DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

Heuristicas

03 / 10 / 2014

Algorítmos y Estructura de Datos III

Integrante	LU	Correo electrónico
Ariel Paez	668/09	twizt.hl@gmail.com
Patito Campanita	866/10	campanita24@gmail.com
Augusto Mascitti	954/11	mascittija@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. k-Partición de Mínimo Peso (k-PMP)

1.1. Introducción

En este último Trabajo Práctico de la materia vamos a afrontar la resolución del problema de encontrar una k-Partició de Peso Mínimo (de ahora en más k-PMP), el cual se trata de un problema que pertenece a la clase NP-Completo[1] y por lo tanto no se conoce un algoritmo en tiempo polinomial que lo resuelva.
Por tal motivo, vamos a buscar una solución exacta mediante un BackTraking y luego realizar algoritmos aplicando distintas heurísticas (Golosa, Busqueda Local, GRASP) para analizar las ventajas y desventajas de cada una.

1.2. Problema

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 $V_1, \ldots, V_k$. Las aristas intrapartición de una k-partición, son aquellas aristas de G cuyos extremos se encuentran en un mismo conjunto de la partición, es decir, una arista $uv \in E$ es intrapartición si existe un $i \in 1, \ldots, k$, tal que $u, v \in V_i$. Dada una función de peso $\omega : E \to \mathbb{R}_+$, definida sobre las aristas de G, el peso de una k-partición es la suma de los pesos de las aristas intrapartición.

El problema k-PMP consiste en encontrar, dado un grafo simple $G = (V, E)$, un entero k y una función $\omega : E \to \mathbb{R}_+$, una k-partición, que tenga a lo sumo k subconjuntos, de V con peso mínimo.

1.3. Ejemplos

En el seiguiente ejemplo vamos a ver lo que sería una partición cualquiera del grafo y luego la k-PMP

Se el siguiente grafo:

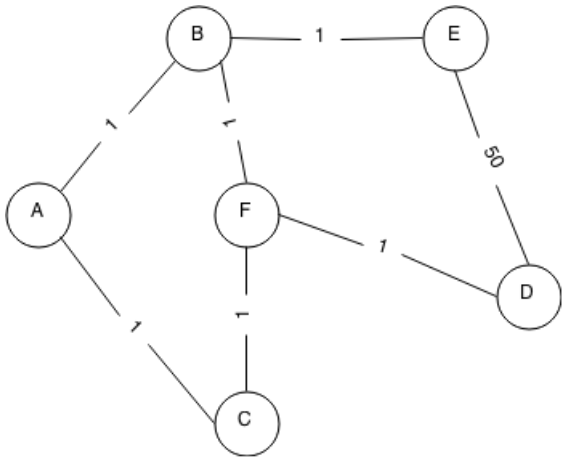


Figura 1: Gráfo Original.

[1] http://en.wikipedia.org/wiki/Minimum_k-cut

Podemos realizar la siguiente 2-partición

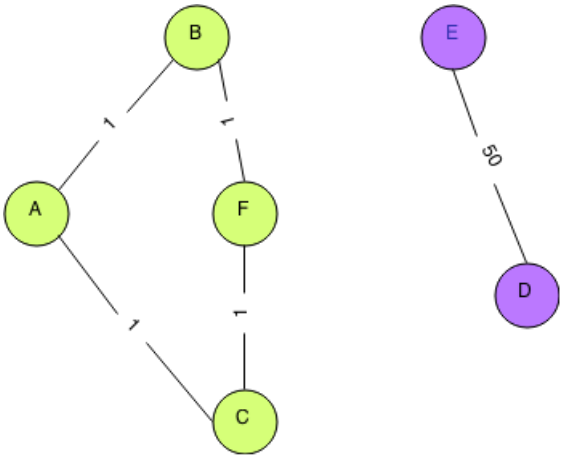


Figura 2: 2-Partición.

Se puede apreciar fácilmente que esta partición no cumple que sea mínima dentro de todas las 2-particiones posibles. Pues el peso total es de 54. Y por ejemplo existe la siguiente 2-particion en donde la suma del peso de sus aristas intraparticiones es igual a 0.

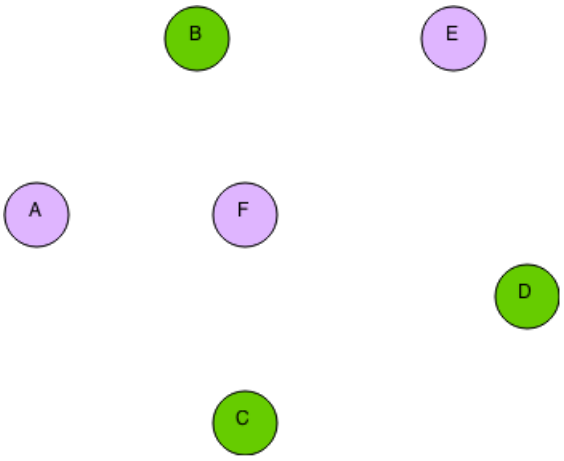


Figura 3: 2-Partición de peso mínimo.

1.4. Testing

Luego de implementar todos los algoritmos pertinentes para la resolusión del problema k-PMP, vamos a realizar distintos tipos de tests y análisis.

Por un lado vamos a generar tres familias de grafos de las cuales conocemos sus soluciones para poder medir tanto el rendimiento del algoritmo exacto y las distintas heurísticas, como la distancia de las soluciones de las heurísticas contra las del algoritmo exacto. Para esto tenemos un generador de grafos en el cual podemos configurar la cantidad de nodos y la cantidad de

aristas totales con pesos aleatoreos, siendo 0 % un grafo con n nodos aislados y el 100 % un grafo completo de n nodos. Con este generador vamos a generar las tres familias de grafos que se diferencian básicamente por su cantidad de aristas. Siendo estos tres grupos, grafos con el 15 % de aristas, 50 % y 100 % y vamos a variar n desde 6 a 15 por ser un numero manejable para el algoritmo exacto.

Luego para cada heuristica vamos a intentar identificar familias de grafos para las cuales se comporten razonablemente bien y ver si existen algunas familias para las cuales las mismas los resultados sean inadmisibles.

2. Comparativas k-PMP

2.1. k-PMP y Ej3-TP1

El ejercicio 3 del trabajo práctico 1 "Biohazard", consistía en distribuir químicos en camiones para que puedan ser transportados de una central de químicos a otra. El problema estaba en que hay ciertos químicos que no pueden transportarse juntos porque presentan un gran nivel de peligrosidad. Por lo que se determinó un coeficiente de peligrosidad que determina cuan peligroso es transportar dos ciertos químicos juntos, y un umbral de peligro que cada camión no debe sobrepasar en la suma de coeficientes de peligrosidad que existe entre los quimicos que transporta. Con esto y dado un conjunto de productos con sus coeficientes de peligrosidad se buscaba distribuirlos de forma de minimizar la cantidad de camiones utilizados, obviamente sin que cada uno de estos sobrepase el umbral de peligro.

El problema es basicamente el mismo que k-PMP, con la diferencia de que en k-PMP no existe umbral alguno, pero existe un limite en la cantidad de camiones, k, de manera que lo que se busca optimizar no es la cantidad de camiones, sino (llevando el ej Biohazard a k-PMP) la peligrosidad total de los k camiones.

2.2. k-PMP y Coloreo de Vértices

Se llama coloreo a la asignación de colores a los vértices de un grafo tal que dos vértices que compartan la misma arista tengan colores diferentes. Un coloreo que usa a lo sumo k colores se llama k-coloreo. Un grafo al que se le puede asignar un k-coloreo se lo llama k-coloreable. Por ultimo un subconjunto de vertices con el mismo color asignado en el coloreo se llama clase de color y cada clase forma un conjunto independiente. De esta forma podemos definir un k-coloreo como una particion del conjunto de aristas del grafo en k conjuntos independientes. (Para la siguiente explicacion vamos a suponer que el grafo no posee aristas de peso 0). Como se explicó antes, el problema k-PMP consiste en particionar las aristas en k conjuntos de modo que la suma de los pesos de las aristas intraparticion sea mínima. Cuando el peso de cada k-partición resulta en 0 quiere decir que dentro de cada particion no existen aristas intraparticion, osea quiere decir que se pudo dividir el grafo en k conjuntos independientes de vertices, en otras palabras, tenemos un grafo k-partito. Ahora si a cada k-particion se le asigna un color, resulta en un coloreo válido del grafo dado.

2.3. k-PMP en la vida real

Con k-PMP se pueden modelar aquellos problemas en los que hace falta agrupar un conjunto de cosas de forma que se minimice la relación entre estos. Por ejemplo:

ArcadeNoe
Suponemos que tenemos un arca con k sectores, en los cuales debemos repartir todos los animales de forma de minimizar el riezgo de riña entre ellos. Para esto asignamos un valor de peligrosidad a los pares de animales que tienen alguna posibilidad de pelearse entre si. En este caso los animales pueden ser representados con nodos y el valor de peligrosidad entre ellos esta dado como el peso de la arista que los une.

CampusUniversitario

Se desea construir un campus universitario en el que se edificarán k pabellones, y se pretende poder distribuir un total de n carreras en los pabellones de forma que aquellas carreras que esten directamente relacionadas se destinen a un mismo edificio. Para esto se define un valor x a la relacion entre cada carrera, el cual determina cuan relacionadas están entre ambas, siendo 0 el coeficiente que identifica la mayor relación.

TorneodeFutbol
Se tiene una cantidad n de equipos de futbol

3. Algoritmo exacto: Backtraking

3.1. Algoritmo

Para resolver el problema del k-PMP vamos a utilizar un algoritmo que se basa en la técnica de Backtracking. Este recorre todas las posibles particiones del Grafo y basta con devolver aquella que cumpla que la sumatoria de los pesos de sus aristas intraparticiones sea mínimo y tenga menos de k-Particiones. Luego se aplican algunas podas para mejorar un poco el tiempo de ejecución de esta solución, como guardar la mejor solución hasta el momento o no calcular aquellas que tenga mas de k particiones pues no pueden ser solución del problema.

Básicamente el algoritmo exacto va iterando los nodos y colocandolos en las particiones para ir verificando si voy generando una solución mejor. En el algoritmo sin podas agrego un nodo a una partición y calculo la sumatoria de las aristas intrapartición del grafo que si es la mínima hasta el momento, la guardo como solución del problema, para luego llamar recursivamente a la función con el próximo nodo. Luego de salir de la recursión saco el nodo que habia colocado en la particion y pruebo colocandolo en la próxima partición para reiterar los cáluclos y ver si puedo disminuir el peso de la sumatoria de las aristas intrapartición.

COMBINAR($list < Particion > particiones, VerticeverticeAUbicar, doublepesoAcumulado$)

```
1  for (p in particiones)
2  pesoViejo = p.getPeso()
3  pesoNuevo = pesoDelGrafoConVerticeEnParticion(p,verticeAUbicar)
4  if pesoAcumulado <= pesoGrafoConModificacion
5      then continue
6      else
7  p.setPeso(pesoNuevo)
8  pesoAcumulado = pesoGrafoConModificacion
9  combinar(particiones, proximoVertice(verticeAUbicar), pesoAcumulado)
10 pesoAcumulado = pesoGrafoConModificacion − pesoNuevo
11 p.sacarNodo(verticeAUbicar)
12 p.setPeso(pesoOld)
```

3.1.1. Podas

La poda que aplicamos al algoritmo de BackTraking es bastante sencilla pero efectiva. Básicamente me guardo la última mejor solución. Y si en el camino de recorrer las particiones, al agregar un nodo en alguna de las particiones, este genera una peor solución que la que ya tengo calculada, directamente la descarto y paso a intentar colocar este nodo en la proxima partición.

En la sección de experimentación vamos a mostrar los resultados de correr los algoritmos con o sin poda.

3.2. Código Relevante

El esta sección del código pondremos partes del código relevante.

```
class Arista {
  public:
    Arista(Vertice v, Vertice w);
    Arista(Vertice v, Vertice w, double peso);
    Vertice getVertice1();
    Vertice getVertice2();
    double getPeso();

  private:
    Vertice v;
    Vertice w;
    double peso;
};

class Grafo{
  public:
    Grafo(int n, list<Arista> aristas);
    Arista* getArista(Vertice v, Vertice w);
    Arista** getAristas(Vertice v);
    list<Arista*>* getAristas();
    int getCantVertices();

  private:
    Arista *** ady;
    list<Arista*>* aristaList;
    int n; \\ |V| := cantidad de vertices
};

class Particion{
  public:
    Particion(int nro);
    double cuantoPesariaCon(Grafo *G, Vertice vertice);
    double cuantoPesariaSin(Grafo *G, Vertice vertice);
    void agregar(Grafo *G, Vertice vertice);
    void agregarSinActualizarPeso(Vertice vertice);
    void quitar(Grafo *G, Vertice vertice);
    void quitarUltimo(Grafo *G);
    void quitarUltimoSinActualizarPeso();
    int getNro();
    double getPeso();
    void setPeso(double peso);
    list<Vertice> getVertices();

  private:
    int nro;
    list<Vertice> vertices;
    double peso;
    double pesoConVerticeX;
    Vertice verticeX;
};

void Exacto::combinar(list<Particion> &k_particion, Vertice verticeAUbicar,
    double pesoAcumulado){

  //Lo siguiente equipara a decir: Si no hay mas quimicos para cargar..
  if (verticeAUbicar == G->getCantVertices()) {
    if (pesoAcumulado < this->mejorPeso){
      this->mejorPeso = pesoAcumulado;
      this->mejorKParticion = k_particion;
```

```
      if (mostrarInfo) mostrarPotencialSolucion(this->mejorKParticion, this->
          mejorPeso);
    }
    return;
  }

  list<Particion>::iterator itParticion;
  for (itParticion = k_particion.begin(); itParticion != k_particion.end();
      itParticion++){

    double pesoOld = itParticion->getPeso();
    double pesoNew = itParticion->cuantoPesariaCon(G, verticeAUbicar);
    double difPeso = pesoNew - pesoOld;

    if (this->podaHabilitada && (this->mejorPeso <= pesoAcumulado+difPeso))
      continue;

    // 'agregar' no vuelve a calcular el peso. Ya lo calcul'o en
        cuantoPesariaCon donde se cachea.
    itParticion->agregar(G, verticeAUbicar);
    pesoAcumulado += difPeso;

    combinar(k_particion, verticeAUbicar+1, pesoAcumulado);

    pesoAcumulado -= difPeso;
    itParticion->quitarUltimoSinActualizarPeso();
    itParticion->setPeso(pesoOld);
  }


  // Si hay menos de k particiones el vertice se puede ubicar en una particion
      nueva.
  if (k_particion.size() < k) {
    Particion particionNueva(k_particion.size());
    particionNueva.agregar(G, verticeAUbicar);
    k_particion.push_back(particionNueva);
    //No hace falta fijarse que pesoAcumulado sea menor a mejorPeso porque el
        pesoAcumulado no se modifica al hacer una particion nueva sin aristas.
    combinar(k_particion, verticeAUbicar+1, pesoAcumulado);
    k_particion.pop_back();
  }

}
```

3.3. Complejidad

El analisis de complejidad lo vamos a realizar en base a la cantidad de particiones posibles en k-subconjunto. Ya que si no aplicamos ninguna poda las estaríamos generando todas para verificar cual de ellas es la mínima.
Para el primer vertices v_1, tenemos una única partición posible en donde ubicarlo, que la vamos a denominar p_1.
Para el segundo vertice v_2, lo podemos ubicar tanto en la primer partición como en una nueva llamemosla p_2.
Para el tercer nodo v_3, si ya habíamos ubicado a v_1 y v_2 en p_1, podemos colocarlo también en p_1 o bien crear una nueva partición p_2 que no sería la misma que la anterior pues esa no existiría producto que v_1 y v_2 se encuentran en p_1, pero en cambio si ubicamos a v_1 en p_1 y

a v_2 en p_2, podemos entonces ubicarlo en alguno de estos dos conjuntos o príamos crear una tercera partición p_3 (siempre y cuando k >=3) para ubicar a v_3.

Así sucesivamente hasta llegar a k vértices. Luego, para v_1 hay 1 posibble parrición, para v_2 hay 2, para v_3 hay 3, hasta v_k con $k$particiones posibles y eso es nos deja $k!$.

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

Finalmente podemos concluir que la complejidad del algoritmo es del orden de $O(k! * k^{(n-k)})$ en el peor caso, sin considerar las podas que reducen la cantidad de soluciones que computo. Para este analisis se considero que el resto de las operaciones son $O(1)$ lo cual es así pues todas son operaciones elementales.

3.4. Experimentación

3.4.1. Casos Bordes

En esta sección vamos a analizar ciertas familias de grafos en las cuales sabemos como se debería comportar el algoritmo exacto y que resultado tendría que devolver, para corroborar su correcto funcionamiento. Esto no es una demostración de su correctitud, pero es una buena forma de ver si no se nos escapa la torutga.

Las familias que vamos a analizar son las siguientes:

- Grafos con n nodos aisldos
- Grafos en forma de estrella
- Grafos que sean ciclos simples de n nodos
- Grafos completos con todos pesos 1 en sus aristas

GRAFOSCONNNODOSAISLADOS

Para esto generamos instancias de 0 a 100 nodos y no agergamos ninguna arista. Luego corrimos el algoritmo para k entre 2 y 50.
Pudimos verificar que el resultado de el algoritmo para cualquier grafo sin importar ni la cantidad de nodos, ni las particiones el resultado es siempre 0 como peso de la suma de las aristas intrapartición lo cual es efectivamente cierto pues no existen aristas.

Corroboramos que el algoritmo exacto funciona correctamente para este tipo de instancias.

GRAFOSESTRELLA

Para esto con el generador de instancias seleccionamos un nodo de forma aleatoria y lo conectamos al resto de los nodos.
Luego corrimos el algoritmo y nuevamente verificamos que se para cualquier cantidad de nodos y cualquier cantidad de particiones el peso total de la sumatoria de sus aristas es igual a 0. Es razonable pues al nodo central se lo puede ubicar en alguna particion y al resto en cualquier otra, con lo cual no hay aristas intraparticion.

$CICLOSSIMPLES(C_n)$

Generamos grafos que eran ciclos de n elementos. Y esperabamos que tambien nos de 0 la sumatoria de las aristas sin importar la cantidad de nodos ni las particiones. Pero nos olvidamos que los ciclos simples de longitud impar no eran bipartitos. Con lo cual para cualquier grafo que sea un ciclo de longitud impar si corro el algoritmo para encontrar una 2-particion mínima esta nunca va a ser 0.

Pudimos corroborar que para todos los grafos que sean circuitos simples para cualquier k mayor o igual a 3 su k-PMP es 0. Y para todos los que sean circuitos simples de longitud impar y para una 2-particion la sumatoria de sus aristas intraparticion es igual a la menor de las aristas

$K_nCONPESOS1$

Con el generador de isntancias creamos grafos completos de n elementos con todos sus pesos en 1.

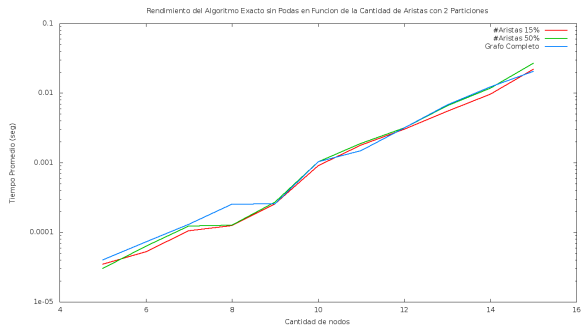
Pudimos verificar que el peso de las sumatoria de sus aristas intraparticion es:

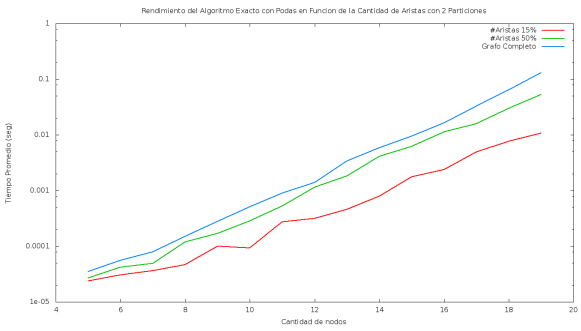
$$k * (((ndivk) * ((ndivk) - 1)/2)) + (nmodk) * (ndivk)$$

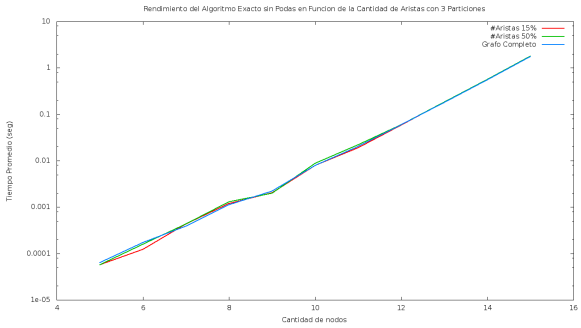
Básicamente esto es así pues en cada partición voy a tener como mínimo n div k nodos que se conectan todos entre todos por ser un grafo completo. Con lo cual esa partición va a aportar ((n div k) * ((n div k) - 1)/2) al peso total pues todas sus aristas intraparticion pesan 1. Luego tengo k particiones y por ultimo el resto de dividir a n por k son los nodos que me restan que los tengo que ubicar en alguna particion y estos se conectan con el resto de los nodos de esa particion.

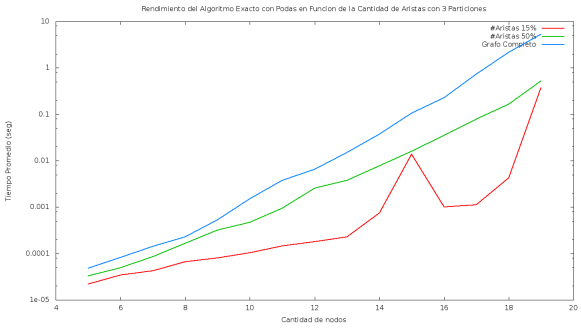
3.4.2. Rendimiento

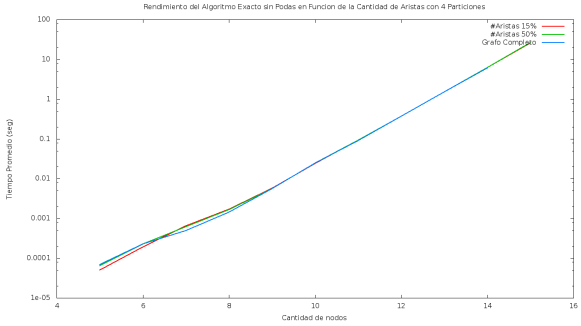
Para el análisis de rendimiento, corrimos el algoritmo variando la cantidad de particiones de 2 a 7, 100 instancias aleatoreas para cada n entre 5 y 20 de las cuales sacamos el promedio de tiempos, para las tres familias de grafos con el 15 %, 50 % y 100 % de sus aristas, tanto con el algoritmo sin podas y con podas.

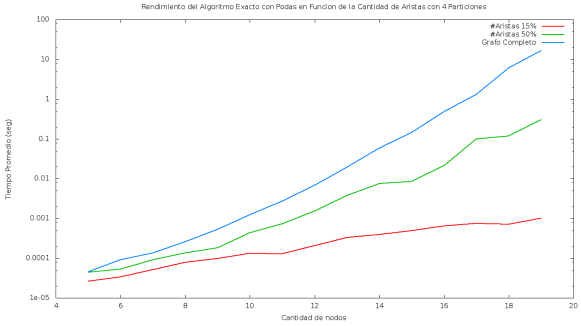


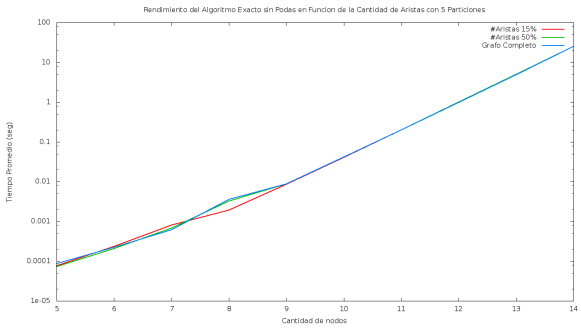


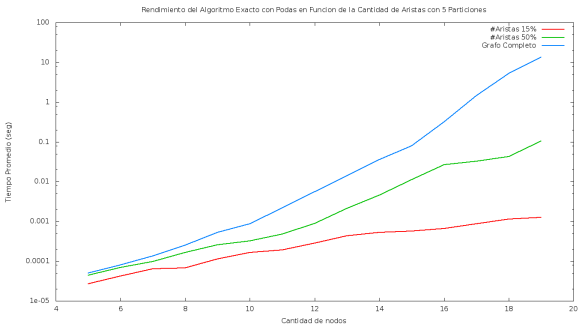

Rendimiento del Algoritmo Exacto sin Podas en Funcion de la Cantidad de Aristas con 3 Particiones
#Aristas 15%
#Aristas 50%
Grafo Completo
Tiempo Promedio (seg)
Cantidad de nodos

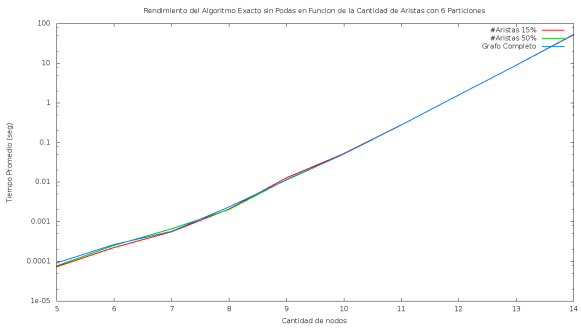

Rendimiento del Algoritmo Exacto con Podas en Funcion de la Cantidad de Aristas con 3 Particiones
#Aristas 15%
#Aristas 50%
Grafo Completo
Tiempo Promedio (seg)
Cantidad de nodos

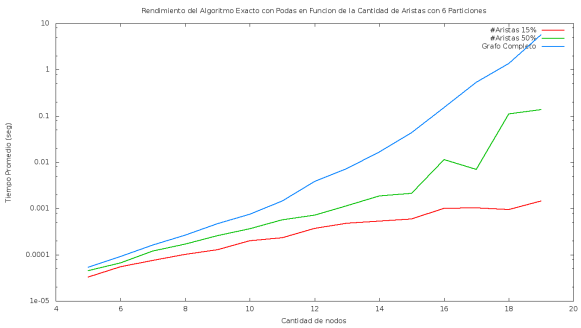

Rendimiento del Algoritmo Exacto sin Podas en Funcion de la Cantidad de Aristas con 4 Particiones
#Aristas 15%
#Aristas 50%
Grafo Completo
Tiempo Promedio (seg)
Cantidad de nodos

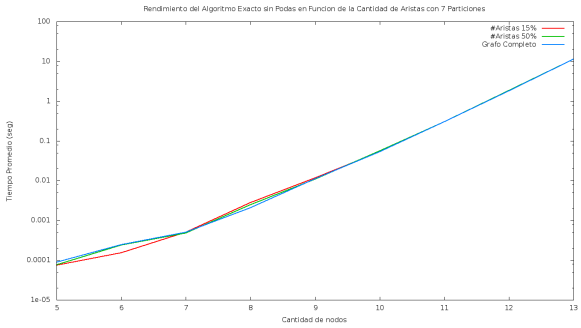

Rendimiento del Algoritmo Exacto con Podas en Funcion de la Cantidad de Aristas con 4 Particiones
#Aristas 15%
#Aristas 50%
Grafo Completo
Tiempo Promedio (seg)
Cantidad de nodos

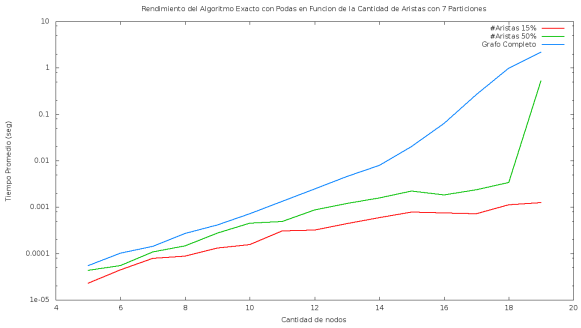

Rendimiento del Algoritmo Exacto sin Podas en Funcion de la Cantidad de Aristas con 5 Particiones
#Aristas 15%
#Aristas 50%
Grafo Completo
Tiempo Promedio (seg)
Cantidad de nodos


Rendimiento del Algoritmo Exacto con Podas en Funcion de la Cantidad de Aristas con 5 Particiones
#Aristas 15%
#Aristas 50%
Grafo Completo
Tiempo Promedio (seg)
Cantidad de nodos


Rendimiento del Algoritmo Exacto sin Podas en Funcion de la Cantidad de Aristas con 6 Particiones
#Aristas 15%
#Aristas 50%
Grafo Completo
Tiempo Promedio (seg)
Cantidad de nodos


Rendimiento del Algoritmo Exacto con Podas en Funcion de la Cantidad de Aristas con 6 Particiones
#Aristas 15%
#Aristas 50%
Grafo Completo
Tiempo Promedio (seg)
Cantidad de nodos


Rendimiento del Algoritmo Exacto sin Podas en Funcion de la Cantidad de Aristas con 7 Particiones
#Aristas 15%
#Aristas 50%
Grafo Completo
Tiempo Promedio (seg)
Cantidad de nodos


Rendimiento del Algoritmo Exacto con Podas en Funcion de la Cantidad de Aristas con 7 Particiones
#Aristas 15%
#Aristas 50%
Grafo Completo
Tiempo Promedio (seg)
Cantidad de nodos

4. Heurística Golosa

4.1. Resolución

Para nuestra heurística golosa comenzamos ordenando las aristas por peso de mayor a menor. Una vez obtenida la lista de aristas ordenadas por peso la iteramos escogiendo primero un nodo (arbitrariamente) de la arista más pesada y si la arista aun no se encuentra ubicada, se agrega a un nuevo conjunto siempre y cuando no hayamos sobrepasado la cantidad k de conjuntos creados. En caso de que ya tengamos k conjuntos o en caso de que hayamos terminado de recorrer las aristas se termina el ciclo. Ahora verificamos si la cantidad de conjuntos creados es menor a k, en caso afirmativo habremos ubicado todos los nodos con aristas en alguno de los conjuntos. Si quedan nodos sin ubicar, estos se agregan a cualquier conjunto, como ya agregamos todos los nodos de grado mayor a uno, los que restan se puede decir que tienen grado 0, por lo tanto, no importa en que conjunto sean agregados, estos no crearán aristas intraparticion y como consecuencia no sumarán peso a ningún conjunto. Si la cantidad de conjuntos creados es igual a k, actuamos de forma diferente, aquí recorremos todas las aristas, actuando solo con las que no hayan sido agregadas a ningún conjunto de la siguiente manera: verificamos cuanto peso agregaría en cada conjunto para luego realmente añadirlo al conjunto que sume menos peso, en caso de encontrar uno en el que no cree aristas intraparticion, es agregado a este sin seguir revisando los restantes.

Como se puede observar usamos dos componentes greedys en la heurística: La primera esta cuando se ordenan las aritas decrecientemente y se crean los k conjuntos a medida que se separan los nodos de las aristas de mayor peso, para que estos no generen aristas intraparticion. La segunda componente greedy se encuentra cuando al finalizar de crear los k conjuntos se recorre nodo por nodo verificando si ya se encuentra agregado y en caso negativo, verificando en que conjunto genera menos peso para finalmente agregarlo a este.

A modo de ejemplo presentamos algunas imagenes para mostrar el funcionamiento de la heuristica:

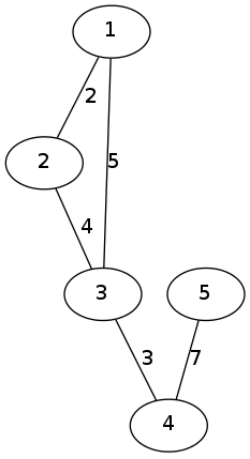


Figura 4: (1) Grafo de ejemplo

La primer imagen corresponde al grafo al que se le aplicará k-PMP (1)

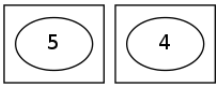


Figura 5: (2) Conjuntos 1 y 2 luego de separar los nodos de las aristas mas pesadas

Luego de ordenar las aristas escoge las de mayor tamaño y separa sus nodos en k conjuntos k=2 para este ejemplo. (2)

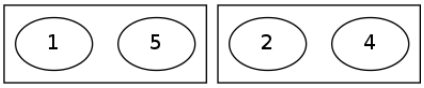


Figura 6: (3) Conjuntos luego de comenzar a ingresar los nodos restantes

Luego como ya hay 2 conjuntos creados procede a verificar nodo los nodos y agregandolos al conjunto que menos sume. En el caso del nodo 1, se agrega de inmediato al primer conjunto porque no genera arista intraparticion. Para el nodo 2 primero verifica en el 1er conjunto, aqui sumaria 2 de peso, se verifica en el siguiente conjunto, y como no suma peso, se ingresa ahí. (3)

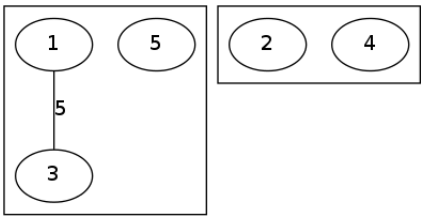


Figura 7: (4) Resultado de haber corrido la heuristica greedy al grafo (1)

Finalmente resta agregar el nodo 3 el cual genera en el conjunto uno un peso igual a 5, y en el conjunto 2 un peso igual a 7 al unirse con los nodos 2 y 4. Por lo tanto el nodo 3 es introducido en el conjunto 1. (4)

A continuación presentamos un pseudo-grafico del algoritmo

- ordenar *aristas* por peso en orden decreciente
- mientras restan *aristas*
  - si no hay k conjuntos, crear conjunto y agregar *nodo*1 de *arista* (si este no pertenece a ningún conjunto)
  - sino cortar el ciclo
  - si no hay k conjuntos, crear conjunto y agregar *nodo*2 de *arista* (si este no pertenece a ningún conjunto)
  - sino cortar el ciclo

- Si hay menos de k conjuntos creados
    - agregar las aristas que quedaron fuera a alguno de los conjuntos creados
- Sino
    - desde nodo 0 a nodo n-1
        - si el nodo no esta en ningún conjunto
            - verificar conjunto por conjunto cuanto peso generaría agregarlo a este
            - agregar el nodo al conjunto en el que genere menos peso

4.2. Analisis de complejidad

Vamos a analizar el codigo paso por paso analizando la complejidad temporal del peor caso. Lo primero que hacemos es ordernar las aristas, para esto utilizamos la funcion *sort* de la *std* que posee una complegidad temporal de $O(n.log(n))$, en este caso como se aplica a las aristas esto es $O(m.log(m))$ siendo m la cantidad de aristas. A continuación, y haciendo uso del psudo-codigo proporcionado:

- mientras restan *aristas*
    - si no hay k conjuntos, crear conjunto y agregar $nodo1$ de *arista* (si este no pertenece a ningún conjunto)
    - sino cortar el ciclo
    - si no hay k conjuntos, crear conjunto y agregar $nodo2$ de *arista* (si este no pertenece a ningún conjunto)
    - sino cortar el ciclo

Este scope tiene una complejidad $O(m)$ ya que se recorren todas las aritas, en el peor caso creando conjuntos y agregandolos a estos, pero estas dos acciones tienen una complejidad constante.

A continuacion si se crearon menos de k conjuntos se agregan los nodos que restan a uno de estos. En el peor caso esto pertenece al orden de $O(n)$ siendo n la cantidad de nodos. Si hay k conjuntos creados se procede de la siguiente manera:

- desde nodo 0 a nodo n-1 $O(n)$
    - si el nodo no esta en ningún conjunto $O(1)$
        - verificar conjunto por conjunto cuanto peso generaría agregarlo a este $O(k+n)$
        - agregar el nodo al conjunto en el que genere menos peso $O(1)$

4.3. Instancias desfavorables

4.4. Experimentacion

5. Heurística de Búsqueda Local

6. Heurística GRASP

7. Conclusiones

Referencias

[1] C++ reference, http://www.cplusplus.com/reference/algorithm/sort/

[2] C++ reference, http://www.cplusplus.com/reference/map/multimap/