conjunto, el tercero en los conjuntos ya utilizados o uno nuevo, etc. El árbol resultante tenía la pinta de la **Figura 3**.

[1] Garey M.R. and Johnson D.S., Çomputers and intractability: a guide to the theory of NP- Completeness", W. Freeman and Co., 1979. (a),(c)

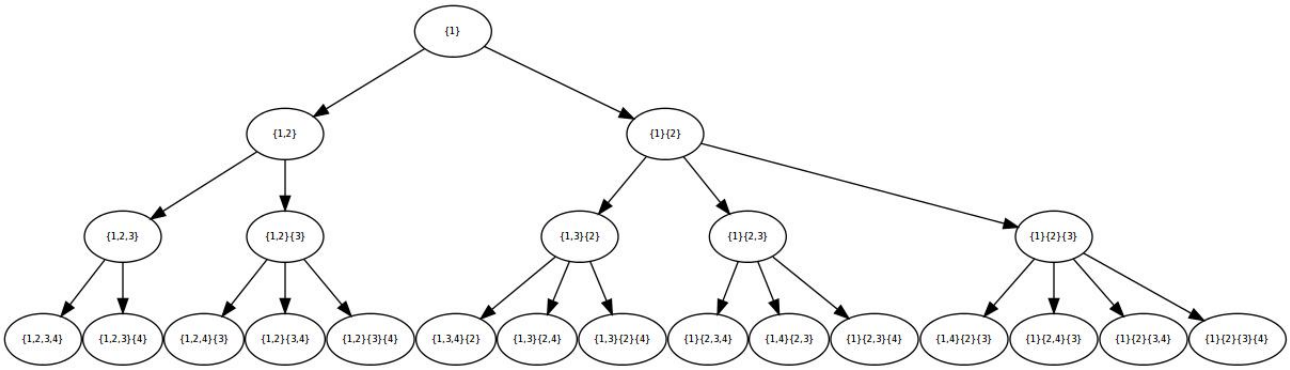


Figura 3: Árbol de combinaciones para tres elementos

Si queremos hallar la mejor solución para el problema k-PNP debemos explorar todas las posibles k-Particiones y esto es un subconjunto de todas las particiones exploradas por el algoritmo de backtracking del Ejercicio 3 del TP1, por lo que sólo es necesario evitar las particiones que no nos interesan, es decir, aquellas con más de k conjuntos o aquellas con menos, aunque esto último es simplemente una decisión de diseño explicada más adelante.

2.2. k-PMP y el problema de coloreo de vértices de un grafo

Qué sucede si el peso de una k-PMP es cero? En tal caso tendríamos una k-partición tal que no existen aristas intrapartición, es decir, obtenemos k conjuntos independientes de vértices.

Si a cada conjunto se le asigna un color obtenemos un coloreo válido para el grafo (esto sucede siempre y cuando el grafo sea ponderado con aristas de peso positivo mayor estricto que cero). Si el peso de la k-Partición no es cero, existe alguna arista tal que dos vértices del mismo conjunto son adyacentes, lo que imposibilita colorear.

Por otro lado, sabemos que si la solución del problema k-PMP no es cero para un cierto k, tampoco lo va a ser para $k' < k$ pues si lo fuera, podríamos partir de la solución de $k' - PMP$ y reducir el peso agregándole un conjunto nuevo, sacando al menos un vértice que tenga un adyacente el mismo conjuntos de los k' calculados.

Además si $k = |V|$ utilizando un conjunto para cada vértice la suma sería cero. Por lo tanto un algoritmo para resolver coloreo sería resolver k-PMP para k=1, luego para k=2, así hasta llegar a un k-PMP donde el peso sea cero.

En cuanto a la complejidad, sería n por la complejidad del algoritmo óptimo de k-PMP.

Es sabido que encontrar el mínimo k para colorear un grafo es un problema NP-Completo, por lo que se reafirma el hecho de que k-PMP también lo es, porque si no lo fuera, coloreo se podría resolver con el algoritmo antes mencionado y no sería NP-C.

2.3. k-PMP y modelado de problemas reales

Este modelo matemático puede ser aplicado a diferentes situaciones en donde se disponen de recursos y se busca la mejor manera de que trabajen juntos.

Por ejemplo en la computacin distribuida, cada vrtice puede ser una computadora y cada arista la velocidad de la conexin entre un par de ellas.

Si el problema admite ser dividido en K subproblemas independientes entre si, entonces se puede buscar K conjuntos de computadoras para que trabajen en simultaneo y ataquen cada uno de estos K subproblemas minimizando la prdida producida por la red. Como los mismos son independientes las computadoras de las distintas particiones no necesitan comunicarse entre si hasta terminar el problema y las comunicaciones intra particin son ptimas.

Para el caso anterior habra que restringuir la no existencia de aristas para evitar computadoras no conectadas en alguna particin.

Otro grupo de problemas para los cuales sirve son los de cableado, donde es deseable eliminar las conexiones de mayor costo. Un ejemplo de esto es VLSI (Very Large Scale Integration).

Tambin tienen relacin con tareas de secuenciamiento de trabajos, de una forma similar al ejemplo de computo en paralelo, lo que se puede buscar mediante un modelado en grafo y un algoritmo que resuelva k-PMP es que tareas o trabajos no conviene hacer en simultaneo. Cada particin tendra entonces las tareas que pueden correr en simultaneo con un menor costo global.

3. Algoritmo exacto

A continuación presentamos un algoritmo exacto para el problema k-PMP. Como se explicó en puntos anteriores, está basado en la técnica de backtracking que explora todas las particiones de un conjunto.

3.1. Algoritmo

Algorithm 1: Backtrack

```
   Data: P, Ps, G = (V, E), k, v, verticesPorUbicar, K, particionesLibres
   Result: Ps
 1 if verticesPorUbicar = 0 then
 2     if ω(P, G) < pesoMin then
 3         Ps = P;
 4         pesoMin = ω(P, G);
 5 j = 0;
 6 if verticesPorUbicar ≤ particionesLibres then
 7     j = K − particionesLibres;
 8 for p = p_j, ..., p_k ∈ P do
 9     s_k = k;
10     if p = ∅ then
11         particionesLibres − −;
12         if k < K then
13             s_k + +;
14     if ω(P, G) > pesoMin then
15         return
16     p ⋃ {v};
17     verticesPorUbicar − −;
18     Backtrack(P, Ps, G, s_k, v + 1, verticesPorUbicar, K, particionesLibres);
19     verticesPorUbicar + +;
20     p − {v};
21     if p = ∅ then
22         particionesLibres + +;
```

La función resuelve el problema de ubicar el nodo v pasado por parámetro en uno de los K subconjuntos (ciclo principal del algoritmo). En cada iteración colado v en un subconjunto y hace un llamado recursivo con el próximo nodo a ubicar, cuando retorna del llamado recursivo saca el nodo del subconjunto y vuelve a iterar para colocarlo en otro y volver a hacer la recursión (líneas 16 a 20).

- P es el conjunto de los k subconjuntos, es decir la k-Partición.
- K es la totalidad de los subconjuntos que queremos calcular, mientras que k es el máximo número de subconjuntos en los que es posible ubicar el vértice v, que crece a medida que vamos ubicando nodos.
- ω es la función que toma la K-Partición P y un Grafo y calcula la suma de todas las aristas intrapartición.
- $pesoMin$ corresponde al ω de la solución parcial (K-Partición) de menor peso.

Las líneas 1 a 4 corresponden al "paso base" del algoritmo, es decir a una hoja del árbol. Si la solución obtenida en esa rama luego de haber ubicado todos los nodos en distintas particiones, es menor que la mejor solución parcial guardada, la actualizamos.

3.1.1. Podas

En este algoritmo no nos interesa explorar ramas que tengan soluciones con conjuntos vacíos, pues de existir podemos tomar nodos de otros subconjuntos y colocarlos en estos sin afectar el peso total o incluso mejorándolo. Si la cantidad de vértices por ubicar es igual a la cantidad de subconjuntos vacíos, el algoritmo los completará todos ubicando un vértice en cada uno (líneas 5 a 7). Esto corta las ramas de aquellas soluciones que contengan conjuntos vacíos.

Por otro lado, no queremos calcular particiones con más de K subconjuntos por lo que las líneas 12 y 13 se encargan de aumentar el siguiente valor máximo de conjuntos a probar (s_k) sólo si no se superaron los k conjuntos buscados.

Por último es lógico no continuar si la suma de las aristas intrapartición de la solución parcial obtenida hasta cierto momento es mayor que la que ya obtuvimos en alguna otra rama (líneas 14 y 15).

3.1.2. Análisis de complejidad

Como comentamos el anteriormente, para el primer nodo v_1, tenemos sólo un conjunto, llamemoslo c_1, para ubicarlo. Para v_2, el c_1 o uno nuevo, el c_2. Para el tercer nodo v_3, si ubicamos el v_1 en c_1 y el v_2 en c_1, podemos colocarlo también en c_1 o bien crear un nuevo subconjunto c_2 (notar que este c_2 no es el mismo que el anterior), pero si ubicamos v_1 en c_1 y v_2 en c_2, entonces tenemos esos dos subconjuntos o crear un conjunto c_3.

As sucesivamente hasta llegar a k vértices. Luego, para v_1 hay 1 posibilidad, para v_2 hay 2, para v_3 hay 3, hasta v_k con k y eso es $k!$.

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

Por lo tanto, la complejidad del algoritmo es $O(k! * k^{(n-k)})$ en el peor caso, sin tener en cuenta para el cálculo las podas mencionadas anteriormente. Notar que todas las demás operaciones se realizan en $O(1)$.

3.2. Experimentación

Para hacer la experimentación tuvimos en cuenta k (la cantidad de subconjuntos), la cantidad de vértices (que no pudo superar los 16 por ser muy lento) y la densidad del grafo. Esta última está representado como un valor entre 0 y 1 que corresponde a la probabilidad de que exista una arista entre dos vértices, por lo que un grafo de densidad 0.20 contiene considerablemente menos aristas que uno de 0.8, sien el grafo de densidad 1, el completo.

Para empezar analizaremos el comportamiento para un mismo k, con diferentes densidades. Todos los gráficos a continuación se presentan en escala logarítmica en el eje del tiempo Y.
En la **Figura 4** vemos los resultados para $k = 1$. Podemos apreciar como se comporta de manera similar para las distintas densidades, esto parece bastante claro, ya que sin importar el peso de las aristas, es necesario colocar a todos los vértices en un mismo conjunto, lo que realiza de manera lineal pues no tiene más que una opción por vértice.

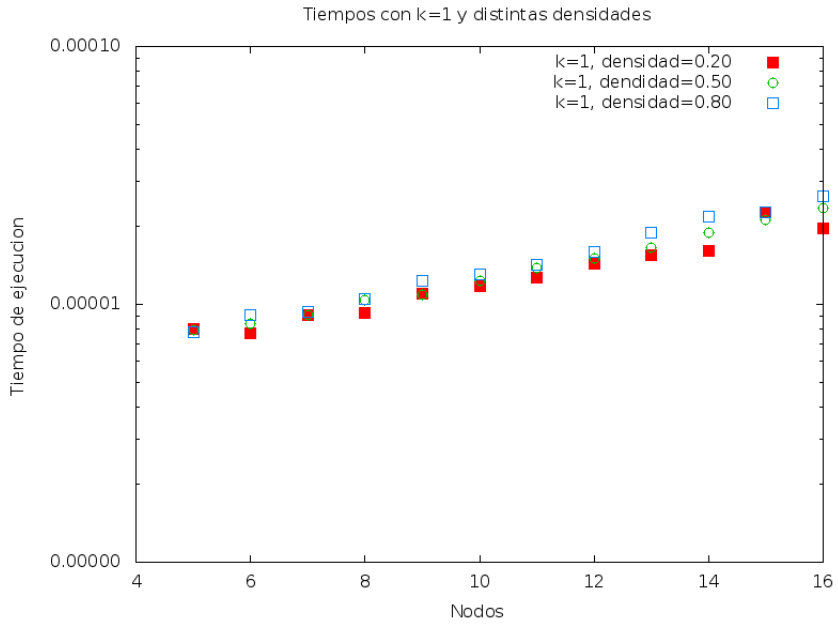


Figura 4: Comparación de k=1 con distintas densidades

En el caso de $k = 2$ **Figura 5**, podemos ver que el algoritmo se comporta mejor con grafos de densidad más baja. Esto puede tener relación con el hecho de que al haber sólo dos posibilidades por vértices y pocas aristas, es posible que en una rama lleguemos a una solución lo bastante buena como para podar muchas ramas.

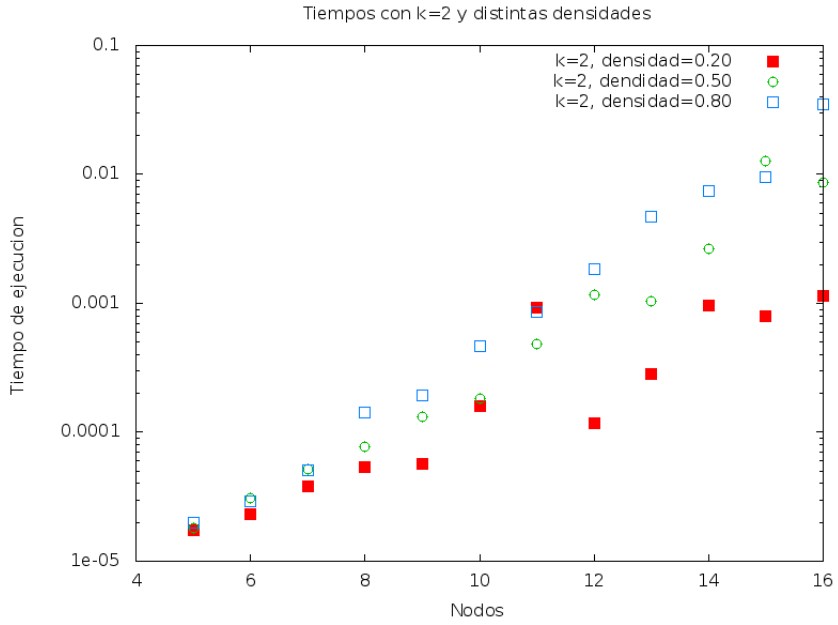


Figura 5: Comparación de k=2 con distintas densidades

Si bien con $k = 2$ parece que mejora con grafos poco densos, en la **Figura 6** y **7** notamos como se va revirtiendo este comportamiento a medida que aumenta el k, hasta incluso, como se ve en la **Figura 8** parece que para 0.50 y 0.80 casi que obtenemos resultados similares.

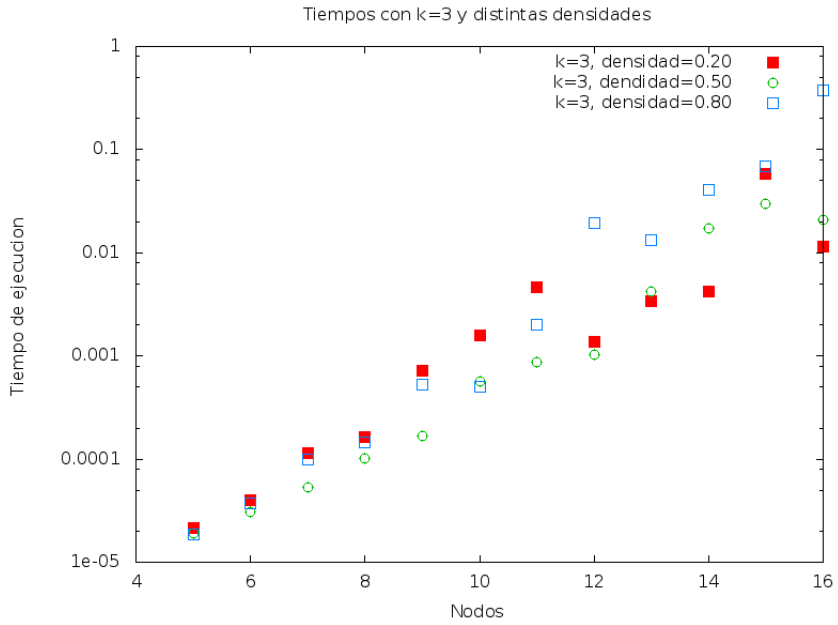


Figura 6: Comparación de k=3 con distintas densidades

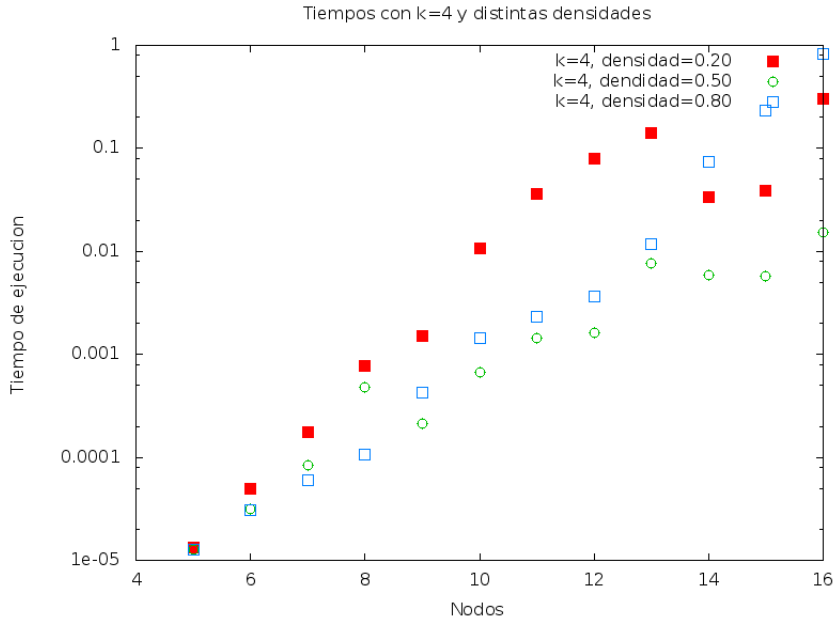


Figura 7: Comparación de k=4 con distintas densidades

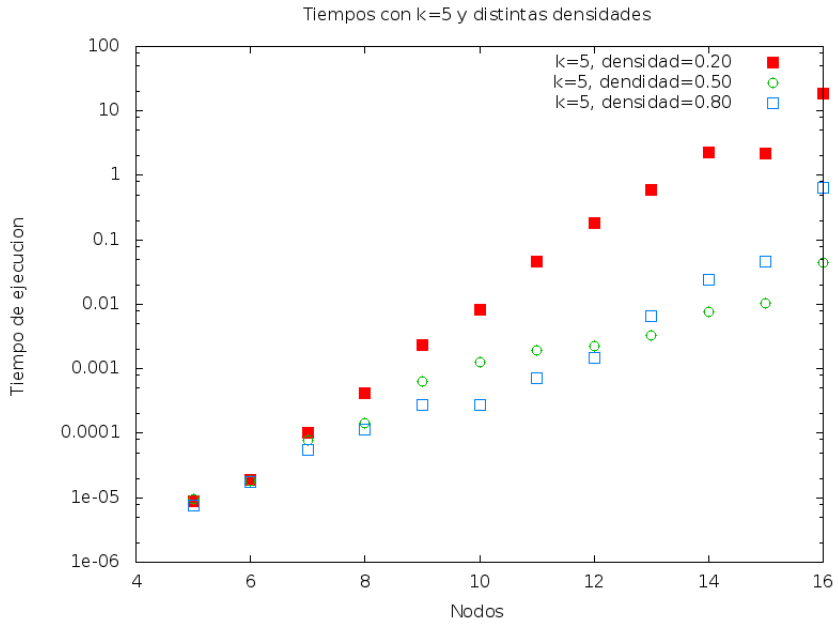


Figura 8: Comparación de k=5 con distintas densidades

Estos resultados son anti intuitivos, pues uno creería que a menor densidad, mayor posibilidad de encontrar una solución buena y que con más aristas mayor la necesidad de explorar el árbol. Esto puede estar sujeto al hecho de que a mayor cantidad de aristas, la probabilidad de cortar ramas aumenta, pues si la solución encontrada es de un peso considerablemente bajo, vamos a cortar en todas aquellas ramas que lo superen, esto siempre y cuando nos encontremos con un grafo con peso uniforme en las aristas y no con casos patológicos como aquel con todos pesos iguales, que nos obliga a recorrer el árbol entero.
Otro factor interesante que modifica el comportamiento del algoritmo es cuan cerca está k de de n, por ejemplo, no siempre es cierto que el número de 5-Particiones de un conjunto es mayor que el número obtenido por un $k < 5$ ya que si el conjunto tiene 6 elementos, hay 15 posibles 5-Particiones mientras que para $k = 3$ hay 90. Este número si bien no se mencionó con anterioridad corresponde el *Número de Stirling de Segunda Clase* y cada nivel del árbol de backtracking contiene la cantidad de k-particiones posibles para esa cantidad de nodos. Por último, para mostrar que el algoritmo se comporta como lo indica la complejidad teórica, utilizamos k=5 y vértices del 5 al 16.

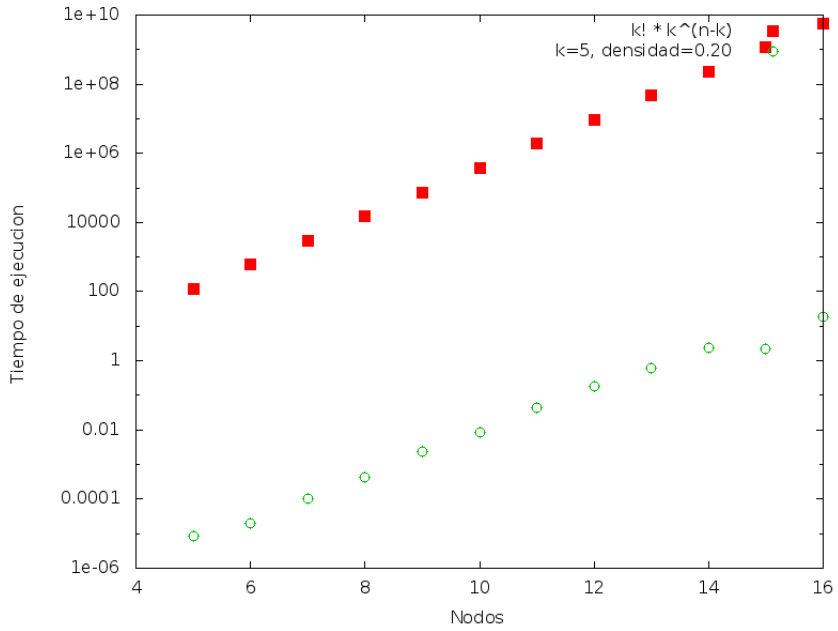


Figura 9: Comparación complejidad teórica contra las corridas sin poda para k=5

Como se aprecia en la **Figura 9** el algoritmo se comporta de manera similar para un caso en que no le es posible aplicar la poda

4. Heurística constructiva golosa

Para la heurística constructiva golosa tenemos en cuenta como cada "paso" de la heuística el colocar un nodo del grafo en alguna de las k particiones, como elección candidata cada una de las k particiones donde puede colocarse el nodo y como "solución local" de cada paso el conjunto que tiene por resultado el peso mínimo.

Consideremos la siguiente instancia del problema como ejemplo, con $k = 2$:

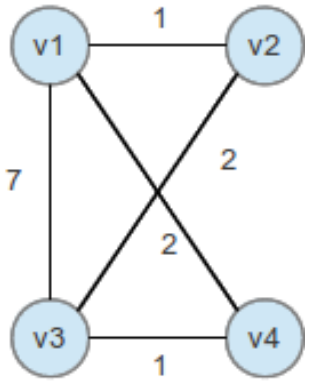


Figura 10: Instanca de un problema 2-PMP

(a) Estado de la solución luego de las dos primeras elecciones.

(b) Estado de la solución luego de la tercera elección.

Figura 11: Ejecución del algoritmo greedy paso por paso.

Como puede observarse dado un grafo el algoritmo considera un posible orden de sus vertices $v_1, \ldots, v_n$ (notemos que la elección de este órden es arbitraria y depende de la implementación) y comienza a colocarlos en cada una de las k particiones del problema de forma secuencial, intentando minimizar el peso en cada elección. En cada paso de la generación de la solución final la elección de ese paso solo tiene en cuenta las aristas que unen al nuevo nodo que todavía no forma parte de la solución con los nodos que fueron agregados a la solución en el paso anterior. Es inmediato observar que utilizando este algoritmo se pierde de analizar posibles combinaciones de nodos en las particiones por lo que podría no alcanzarse la solución óptima.

En el ejemplo anterior:

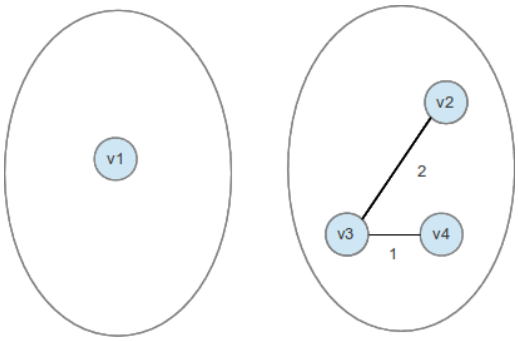


Figura 12: Solución obtenida mediante el algoritmo goloso, $\omega = 3$.

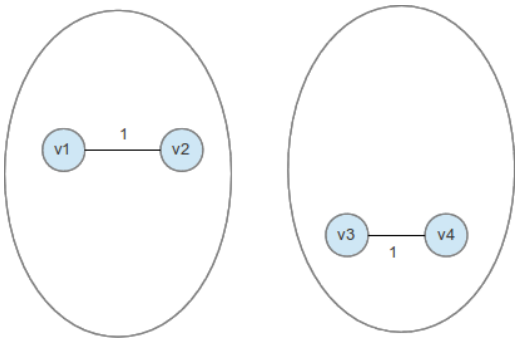


Figura 13: Solución óptima del problema, $\omega = 2$.

4.1. Algoritmo

Algorithm 2: Heurística constructiva golosa k-PMP

Data: $G = (V, E)$, $k : int$
Result: $particiones : V_1, \ldots, V_k$, $weight : int$
1 $V_1, \ldots, V_k = \emptyset$;
2 **for** $v = v_1, \ldots, v_n \in V$ **do**
3 $\quad i$: Tal que $\omega(V_i) = min(\{\omega(V_j \bigcup \{v\}) \ / \ 1 \leq j \leq k\})$;
4 $\quad V_i = V_i \bigcup \{v\}$;
5 **return** $< V_1, \ldots, V_k, \omega(V_1) + \ldots + \omega(V_k) >$

4.1.1. Análisis de complejidad

Observemos que el ciclo de la línea 2 realiza una iteración por cada uno de los nodos del grafo, teniendo complejidad $O(n)$ por la complejidad del cuerpo del ciclo. La línea 4 puede realizarse en tiempo $O(1)$ (dependiendo de la estructura utilizada para representar las particiones). Por último la l ínea 3 tiene cálcular el mínimo de la suma de los pesos de las aristas, de agregar el nuevo nodo a cada una de loas particiones. Esto no es más que realizar la suma del nuevo nodo contra las aristas que forman parte de cada una de las particiones y puede hacerse en tiempo $O(km) = O(kn^2)$.
En definitiva sea $T(n)$ la función de costo del algoritmo goloso, $T(n) \in O(kn^3)$.

4.1.2. Análisis de las soluciones obtenidas

Volviendo a la observacion que realizamos antes, un problema del algoritmo goloso es que no tiene en cuenta al momento de tomar decisiones las aristas de los nodos que todavía no forman parte de la solución.

Observemos el siguiente caso con $2 \leq k \leq 4$:

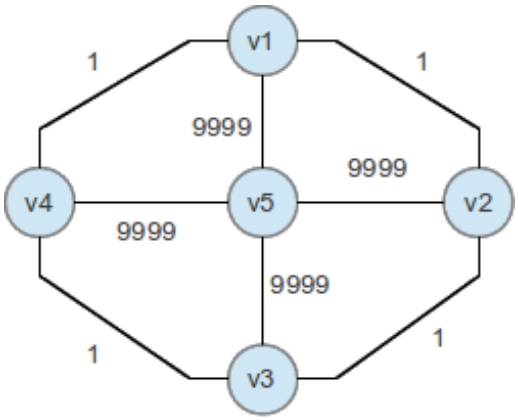
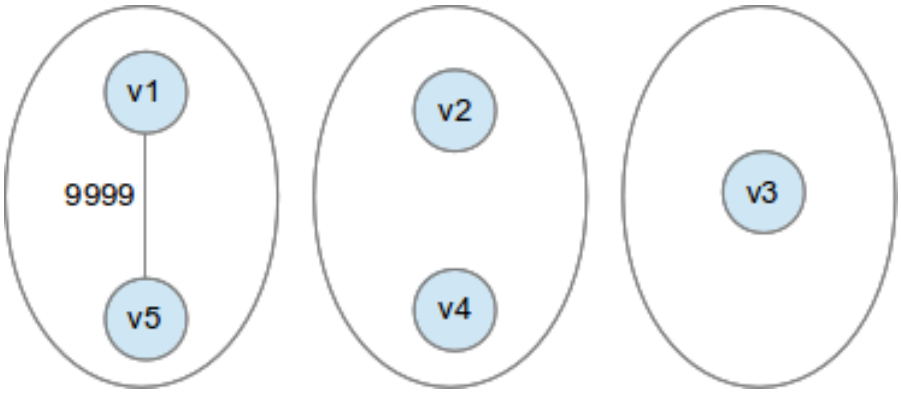
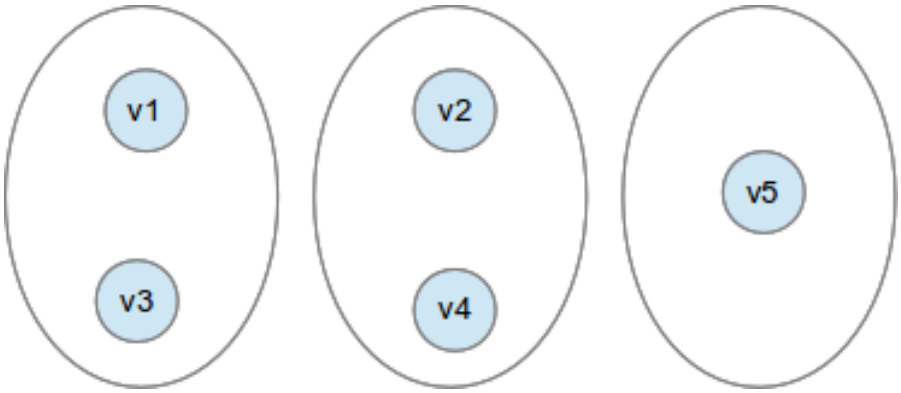


Figura 14: Instancia "estrella + ciclo".

Es inmediato ver que si el algoritmo procesa secuencialmente los nodos $v_1, \ldots, v_5$, los 4 primeros nodos quedaran distribuidos entre las k particiones y el nodo v_5 arruinara el resultado con alguna de sus aristas de peso 9999. Para llegar a la solución óptima es aislar al nodo del centro en una partición y luego distribuir el resto en las particiones restantes.



(a) Solución obtenida mediante el algoritmo goloso, $\omega(S) = 9999$.



(b) Solución óptima del problema, $\omega(S) = 0$.

Figura 15: Ejecución del algoritmo greedy paso por paso.

Esta idea puede generalizarse demostrando que el resultado propuesto por la herística golosa puede ser arbitrariamente malo dependiendo del grafo. Cualquier grafo que tenga una componente "ciclo simple de tamaño $n-1$" $\bigcup$ "estrella de tamaño n", donde el peso de las aristas de la estrella $L > \omega(e_1) + \ldots + \omega(e_{n-1})$, es decir mayor a la suma de los pesos de todas las aristas del ciclo. De esta manera aun tratando una instancia del problema k-PMP donde $k = 2$ resulta conveniente para minimizar el peso de la componente aislar el nodo de la estrella y agrupar todos los nodos del ciclo en una misma partición.
Debido a la implementación de la heurística golosa que utiliza órden lexicografico, si el nodo central de la estrella es el último en ser considerado, nunca se tendran en cuenta las aristas más pesadas para decidir aislar este nodo.

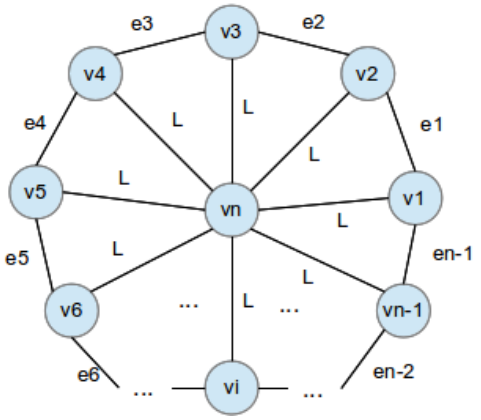
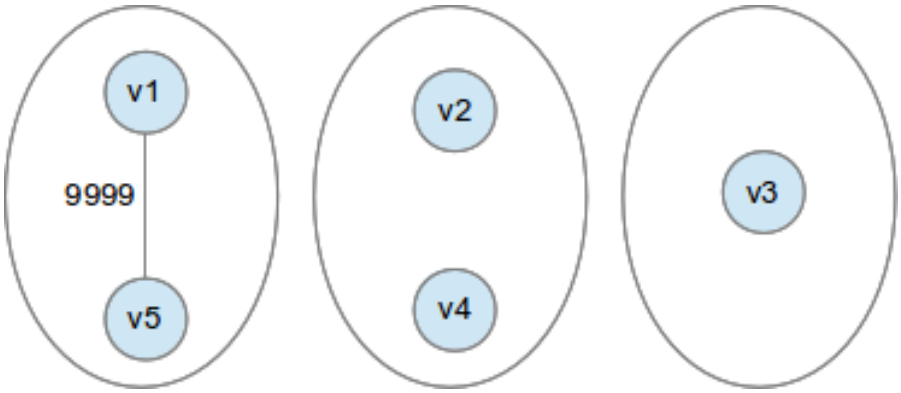


Figura 16: Instancia "estrella + ciclo" de tamao n.

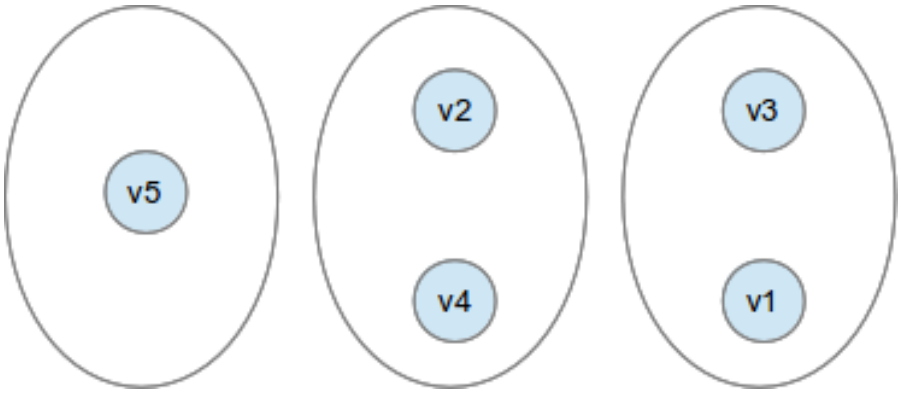
4.2. Experimentación

5. Heurística de búsqueda local

Volviendo al ejemplo anterior para mostrar que el algoritmo goloso puede ser arbitrariamente malo, podemos observar que una mala solución del algoritmo goloso puede "arreglarse" con relativamente bajo costo. Partiendo de la solución a la instancia "ciclo + estrella de tamaño 5" del algoritmo goloso, basta con mover el nodo que se encuentra compartiendo una partición con el nodo del centro de la estrella. Esto puede conseguirse simplemente revisando todos los nodos nuevamente y viendo si se mejora la solución moviendolos a otra partición.



(a) Solución obtenida mediante el algoritmo goloso, $\omega(S) = 9999$.



(b) Solución óptima del problema, $\omega(S) = 0$.

Figura 17: Comparación entre la solución del algoritmo goloso y la óptima del problema.

Cualquier solución del algoritmo goloso que se obtenga de una instancia que contenga una componente con la pinta "ciclo + estrella de tamaño n" puede "arreglarse" de esta forma. Notemos que como peor caso una solución de esta instancia tendra varios nodos del ciclo en la misma partición que los del centro de la estrella, luego esa solución puede mejorarse por otra que mueva cualquiera de esos nodos a otra partición donde se encuentren otros nodos del ciclo. Esta idea puede generalizarse para dar órigen a nuestra heurística de busqueda local, tomando una solución cualquiera (en nuestro caso generada a partir del goloso) y definiendo como vecindad de esa solución todas las solución resultantes de mover un nodo de una partición a otra.
Tiene sentido preguntarse si sera cierto que toda solución puede mejorarse de esta manera. Notemos que si esto fuera cierto y cualquier solución pudiera mejorarse de esta forma, podría alcanzarse la solución óptima desde cualquier solución en tiempo polinomial simplemente realizando sucesivas busquedas locales. El problema radica en que no toda solución puede "arreglarse" moviendo un único nodo de una partición a otra.

Observemos el siguiente ejemplo:

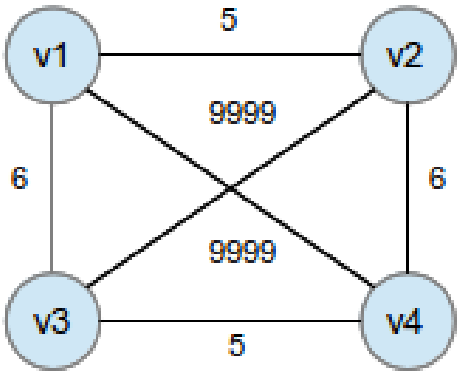
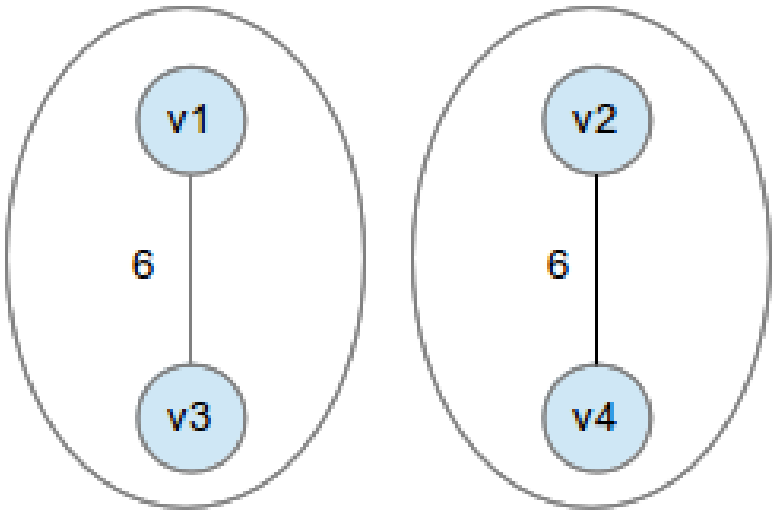
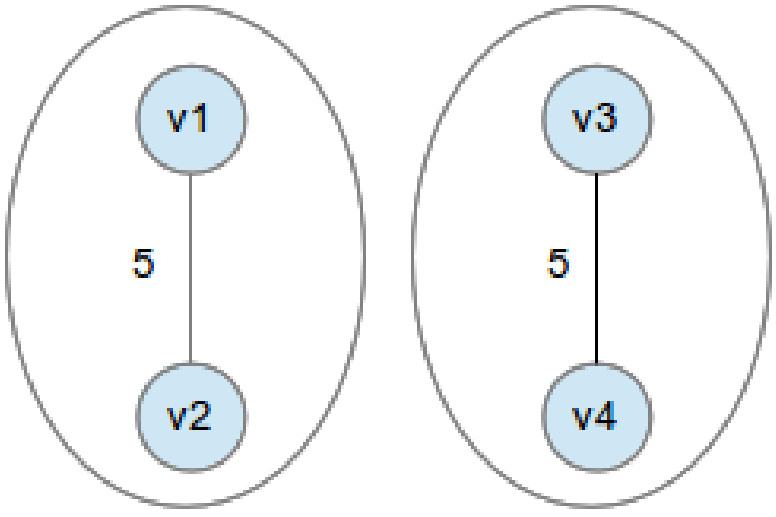


Figura 18: Instancia de un problema 2-PMP.



(a) Solución obtenida mediante el algoritmo goloso, $\omega(S) = 12$.



(b) Solución óptima del problema, $\omega(S) = 10$.

Figura 19: Ejecución del algoritmo greedy paso por paso.

En este ejemplo en particular, esta claro que todas las soluciones vecinas a la solución propuesta por el goloso son se alejan bastante del óptimo (de hecho arbitrariamente según el peso que asignemos a las aristas diagonales). Vamos a exhibir una familia de grafos donde la vecindad de la solución obtenida por el algoritmo goloso, no puede mejorarse moviendo un único nodo a otra partición y esta solución puede alejarse del óptimo tanto como queramos.

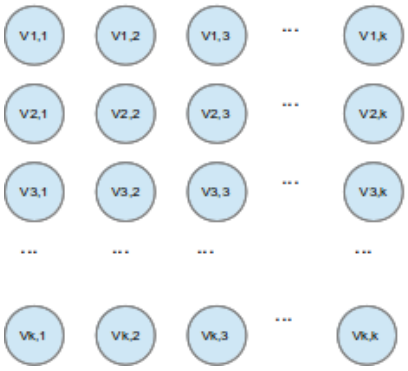
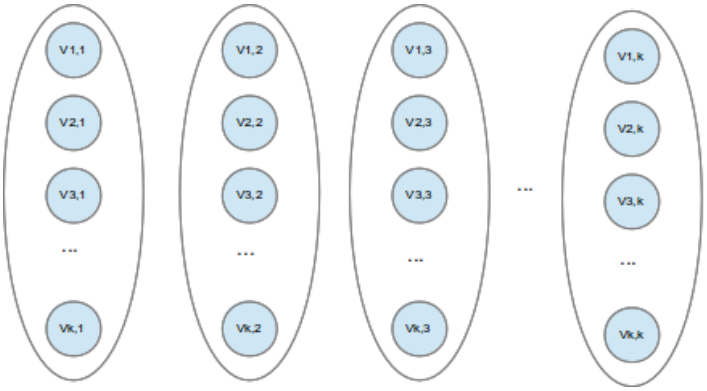
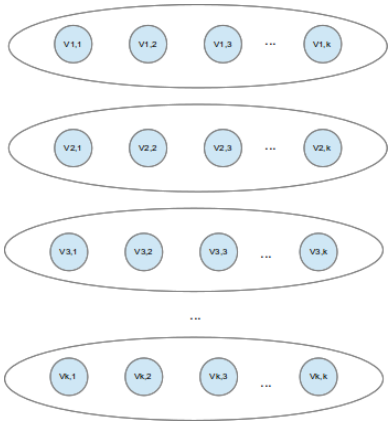


Figura 20: Instancia de un problema 2-PMP.

Pensando la siguiente familia de grafos como una matriz, supongamos que existe una arista que une cada par de nodos $(v_{i,i}, v_{i,i+1})$ horizontalmente de forma tal que el óptimo se alcanza agrupando los nodos horizontalmente en k particiones. Ademas existe una arista entre todos par de nodos que no pertenezcan a la misma partición (columna) ni a la misma fila y estas aristas son más pesadas que la suma de todas las aristas que existan entre nodos de una misma columna. De esta forma, siguiendo un órden lexicografico que recorre por filas y luego por columnas los resultados tendrían la siguiente pinta:



(a) Solución obtenida mediante el algoritmo goloso, $\omega(S) = L_1$.



(b) Solución óptima del problema, $\omega(S) = L_2$.

Figura 21: Ejecución del algoritmo greedy paso por paso.

Como existe una arista entre todo nodo que no pertenezca a la misma fila o la misma columna más pesada que la suma de todas las aristas de la misma columna, nunca puede mejorarse la solución moviendo un nodo de una partición a otra, ya que el peso aumentaria por las aristas que conectan al nodo con el resto de la

partición. Sin embargo, las aristas horizontales entre todo par de nodos son las menos pesadas por lo que no se alcanza el optimo. Esta diferencia ademas puede ser tan mala como queramos.

5.1. Algoritmo

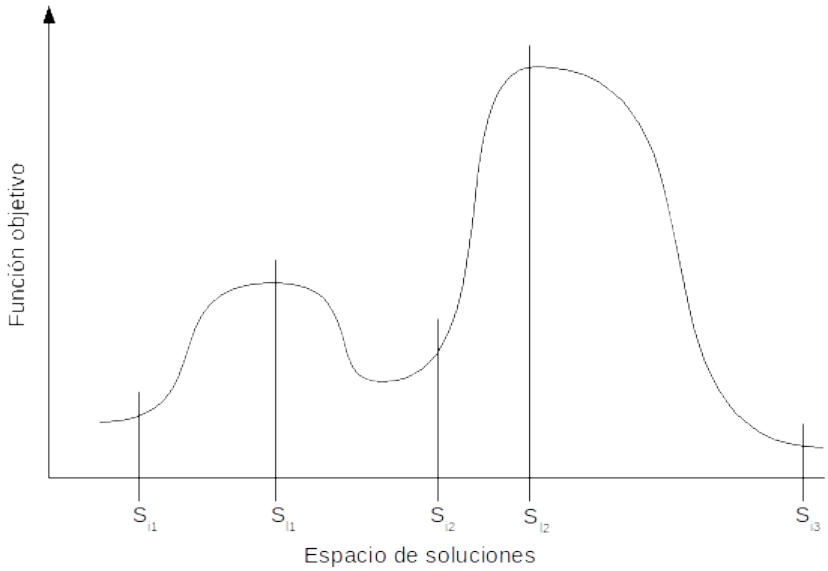
5.1.1. Análisis de complejidad

5.2. Experimentación

6. Metaheurística GRASP

La metaheurística GRASP utiliza los dos algoritmos anteriormente explicados combinados de tal manera de combinar lo mejor de ambos mundos.

Las deficiencias, en términos generales, que tiene la búsqueda local es que depende mucho de la semilla o la solución general.



(a) Problemas con la busqueda local

en la figura de arriba podemos apreciar que de empezar por s_{i1} se llegara a una solucin s_{l1}, pero nunca a la s_{l2} que es la mejor de las dos.

Para evitar esto se puede empezar por alguna solucin random, pero esto tiene el problema que es poco problable que la solucin sea buena. Por ejemplo s_{i3} está más lejos

ñ

6.1. Algoritmo

6.2. Experimentación

7. Comparación de tiempos y cálidad de soluciones