Parte I Problema 1

Dado un grafo simple $G = (V, E)$ y un entero k, se define una k-partición de G como una partición de V en k conjuntos de vértices $V_1 \dots V_k$. Y, dada una función de peso definida sobre las aristas de G, el peso de una k-partición es la suma de los pesos de las aristas intrapartición. Todos los pesos son positivos. Entonces, es importante notar que siempre que busquemos una k-PMP, y si $k < n$, no vamos a tener ninguna partición vacía. Puesto que si tuviéramos alguna partición V_i vacía, podríamos dividir a alguna otra partición no vacía V_j en dos: $V_{j'}$ y $V_{j''}$, quedando una k-PMP de peso menor o igual (porque los pesos de todas las aristas son positivos y al dividir en dos el conjunto V_j puede ser que nos hayamos deshecho de algunas aristas).

Desarrollar los siguientes puntos:

1. Relacionar el problema de k-PMP con el problema 3 del TP 1

   El problema 3 del TP 1 consistía en, dada una matriz de peligrosidad, que nos decía cuál era el peligro de juntar cada par de productos en un mismo camión, determinar alguna posible asignación de n productos en k camiones de manera tal que en ninguno de los k camiones se sobrepasase cierto umbral M de peligrosidad.
   Podríamos pensar a cada producto como un nodo de un grafo simple $G = (V, E)$, donde $|V| = n$ y cada par de nodos u, v están conectados por una arista $e = (u, v)$ de peso igual a la peligrosidad que generan los productos u y v al estar en un mismo camión. Ahora, llamaremos al problema del TP1 "problema 1" y al problema de hallar una k-PMP, "problema 2" y veremos si en distintos casos es posible utilizar cada uno de estos dos problemas para resolver el otro.

   (a) supongamos que tenemos una solución para el problema 1. En dicha solución, se cuenta con los camiones $c_1 \dots c_k$, tales que $(\forall i = 1..k) peligrosidad(c_i) \leq M$, y que contienen a los productos $p_1 \dots p_n$. Ahora, podemos definir un grafo G' tal que cada producto p_i se corresponda con un nodo n_i y que si entre los productos p_i y p_j se genera un hazard x, en el grafo G', los nodos n_i y n_j estén unidos por una arista de peso x. A partir de la solución al problema 1, construimos una solución al problema 2 sobre el grafo G' (que llamaremos `solución hermana` a la solución del problema 1) de la siguiente manera: para cada nodo $n_i \in G'$, si p_i fue enviado al camión c_j, mandamos al nodo n_i al conjunto j. Así, obtenemos una solución para el problema 2 sobre el grafo G' de costo total $S = \sum_{i=1}^{k} peligrosidad(c_i)$. Entonces, podemos asegurar que la solución óptima para el problema de k-PMP sobre G' es, como máximo, S. Esta solución podría no ser óptima porque el problema 1 impone una restricción en el hazard que puede haber en cada camión, pero el problema 2 no impone ninguna restricción en el hazard que puede haber en cada conjunto. Y, en nuestra transformación de una solución del problema 1 en otra solución para el problema 2, hacemos que cada camión corresponda a un conjunto. Con lo cual podríamos no estar considerando algunas soluciones que sí habría que considerar en el problema 2.

   (b) por otro lado, supongamos que, contando sólo con los camiones $c_1 \dots c_k$, sabemos que el problema 1 no tiene solución. Entonces, si construimos el mismo grafo G' que construimos en el item anterior, **no** podemos decir nada sobre la existencia o no de una solución al problema 2 sobre el grafo G', porque el problema 1 impone una restricción en el hazard que puede haber en cada camión, pero en el problema 2 puede haber cualquier cantidad de hazard en cada conjunto. Entonces, al igual que en el item anterior, debido a la forma en que transformamos soluciones al problema 1 en soluciones al problema 2, podrían existir soluciones para el problema 2 en el grafo G', que no tengan `solución hermana` en el problema 1. Esto, obviamente, no sucede en el caso en que $M = \infty$, puesto que en ese caso, no hay ninguna restricción en el problema 1 para el hazard que puede haber en cada camón.

(c) supongamos que tenemos una solución para el problema 2, sobre el grafo G'. En dicha solución, se cuenta con los conjuntos $1, 2, \dots k$ y los nodos $v_1 \dots v_n$. A partir del grafo G', podemos construir una instancia para el problema 1 de la siguiente forma: el nodo v_i, corresponderá al producto p_i. Para cada par de nodos v_i y v_j unidos por una arista de peso x, vamos a asignar una hazard de x unidades para el par de productos p_i y p_j. Además, dada una solución para el problema 2, construimos una `solución hermana` para el problema 1 de la siguiente manera: si el nodo n_i fue enviado al conjunto j, entonces mando al producto p_i al camión c_j.
Supongamos además que para cada conjunto i, $peso(i) \leq M$ para algún número M. Entonces, con nuestra transformación podemos transformar nuestra solución óptima para el problema 2 en una solución óptima para el problema 1, pues cada conjunto se correspone con un camión, y en el problema 2 se busca utilizar la menor cantidad de conjuntos posibles (de la misma forma que en el problema 1 se busca utilizar la menor cantidad de camiones posibles). Sin embargo, si la solución utilizara algún conjunto j con $peso(j) > M$, entonces nuestra transformación no generaría una solución válida para el problema 1.

2. Relacionar el problema de k-PMP con el problema de coloreo de los vértices de un grafo.

   Primero que nada, hay que decir que por más que tengamos un método para hallar un k-coloreo de un grafo, es muy probable que esto nos nos sirva para hallar una k-PMP de dicho grafo puesto que el k-coloreo no considera el valor de las aristas de G sino que simplemente se preocupa por hacer un análisis de adyacencias y no-adyacencias entre los nodos de dicho grafo.

   Sin embargo, si somos capaces de determinar una k-PMP para cualquier grafo G, entonces también somos capaces de determinar la existencia de un k-coloreo para cualquier grafo G'.

   Tomemos el grafo G para el cual queremos hallar un k-coloreo. Y a partir de dicho grafo, construyamos un nuevo grafo G' tal que:

   (a) $V(G) = V(G')$
   (b) $E(G) = E(G')$, pero poniéndole a cada arista $f = (u, v)$ un peso: peso$(f) = \infty$.
   (c) para los pares de nodos u, v que no sean adyacentes en G, en G' agregar una arista $f = (u, v)$ que los una, tal que $peso(f) = 0$.

   Luego, buscamos una k-PMP en el nuevo grafo G'. Si el peso total generado por dicha partición es cero, entonces quiere decir que es posible encontrar k conjuntos de vértices tales que en cada conjunto cada par de vértices está unido por una arista de peso cero (i.e. en cada conjunto, no puede haber dos nodos que fueran adyacentes en G, porque sino en G' habría un eje de peso infinito que una a estos nodos y haría que el peso total del conjunto sea mayor que cero).

   Entonces, definimos el k-coloreo $C : V(G) \to \mathbb{N}/\ C(v) = i$ sii k-PMP sobre G' manda a v al conjunto i.

   Podemos asegurar que C es un k-coloreo porque k-PMP siempre utiliza los k conjuntos y en nuestra definición hacemos corresponder cada conjunto a un color diferente.

   Podemos asegurar que si C es un coloreo válido entonces es óptimo y que si C no es un coloreo válido, entonces G no es k-coloreable. Esto es así porque C es válido si y sólo si se encontró una k-PMP de costo total cero en G': si el costo fuera mayor a cero, debería haber alguna arista que genere parte de este costo, y las únicas aristas con peso positivo en G' son las que habían en $E(G)$. Y si esta arista genera costo en G', quiere decir que k-PMP mandó los dos nodos incidentes en dicha arista al mismo conjunto. Entonces, nuestra función C mandaría a estos dos nodos vecinos al mismo color. Pero entonces el coloreo resultante no sería válido. En cambio, si el costo total de la solución de k-PMP sobre G' fuera cero, entonces nuestra función C generaría un coloreo válido pues como el costo total es cero, sabemos que no se mandan nodos que fueran vecinos en G al mismo conjunto. Y nuestra función C manda los nodos que están en el mismo conjunto al mismo color.

   Veamos con un ejemplo, en el cuál buscamos un 2-coloreo para G a partir de una 2-PMP de G':

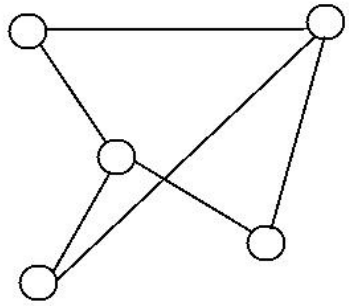

Figure 1.0.1: grafo G

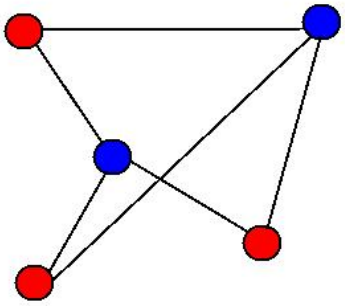

Figure 1.0.2: 2-coloreo para G

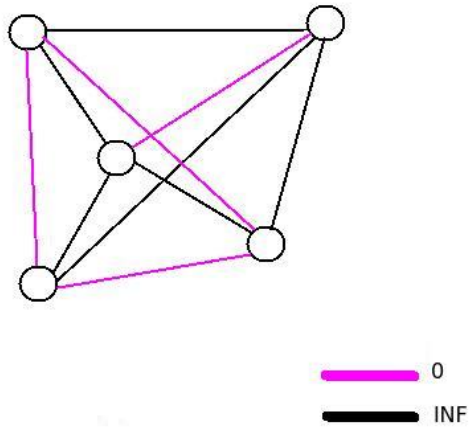

Figure 1.0.3: grafo G'

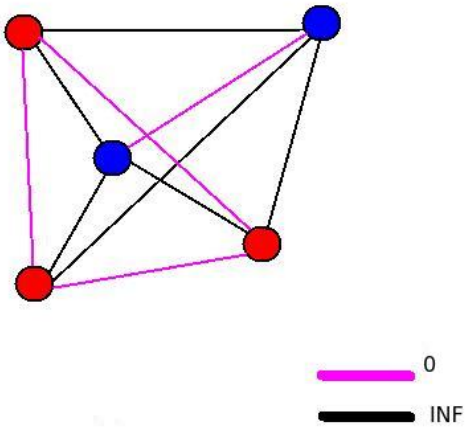

Figure 1.0.4: 2-PMP para G'

3. Describir situaciones de la vida real que puedan modelarse utilizando k-PMP. Algunas situaciones de la vida cotidiana divertidas que pueden ser modeladas utilizando k-PMP podrían ser:

   (a) Quiero distribuir a n alumnos en k aulas de forma que estos se intenten copiar lo menos posible en el examen: La probabilidad de que un alumno se intente copiar en el examen es mayor cuanto más cómodo se sienta el alumno en el aula. Y un alumno se siente más cómodo en el aula cuanto mayor sea la sumatoria del índice de confianza que tiene con el resto de los alumnos de su aula.

   (b) Un alumno busca distribuir n materias en k cuatrimestres minimizando sus horas extra de estudio, si cada par de materias que son cursadas en el mismo cuatrimestre hacen que el alumno tenga que estudiar cierta cantidad de horas extra para poder aprobarlas.

   (c) Se cuenta con n prendas de ropa y con una lavadora. Dependiendo de qué prendas de ropa se metan al mismo tiempo, el lavarropas tarda más o menos tiempo en lavarlas. Se busca seleccionar k conjuntos de prendas tales que se minimize el tiempo total que la lavadora esta encendida.

Parte II Problema 2

Diseñar e implementar un **algoritmo exacto** para k-PMP y desarrollar los siguientes puntos:

1. Explicar detalladamente el algoritmo implementado. Elaborar podas y estrategias que permitan mejorar los tiempos de resolución
2. Calcular el orden de complejidad temporal de peor caso del algoritmo
3. Realizar una experimentación que permita observar los tiempos de ejecución del algoritmo en función del tamaño de entrada y de las podas y/o estrategias implementadas

Parte III Problema 3

Diseñar una **heurística constructiva golosa** para k-PMP y desarrollar los siguientes puntos:

1. Explicar detalladamente el algoritmo implementado: La heurística que pensamos se basa en algunas observaciones claves:

   (a) la primera, es que nuestro algoritmo goloso va a ir agregando nodos a cada uno de los conjuntos según algún criterio. Y, como lo que se quiere es minimizar la suma total de los costos de los conjuntos, podríamos agregar cada nodo al conjunto en el cuál este agrega el menor peso posible y dejarlo en ese conjunto. Entonces, según esta observación, si llamamos w_i al peso total del conjunto S_i, agregaríamos el nodo t al conjunto S_i tal que se minimize w_i.

   (b) otra idea que sumamos a esta heurísitica es recorrer los nodos que vamos agregando a cada conjunto en un orden. Antes de recorrer la lista de nodos para ir asignándolos a conjuntos, ordenamos la lista de nodos según:

      i. según el peso de la máxima arista incidente en cada nodo y, en caso de empate

      ii. según la suma total de las aristas incidentes en cada nodo (de mayor a menor) y, en caso de empate

      iii. según la cantidad de vecinos de cada nodo (de mayor a menor)

      La razón por la cuál elegimos el primer criterio, es para asegurarnos de agregar al principio los nodos que tienen aristas incidentes de mucho peso a conjuntos distintos. Es esperable que este tipo de ordenamiento dé buenos resultados en grafos con pocas aristas de mucho peso y muchas aristas de bajo peso en comparación, ya que el algoritmo va a enviar los nodos que son adyacentes a esas aristas de mucho peso a conjuntos distintos, logrando así que las aristas de mucho peso no sumen costo a la solución final.

      La razón por la cuál elegimos el segundo criterio (para desempatar), es que, si asignamos conjunto primero a los nodos que generan mayor hazard (con mayor suma total de sus aristas incidentes), intentamos agregar siempre en conjuntos distintos a los nodos que potencialmente generan mayor hazard. Así, cada vez que tengamos que agregar un nuevo nodo a algún conjunto, esperamos tener mayor cantidad de conjuntos en donde meter dicho nodo sin generar demasiado hazard extra.

      La razón por la cuál elegimos el tercer criterio es similar a la razón del segundo criterio: siempre intentamos que se agreguen en distintos conjuntos los nodos que potencialmente pueden generar más hazard. Y creemos que cuantos más nodos sean adyacentes a un nodo, mayor será la probabilidad que tenga algún vecino de peso alto.

A continuación exponemos un pseudocódigo para el algoritmo goloso que implementamos.

```
   input  : n: cantidad de nodos, m: cantidad de aristas, k: cantidad de conjuntos, G: grafo
   output: conjuntos: matriz de enteros tal que conjuntos[j][i] == 1 sii el conjunto j contiene
           al nodo i
 1 vector< int > conjuntos
 2 ordenar G según el peso de la máxima arista incidente a cada nodo y en caso de empate
   según la suma de las aristas adyacentes a cada nodo
 3 for cada nodo i = 1 ... n de G do
 4 |   valorMin = INF
 5 |   for cada conjunto j = 1 ... k do
 6 |   |   - calcular h = hazard que genera agregar el nodo i al conjunto j (esto lo hace la
   |   |   función CalcularHazard)
 7 |   |   if h < valorMin then
 8 |   |   |   - valorMin = h
 9 |   |   |   - si antes habíamos agregado al nodo i a algún conjunto, sacarlo de dicho
   |   |   |   conjunto (conjuntos[conjAnterior][i] = 0).
10 |   |   |   - agregar nodo i al conjunto j (conjuntos[j][i] = 1)
11 |   |   |   - conjAnterior = j
12 |   |   end
13 |   end
14 end
15 return conjuntos
```

Algorithm 1: Algoritmo 1

Para devolver la solución en el formato que pide el enunciado, además implementamos la función transformarSolución. A continuación presentamos el pseudocódigo de dicha función.

```
   input  : matriz conjuntos (M, de tamaño k x n)
   output: res: vector de enteros, tal que res[i] == j sii el nodo i está en el conjunto j
1 for cada nodo i = 1 ... n do
2 |   for cada conjunto j = 1 ... k do
3 |   |   if conjuntos[j][i] == 1 then
4 |   |   |   res[i] = j
5 |   |   end
6 |   end
7 end
```

Algorithm 2: función transformarSolución

Luego, veamos qué hace este algoritmo con un ejemplo:

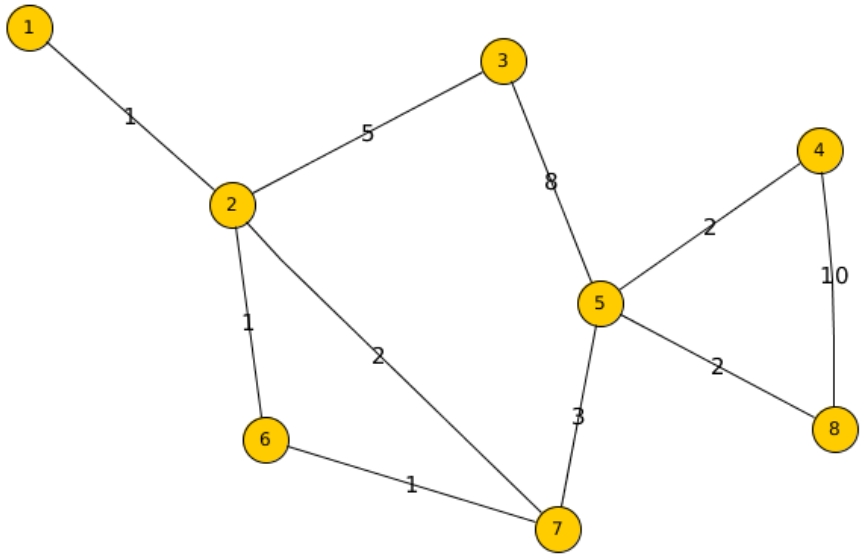

Figure 3.0.5: Instancia 1

Ejemplo de entrada: instancia 1

```
8 10 3
1 2 1
2 6 1
2 3 5
2 7 2
6 7 1
7 5 3
3 5 8
5 4 2
5 8 2
4 8 10
```

Ejemplo de salida: instancia 1

```
1 1 2 3 1 3 1 2
```

En el ejemplo, primero el algoritmo ordena los nodos del grafo G según el peso de sus máximas aristas. Es decir con el orden: 1 6 7 2 3 5 4 8 (en caso de empate, ordena según la suma de las aristas incidentes a los nodos). Luego, recorre esta lista de nodos y los va agregando de a uno al conjunto que dicho nodo agrega menos peso. Por ejemplo, primero agrega los nodos 1 y 6 al conjunto 1, y luego como el nodo 7 agrega 2 unidades de peso al conjunto 1 pero 0 al 2 (porque el conjunto 2 todavía no tiene ningún nodo), entonces agregamos el nodo 7 al conjunto 2. Con un razonamiento similar, en el próximo paso asignamos el nodo 2 al conjunto 3. De esta forma, se llega a la asignación de conjuntos 1 1 2 3 1 3 1 2 a cada uno de los nodos del grafo.

2. Calcular el orden de complejidad de peor caso del algoritmo:

   Como se puede observar en el pseudocódigo que expusimos en 1.b), el algoritmo que desarrollamos utiliza dos for, que van uno entre $1 \dots n$ y otro entre $1..k$. Si la complejidad de lo que hay adentro de estos ciclos es $O(T(n))$, entonces la complejidad total del algoritmo sería $O(knT(n))$.
   Dentro de estos ciclos lo que hacemos es llamar a la función CalcularHazard. Como se puede ver en el pseudocódigo expuesto a continuación, lo que hace dicha función es recorrer para el

nodoActual todos sus adyacentes e ir chequeando si cada adyacente está en el conjunto *conj*. Si sí lo está, entonces actualizamos el hazard que generaría agregar al nodo *nodoActual* al conjunto *conj*. Esto lo hacemos de forma lineal en $|V(G)| = n$, a partir de haber guardado el grafo G en una estructura conveniente (lista de adyacencia de nodos). Luego, $T(n) = O(n)$.
Entonces, la complejidad de nuestro algoritmo en el peor caso es $O(knT(n)) = O(kn^2)$, ya que la función $transformarSolucion$ consta de dos for que van entre 1 y n, con lo cual tiene complejidad $O(n^2) \subset O(kn^2)$ y no suma complejidad temporal al algoritmo.

```
input  : grafo (G), conjunto al que quiero agregar un nodo (conj), nodo que quiero agregar
         a conj (nodoNuevo)
output: costo de agregar el nodo nodoNuevo al conjunto conj
1 hazardAgregado = 0
2 for cada nodo i adyacente a nodoNuevo do
3     if el i-ésimo nodo adyacente a nodoNuevo está en el conjunto conj then
4         deltaPeso = peso de la arista que une nodoNuevo con el i-ésimo nodo adyacente a
          nodoNuevo
5         hazardAgregado = hazardAgregado + deltaPeso
6     end
7 end
8 return hazardAgregado
```

Algorithm 3: Función CalcularHazard, que calcula el costo de agregar el nodo *nodoNuevo* al conjunto *conj*.

3. Describir instancias de k-PMP para las cuales la heurística no proporciona una solución óptima. Indicar qué tan mala puede ser la solución respecto de la solución óptima.

   Veamos dos ejemplos en los que nuestra heurística no funciona correctamente. En estos ejemplos, la solución es infinitamente peor que la óptima.

   En el primer ejemplo, se busca una 2-PMP del grafo G_0. Dicho grafo es un ciclo de nodos unidos por aristas de peso infinito y con algunas aristas internas, de peso 1. Como se puede observar en los esquemas, la heurística proporciona una solución de peso infinito, mientras que la solución óptima tiene peso cero. El problema con el que se encuentra la heurística radica en que se recorren los nodos en cierto orden, y una vez que se coloca cada nodo en algún conjunto, este no puede ser cambiado a otro. En este ejemplo en particular, se recorren los nodos del grafo en el orden: N6, N4, N2, N3, N1, N5. Entonces, cuando llegamos a agregar el nodo N3, ya se han agregado los nodos N6 y N2 al conjunto A, y el nodo N4 al conjunto B. Entonces, a esta altura ya sea que agreguemos el nodo N3 al conjunto A o al B, en cualquiera de los conjuntos agregaremos hazard infinito.

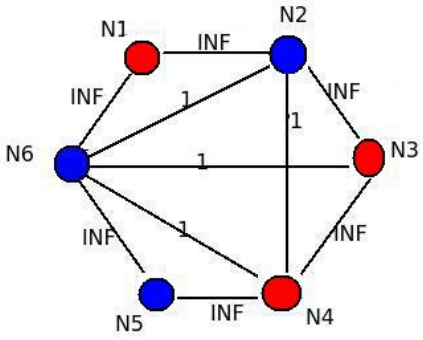

Figure 3.0.6: Solución de la heurística

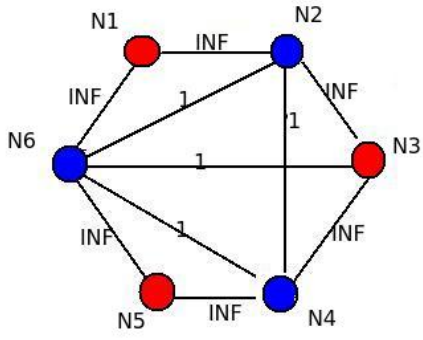

Figure 3.0.7: Solución óptima

En el ejemplo 2, se busca una 2-PMP del grafo G_1. Dicho grafo es completo bipartito, a excepción de dos nodos u y v en uno de los conjuntos que conforman la bipartición, que están unidos por una arista de peso 1. Las aristas que unen cada nodo de un lado de la bipartición con nodos del otro lado de la bipartición tienen, como se observa en el dibujo, peso INF. La solución óptima a este problema es la presentada en la segunda figura, y es de peso $1 < \infty$, mientras que la solución que brinda nuestra heurística es de peso infinito, y una posible representación de la misma esta esquematizada en la primera figura. La heurística falla porque vamos agregando los nodos a cada conjunto en un orden que nos conduce a una solución no óptima. En este caso, los nodos u y v son los que primero van a ser agregados a conjuntos. Entonces, como nuestro algoritmo es goloso y cada vez que agrega un nodo a un conjunto, lo agrega al conjunto que minimiza el hazard nuevo que se genera en la solución, se envía u a un conjunto (rojo) y v al otro (azul). Esto hace que, más adelante, para cada nodo del conjunto B, como está unido tanto a u como a v, no importa si lo agregamos al conjunto rojo o al conjunto azul, en ambos casos el hazard extra que genera es siempre INF. Esto hace que, al final, nuestra solución no sea la óptima.

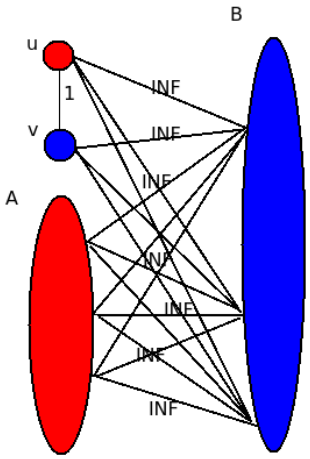

Figure 3.0.8: Mejor solución posible de la heurística

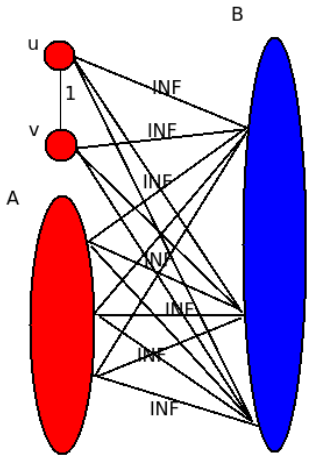

Figure 3.0.9: Solución óptima

4. Realizar una experimentación que permita observar la performance del algoritmo en términos de tiempo de ejecución en función del tamaño de la entrada.

Parte IV Problema 4

Diseñar una **heurística de búsqueda local** para k-PMP y desarrollar los siguientes puntos:

1. Explicar detalladamente el algoritmo implementado. Plantear al menos dos vecindades disintas para la búsqueda.
   En el algoritmo que implementamos seguimos los siguientes pasos:

   (a) asignamos de forma aleatoria uno de los k conjuntos a cada nodo

   (b) mientras haya alguna solución vecina mejor que la actual, actualizamos todos los conjuntos de nodos y el peso de la solución

   (c) repetimos los dos pasos anteriores hasta que no haya una solución vecina mejor ó, hasta haberlo repetido una cantidad límite de x veces.

   La razón por la cual repetimos el proceso para varias soluciones iniciales es porque dependiendo de la solución inicial que consideremos, hallaremos un mínimo local distinto. Con lo cuál, para que la probabilidad de hallar el mínimo global sea lo más alta posible, hay que considerar la mayor cantidad de soluciones iniciales distintas posible.

   Las dos vecindades que definimos fueron:

   (a) para cada nodo, chequear si cambiándolo a algún otro conjunto, se podría reducir el costo total de la solución.

   (b) para cada nodo, chequear si swappeandolo de conjunto con otro nodo ó mandándolo a un conjunto que esté vacío se reduce el costo.

   A continuación, presentamos el pseudocódigo para cada una de las vecindades.
   Nota: La función calcularCosto(G,v,i) recorre los nodos adyacentes al nodo i, y acumula la suma de las aristas que unen a i con algún otro nodo adyacente a i en la solución actual.

```
   input  : grafo G, solución parcial v
   output: modifica v y genera una nueva solución parcial, v'
 1 for cada nodo i = 1 ... n do
 2 |   for j = cada conjunto 1 ... k do
 3 |   |   costoDelNodoViejo = calcularCosto(G, v, i)
 4 |   |   v' = cambiar nodo i al conjunto j
 5 |   |   costoDelNodoViejo = calcularCosto(G, v', i)
 6 |   |   if costoDelNodoNuevo < costoDelNodoViejo then
 7 |   |   |   v = v'
 8 |   |   |   actualizo costo de v (en O(1))
 9 |   |   |   res = 1
10 |   |   end
11 |   |   if costoDelNodoNuevo ≥ costoDelNodoViejo then
12 |   |   |   res = 0
13 |   |   end
14 |   end
15 end
16 return res
```

Algorithm 4: Algoritmo de búsqueda local con vecindad 1

```
   input  : grafo G, solución parcial v
   output: modifica v y genera una nueva solución parcial, v'
 1 for  cada nodo i = 1 ... n do
 2 |   // Primera parte: chequear el costo generado si hacemos algún swap
 3 |   costoOriginal = v.second // costo de la solución sin hacer el swappeo
 4 |   for  cada nodo j = 1 ... n do
 5 |   |   costoNodoViejo1 = calcularCosto(G,n,v,i)
 6 |   |   costoNodoViejo2 = calcularCosto(G,n,v,j)
 7 |   |   swap(v.first[i], v.first[j]) // swapeo los nodos de conjunto
 8 |   |   costoNodoNuevo1 = calcularCosto(G,n,v,i)
 9 |   |   costoNodoNuevo2 = calcularCosto(G,n,v,j)
10 |   |   costoViejo = costoNodoViejo1 + costoNodoViejo2
11 |   |   costoNuevo = costoNodoNuevo1 + costoNodoNuevo2
12 |   |   if costoNuevo < costoViejo then
13 |   |   |   v.second = v.second - costoViejo + costoNuevo // actualizo costo haciendo swap
14 |   |   |   break // encontré swap conveniente. Dejo de buscar posibles swaps
15 |   |   else
16 |   |   |   swap(v.first[i], v.first[j]) // si no encontré un swap que disminuya el costo
17 |   |   end
18 |   end
19 |   // Segunda Parte: calcular costo generado cambiando un nodo a un conjunto vacío
20 |   if  hay algún conjunto vacío then
21 |   |   for  cada nodo j = 1 ... n do
22 |   |   |   if cambiar nodo j al conjunto conjuntoVacio genera menos hazard que el swap ó
   |   |   |   genera menos que la solución actual (si no hicimos ningún swap) then
23 |   |   |   |   - deshago el swap (sólo si habíamos hecho un swap)
24 |   |   |   |   - mando el nodo j al conjunto conjuntoVacio
25 |   |   |   |   - break // si encontramos un conjunto propicio, dejamos de buscar
26 |   |   |   end
27 |   |   end
28 |   end
29 end
```

Algorithm 5: Algoritmo de búsqueda local con vecindad 2

En el caso de la primera y segunda vecindad, una instancia posible y su solución serían (partiendo de la solución aleatoria inicial 3 3 3 3 1 2 3 3).

Ejemplo de entrada

```
8 10 3
1 2 1
2 6 1
2 3 5
2 7 2
6 7 1
7 5 3
3 5 8
5 4 2
5 8 2
4 8 10
```

Ejemplo de salida: vecindad 1

```
2 1 3 2 1 2 3 3
```

Ejemplo de salida: vecindad 2

```
3 3 3 3 1 3 3 2
```

Vale aclarar que es esperable que, para distintas vecindades, en general nos den resulados diferentes. Además, en este ejemplo en particular, usando la vecindad 2 obtenemos una solución muy mala (de costo total igual a 10), mientras que para la vecindad 1, obtenemos una solución buena (costo total igual a 0). En este ejemplo en particular, como inicialmente no hay ningún conjunto vacío, cuando utilizemos la vecindad 2, sólo se van a efectuar swaps de nodos entre conjuntos, y la cantidad de nodos que hay en cada conjunto se va a mantener invariante (si hubiera algún conjunto vacío de seguro se le agregaría un nodo a dicho conjunto, ya que al cambiar un nodo a un conjunto vacío, siempre se puede reducir más el costo total de la solución que haciendo un swap).

2. Calcular el orden de complejidad temporal de peor caso de una iteración del algoritmo de búsqueda local (para las vecindades planteadas). Y si es posible, dar una cota superior para la cantidad de iteraciones de la heurística.

   En el caso de la vecindad 1, el costo de cada iteración del algoritmo de búsqueda local es $O(kn^2)$ y esto se deduce fácilmente del pseudocódigo presentado en el punto anterior (empleamos dos for, y adentro de uno de estos, llamamos a una función que calcula el costo de agregar el nodo a un conjunto, cuyo costo computacional es $O(n)$, pues cada nodo es adyacente como máximo a $n-1$ nodos).

   En el caso de la vecindad 2, el costo de cada iteración del algoritmo de búsqueda es $O(n^3)$. Esto se deduce del pseudocódigo presentado más arriba: el algoritmo consta de una serie de ciclos for, dentro de los cuáles se llama a la función $calcularCosto$, que tiene complejidad temporal $O(n)$. Luego, entre las líneas 4 y 18 la complejidad es $O(n^2)$. La línea 20 tiene una complejidad de $O(k)$, pues debe recorrer todos los conjuntos chequeando la cantidad de nodos que tienen. Entre las líneas 21 y 26, la complejidad es $O(n^2)$ pues se recorren los n nodos en un for y en cada iteración de dicho ciclo, se recalcula el hazard que tiene la nueva solución cambiando al nodo j al conjunto $conjuntoVacio$ para ver si hacer este cambio genera menos hazard que generaba el swap en $O(n)$. Entonces, como todo lo recién mencionado está metido en un for que recorre todos los nodos ($O(n)$), se llega a que la complejidad es $T(n) = O(n^3) + O(nk) + O(n^3) = O(n^3)$.

   Además, podemos asegurar que la cantidad de iteraciones que se efectúan son como máximo S iteraciones, donde S es el peso de la solución aleatoria con la que iniciamos el algoritmo. Esto es así porque en cada iteración o se baja en al menos una unidad el costo de la solución y se sigue iterando (hasta llegar como máximo a costo cero en cuyo caso finalizamos la búsqueda) ó en alguna iteración no se encuentra ninguna solución vecina de costo menor que la solución actual, y se da por finalizada la búsqueda.

3. Realizar una experimentación que permita observar la performance del algoritmo comparando los tiempos de ejecución y la calidad de las soluciones obtenidas, en función de las vecindades utilizadas y elegir, si es posible, la configuración que mejores resultados provea para el grupo de instancias utilizado.

Parte V Problema 5

Utilizando las heurísticas implementadas en los puntos anteriores, diseñar e implementar un algoritmo para k-PMP que use la **metaheurística GRASP** y desarrollar los siguientes puntos:

1. Explicar detalladamente el algoritmo implementado. Plantear distintos criterios de parada y de selección de la lista de candidatos (RCL) de la heurística golosa aleatorizada.

   La metaheurística GRASP (Greedy Randomized Adaptive Search Procedure) es una combinación entre una heurística golosa "aleatorizada" y un procedimiento de búsqueda local. A grandes rasgos, lo que hace nuestro algoritmo es:

   (a) Mientras no se alcance el criterio de terminación:

      i. Obtener una solución $s \in S$ mediante nuestra heurística golosa, aleatorizada
      ii. Mejorar la solución s mediante nuestra heurística de búsqueda local
      iii. Recordar la mejor solución obtenida hasta el momento
      iv. Volver a (a)

   Cuando usemos nuestra heurística golosa, no se elige en cada paso la mejor solución, sino que se confecciona una RCL (lista restricta de candidatos) de acuerdo con algún criterio de selección, y en cada paso se elige aleatoriamente un candidato de dicha lista. Dos opciones podrían ser:

   (a) que la RCL contenga los candidatos cuyo valor sea no menor que un cierto porcentaje α del valor del mejor candidato
   (b) que la RCL contenga los mejores β candidatos
   (c) una combinación de ambas

   En nuestro caso, optamos por la opción (b), ya que es la más simple, y no plantea inconvenientes en cuanto a la complejidad temporal ni algorítmica que requiere al ser implementada. Es decir, en cada paso no agregamos a un conjunto el nodo con máxima arista incidente, sino que nos armamos una lista con los β nodos con mayor máxima arista incidente y elegimos aleatoriamente uno de dichos nodos para agregarlo al conjunto en el que este genere menor hazard.

   Cada solución obtenida de esta forma será mejorada en el próximo paso del algoritmo mediante nuestra heurística de búsqueda local a partir de la vecindad que nos dio mejores resultados. Si de esta forma obtenemos una solución mejor que la última mejor solución obtenida, nos guardamos esta nueva solución.

   Después de esto, el procedimiento continúa generando soluciones con randomized greedy y mejorándolas con búsqueda local, hasta que se cumpla el criterio de parada elegido, que podría ser, por ejemplo: que no se encontró una mejora en las últimas k iteraciones, se alcanzó un límite prefijado de k iteraciones, se obtuvo una solución cercana cero (que es cota inferior para el costo de cada solución), o que se obtuvo muchas veces la misma solución.
   En nuestro caso, optamos por cortar la búsqueda cuando no se encontró una mejora en las últimas k iteraciones.

2. Realizar una experimentación que permita observar los tiempos de ejecución y la calidad de las soluciones obtenidas. Se debe experimentar variando los valores de los parámetros de la metaheurística (lista de candidatos, criterios de parada, etc.) y elegir, si es posible, la configuración que mejores resultados provea para el grupo de instancias utilizado.

Parte VI Problema 6

Una vez elegidos los mejores valores de configuración para cada heurística implementada (si fue posible), realizar una **experimentación sobre un conjunto nuevo de instancias** para observar la performance de los métodos comparando nuevamente la calidad de las soluciones obtenidas y los tiempos de ejecución en función del tamaño de entrada. Para los casos que sea posible, comparar mediante gráficos adecuados y discutir al respecto de los mismos.