# 1. Algoritmo exacto

## 1.1. Desarrollo

El problema que se nos presenta es el de la k-PMP. El mismo trata sobre, dado un grafo G=(V,E) con pesos en las aristas, encontrar una particion de los nodos en k (o menos) conjuntos, tal que sea la partición de menor peso. Lo que hace al peso de una partición, es la suma de los pesos de las aristas intrapartición (sin importar aquellas que no lo son). Una arista se dice *intrapartición* cuando los 2 nodos sobre los que incide pertenecen al mismo conjunto de partición.

Dicho esto, para la realización del algoritmo exacto, optamos por revisar todas las particiones posibles de los n nodos, y quedarnos con la mejor (la de menor peso) de las que tengan k o menos conjuntos. Se elaboraron podas y estrategias previas al analisis de todas las posibles particiones para mejorar los tiempos de ejecución, y mejorar la complejidad espacial.

El algoritmo ***sin podas ni estrategias*** para generar las particiones de un conjunto de n nodos es el siguiente:

Miremos las particiones de un conjutno de 1 elemento, 2 elementos, y 3:

| n=1 | n=2 | n=3 |
|---|---|---|
| [1] | [1] [2] | [1] [2] [3] |
| | [1,2] | [1,2] [3] |
| | | [3,2] [1] |
| | | [3,1] [2] |
| | | [1,2,3] |

Notamos la siguiente similitud entre la transición de uno a otro (ejemplo, transición de 2 a 3):

| n=1 | n=2 | n=3 |
|---|---|---|
| [1] | [1] [2] | [1] [2] [3] |
| | [1,2] | [1,2] [3] |
| | | [1,3] [2] |
| | | [1] [2,3] |
| | | [1,2,3] |

Para generar las particiones un conjunto de 3 elementos, a partir de las particiones de un conjunto de 2 elementos, hace 2 pasos:
- Toma las de 2 elementos y le agrega un nuevo conjunto que solo contiene al n (En el gráfico, las 2 primeras particiones de las listadas para n = 3), ese nuevo grupo de particiones ya formará parte de las particiones finales de 3 elementos.
Esta cantidad de particiones añadidas cumple que #particionesAñadidas = #particionesTotalesDeN-1

-Luego, vuelve a mirar las particiones de n-1, y por cada una (llamemosle X) realizará lo siguiente:
Añadira una partición por cada conjunto de X, y en cada una de ellas, agregará el valor de n a un conjunto distinto.

**Ej:** Toma la partición [**1**] [**2**]:
Aplica el **3** al conjunto [**1**], resultando en el conjunto: [**1,3**] [**2**]
Y la agrega a las particiones de 3.
Luego aplica el **3** al conjunto [**2**], resultando en el conjunto: [**1**] [**2,3**]
Y la agrega a las particiones de 3.

Y así se construyó la tercer y cuarta partición de n=3. Repitiendo el procedimiento para la partición [**1,2**] se obtendrá la última partición del conjunto de particiones de 3.

Esa es la **base** del algoritmo exacto utilizado para generar las particiones. Se deja guardada en el código la partición base, que es la única partición de un conjunto de 1 elemento. Y para las demás se realiza el procedimiento mencionado, generando las de n a partir de las de n-1. Tiene una gran *'pinta'* recursiva, pero decidimos hacerlo iterativo.

A continuación, el pseudocódigo del algoritmo *(nuevamente, sin las podas)*:
Recibe un conjunto de particiones que esta vacío para que lo llene.

```
1  getPartitionsOfN(pesos, n, k, PARTICIONES)
2
3      PARTICIONES := {{1}}
4
5      para i desde 2 hasta n inclusive hacer
6        lengthAnterior := particiones.length
7
8        para j desde 0 hasta lengthAnterior inclusive hacer
9          copy := copyOf(j-esima particion de PARTICIONES)
10
11         agregar conjunto {i} a la j-esima particion
12
13         Para p desde 0 hasta copy.length - 1 inclusive hacer
14           copy[p].push_back(i).
15           Agregar copy a PARTICIONES.
16           copy[p].pop_back();
17         Fin Para
18
19       Fin para
20
21     Fin para
22
23     return;
24
25 end of getPartitionsOfN
```

---

**Las podas y estrategias:**

1. **Poda de cantidad de conjuntos.**
   Dado un n, la cantidad máxima de conjuntos que puede tener una particion de n es n. Pero las particiones de n tales que tienen mas de k conjuntos no nos interesan, ya que estamos en el problema de la k-PMP donde nos interesan las particiones que tengan a lo sumo k conjuntos.
   Ejemplo utilizando las imágenes usadas para la explicación del algoritmo; en ese caso, n = 3. Si nuestro k (por ejemplo) es 2, ¿tendría sentido agregar la primer particion al conjunto de particiones final ([1][2][3])? No lo tendría puesto que por definición de una k-PMP no puede ser una solución del problema, y estaríamos generando particiones que luego analizariamos cuando ya sabemos de antemano que no son posibles.

2. **Poda de mejor peso encontrado hasta el momento.**
   Esta poda se basa en que el peso de cualquier partición de n que tenga k o menos conjuntos es ¿= al peso de la solución óptima. Dicho esto, si supieramos que alguna particion de n (y tamaño ¡= k) tiene peso X, entonces cuando estamos construyendo las particiones, si notamos que la particion que estamos construyendo ya sobrepasó ese límite (a pesar de que aún no tenga los n elementos), entonces no tiene sentido seguir construyendola ya que el peso intraparticion de los elementos ya aplicados lo superó y por lo tanto no es óptima.

Si esto ocurre, se elimina esa partición de el conjunto de particiones credas hasta el momento. Este valor X lo conseguimos mediante una *cota inicial* que realizamos previo a la formación de las particiones de n, que será explicado en breve.

3. **Poda de actualización de mejor peso encontrado.**
   En la poda recién explicada, se detallaba que dado un peso X calculado anteriormente, las particiones que se estén construyendo y alcanzaran dicho límite serían removidas. Pero ¿por qué no ir actualizando este límite si vamos encontrando particiones tales que ya tienen todos los nodos cargados y mejor peso que X?
   Eso es lo que hace esta poda, si dada una partición U de n, tal que peso(U)<peso_optimo_parcial, entonces peso_optimo_parcial = peso(U). Esto permitirá que más particiones sean recortadas a futuro.
   *Aclaración:* Como se explicó, esta cota tiene utilidad recién cuando se encuentra una particion n. En nuestro algoritmo, no nos sirve mientras estamos construyendo las particiones de n-1, n-2,...,3,2 ya que necesitamos todos los nodos cargados para poder asegurar que encontramos un peso tal que peso<peso_optimo_parcial. Por ende somos conscientes que hasta la última iteración del for principal (pseudocódigo) no nos sirve, pero debido a que el peso de una partición actual en construcción ya lo calculamos para la poda anterior (poda número 2) realizar el checkeo y la asignación es O(1), así que no perdíamos nada por aplicarla.

4. **Estrategia previa para acotar el peso de la solución óptima.**
   En esta estrategia, ingresamos los nodos en orden entre k conjuntos, para que nos devuelta una configuración posible (potencialmente óptima) y luego tomamos su peso para usarlo como cota en la construcción de las particiones. Si bien la probabilidad de que esta solución sea óptima es muy baja, la probabilidad de que esta estrategia devuelva la peor configuración (y por ende no nos sirva) también lo es. La peor configuración C sería aquella tal que peso(C) >= peso(D) $\forall$ D partición de n.
   Si la cota devolviese la peor solución habría sido como si no hubiera existido, porque no hubiera acotado al problema.
   Para una mayor facilidad el pseudocódigo es el siguiente:

```
1  generarParticionInicial(nodos, k)
2    RESULTADO := Particion vacia;
3    set := 0;
4    incrementarParticion := true;
5
6    para cada nodo hacer
7      Si incrementarParticion
8        agregar un conjunto vacio a RESULTADO
9      Fin Si
10
11     RESULTADO[set].push_back(nodo);
12     set++;
13     Si i == k
14       i = 0;
15       incrementarParticion<-false;
16     Fin Si
17   Fin para
18
19   return RESULTADO;
20 end of generarParticionInicial
```

Dicho esto, a continuación el pseudocódigo *con podas* de la parte del algoritmo que se encarga de la construcción de las particiones:

```
1  getPartitionsOfN(pesos, n, k, PARTICIONES)
2
3      PARTICIONES := {{1}}
4
5      para i desde 2 hasta n inclusive hacer
6        lengthAnterior := particiones.length
7
8        para j desde 0 hasta lengthAnterior inclusive hacer
9          copy := copyOf(j-esima particion de PARTICIONES)
10
11         Si la j-esima particion aun no tiene tamano igual a k
12           agregar conjunto {i} a la j-esima particion
13         Sino
14           Borrar la j-esima particion de PARTICIONES;
15           j--;
16           lengthAnterior--;
17         Fin si
18
19         Para cada conjunto p de la particion copy hacer
20           copy[p].push_back(i);
21           peso := peso(copy);
22
23           Si peso < optimo_peso_parcial
24             Agregar copy a PARTICIONES
25             Si i == n
26               optimo_peso_parcial <- peso;
27             Fin Si
28           Fin Si
29
30           copy[p].pop_back();
31
32         Fin Para
33       Fin para
34     Fin para
35     return;
36 end of getPartitionsOfN
```