DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Algoritmos Sobre Grafos

Algoritmos 3

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Ricardo Colombo	156/08	ricardogcolombo@gmail.com.com
Federico Suarez	610/11	elgeniofederico@gmail.com
Juan Carlos Giudici	827/06	elchudi@gmail.com
Franco Negri	893/13	franconegri2004@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. Comparaciones con otros algoritmos conocidos

1.1. Introducción

Para este trabajo practico se nos pide, a partir de un grafo simple $G = (V, E)$ con pesos en las aristas, encontrar la k-particion tal que minimice el peso de las aristas intrapartición.

Para ello primero intentaremos relacionar este problema con otros problemas conocidos, pensaremos varias maneras de abordarlo y emprenderemos la busqueda de algoritmos eficientes para resolverlo.

Como veremos luego, este problema es 'dificil' de resolver, por lo que para instancias grandes, el algoritmo dejará de ser viable, por lo que tambien desarrollaremos distintas heruristicas que resuelvan el problema de manera aproximada. Probaremos con una heuristica golosa, dos heruristicas de busquedas locales y a partir de esotos, un GRASP.

1.2. Entrada y salida

Todos los algoritmos tomarán como entrada, lo siguiente:

- Un entero **n** $\rightarrow$ Representará el numero de nodos del grafo G.
- Un entero **m** $\rightarrow$ Representará la cantidad de aristas del grafo.
- Un entero **m** $\rightarrow$ Representará la cantidad de conjuntos distintos que disponemos para poner los ejes.
- **m** filas donde, para cada fila i consta de 3 enteros:
  - **u v w** $\rightarrow$ donde **u** y **v** son los nodos adyacentes y **w** el peso de las aristas entre ellos.

La salida, por su parte, constará de una fila con:

- n enteros i_1 i_2 ... i_n

Donde cada i_k representa en que conjunto se encuentra el nodo k

1.3. Ejemplo

Para ejemplificar el problema a resolver, pensemos en un grafo G con 4 nodos, 5 vertices y 2 particiones.

Supongamos ademas que los nodos estan conectados de la siguiente manera.

$1-2$ con peso 2 $1-3$ con peso 3 $1-4$ con peso 3 $2-4$ con peso 1 $3-4$ con peso 2

A primera vista podríamos poner el nodo 1 separados de los nodos 3 y 4 ya que de estar juntos sumarían demaciado a la intrapartición. Por otro lado tenderíamos a poner al nodo 2 y 3 dentro de la misma partición ya que estos suman 0. Como contracara de eso ahora podemos ver que al agregar otro nodo v, sumará tanto el peso de 2 con v y 3 con v. Vemos luego que un algoritmo goloso para intentar resolver este problema puede tomar soluciónes tan malas como se quiera.

Siendo esta una instancia pequeña del problema, tras intentar algunas convinaciones, puede verse que poniendo el nodo 1 en el primer conjunto y los nodos $2, 3, 4$ en el otro, se consigue un peso total de 3.

1.4. Relación con otros problemas conocidos

En primer instancia podemos relacionar el problema de $k-PMP$ con el problema de coloreo. Supongamos que podemos encontrar un k-coloreo para los vértices del grafo G, entonces podríamos subdividir al conjunto de vértices V en k subgrupos según su color, es decir, en un mismo grupo sólo habrá vertices que compartan color. Por la definición de coloreo, dos vértices de un mismo color no pueden tener arista entre sí, por lo tanto los grupos que armamos son conjuntos independientes. Esto quiere decir que mi k-partición no tendría aristas intrapartición ya que cada partición sería un conjunto independiente,

y entonces el peso total de la misma sería 0. Y además sería un peso mínimo ya que no hay aristas con peso negativo y entonces tendríamos la solución a nuestro problema. Hasta aquí hemos visto que con un coloreo igual a k se obtiene la solución al problema, pero observemos que los conjuntos de la k-partición resultado no tienen que ser necesariamente no vacíos, lo que nos lleva a pensar que también nos bastaría conseguir un coloreo menor a k para resolver nuestro problema y ahora veremos cómo puede ser esto posible. Dado un k'-coloreo con $k' < k$, por lo dicho anteriormente podemos armarnos k' subgrupos de vértices agrupandolos por color, los cuales serán conjuntos independientes, es decir, no existirán aristas que incidan en dos nodos de un mismo grupo. Sin embargo, ya no me queda ningún vértice para meter en algún grupo y el problema me pide k particiones, por lo que me estarían faltando otras $k - k'$ particiones más que agregar a mi k-partición. Pero si recordamos nuestra observación que decía que las particiones no deben ser necesariamente no vacías, entonces podríamos agregar a nuestra k-partición $k - k'$ particiones de vértices vacías, con lo cual no estaríamos agregando ninguna arista intrapartición. Entonces mis k' particiones iniciales, por lo visto previamente, no tienen ninguna arista intrapartición y los conjuntos vacíos que agregué posteriormente claramente tampoco tienen aristas intrapartición, por lo que he llegado nuevamente a una k-partición de peso 0, la cual es solución de mi problema y además es óptima.

Por otro lado, dada una solución al problema de k-PMP de peso estrictamente mayor a 0 para un grafo G determinado, podemos afirmar que no existe un k-coloreo para ese grafo. Si existiera un k-coloreo para dicho grafo, entonces, por lo probado en el anterior párrafo, también existiría una k-PMP de peso 0 para tal grafo, lo cual es absurdo ya que partimos de una k-PMP de peso estrictamente mayor a 0.

2. Algoritmo Exacto

2.1. Introducción

Dado un grafo simple G = (V , E) se quiere buscar la solución con un algoritmo exacto a el problema de la k- partición, para esto se realiza un algoritmo con la técnica de backtracking para obtener la solución exacta del problema.

En los puntos siguientes se detalla la solución del mismo definiendo la implementación del algoritmo, las podas y como afectan a la solución en los tiempos.

2.2. Desarrollo

Para explicar el desarrollo de este algoritmo iremos mostrando las partes del algoritmo armando el algoritmo final paso a paso, para comenzar el desarrollo es necesario definir previamente cada una de las estructuras que se utilizan en el mismo. Sabemos que el grafo esta representado por un conjunto de vértices, cada uno de los mismos tiene una etiqueta en este caso es un numero desde 1 hasta n (siendo n la cantidad de vértices del grafo), a su vez tenemos un conjunto de tuplas que representan los ejes en el grafo cada coordenada tiene un numero del 1 al n representando que vértices conecta el eje, a nuestro grafo lo representamos en una matriz de adyacencias a la que denominamos adyacencias en el grafo, donde en cada posición se tiene 0 si las aristas no están conectadas o el valor del peso del eje en caso que estén conectados.

Una solución será representada en un conjunto de k subconjuntos, donde cada uno de estos subconjuntos contendrá los números de los vértices, todos disjuntos de a pares (ósea que no tenemos el mismo numero de vértice en dos subconjuntos), y el peso del conjunto será la suma de cada uno de los pesos de los subconjuntos, donde el peso del subconjunto se entienden a la suma de los pesos de las aristas que conecten dos vértices del mismo subconjunto , además definimos que una k partición A es mejor que otra B si el peso total es menor, es decir si A es mejor que B entonces peso(A) ¡peso(B). Nuestro solFinal en principio va a ser el conjunto donde todos los vértices estén en el primer subconjunto. La idea fundamental del algoritmo de Backtracking es que en cada llamado recursivo intentara insertar un vértice a la solución que esta armando, en caso de no tener una mejor solución tratara de insertar ese vértice en otro subconjunto que no haya probado, así hasta haber probado todos los subconjuntos y sin ir mas lejos si se realiza este ejercicio estaremos realizando todas las combinaciones de los vértices en los k subconjuntos.

De la siguiente manera:

```
1: Si Puedo insertar
2:    para cada i desde 1 hasta cantidadSubConjuntos:
3:       agrego el numeroVertice al subconjunto i en SolParcial
4:       backtracking ( solParcial , solFinal, numeroVertice+1,k,adyacencias)
5:       saco el elemento que agregue al ultimo conjunto
6: Si Puedo insertar
7: devuelvo False
```

Algorithm 1: backtracking(solParcial,solFinal,numeroVertice,cantidadSubConjuntos)

Ahora de esa manera lo único que estamos haciendo es probar todas las combinaciones , pero no estamos quedándonos con la mejor k - partición para esto debemos definir una función que nos diga cuando una solución es mejor que otra y además debemos saber si ya insertamos todos los vértices.

Para lo cual definimos una función que devuelve un numero indicando 0 si inserte todos los vértices llegando a otra solución, luego tengo que ver si es mejor que la que tengo hasta el momento y 1 si tengo que seguir insertando vértices. Quedándonos de la siguiente manera:

Con este algoritmo estaremos probando todas las combinaciones de todos los vértices en los k subconjuntos, en cada paso insertamos un vértice a un subconjunto y llamamos a la recursión con el siguiente vértice. una vez que vuelva de la recursión quitara el vértice del conjunto donde lo ingreso y seguirá el ciclo insertándolo en otro subconjunto y llamando a la recursión de nuevo.

```
1: SI numeroVertice==cantidadDeVertices
2:     devolver 0
3: SI NO
4:     devolver 1
```
Algorithm 2: numero check(adyacencias, solParcial,solFinal, numeroVertice,cantidadVertices)

```
1: entero resultadoCheck = check(adyacencias,solParcial,solFinal,numeroVertice,cantidadVertices)
2: si resultadoCheck = 0
3:     Remplazo mi solFinal por solParcial
4: Si NO
5:     para cada i desde 1 hasta cantidadSubConjuntos:
6:         agrego el numeroVertice al subconjunto i en SolParcial
7:         backtracking ( solParcial , solFinal, numeroVertice+1,k,adyacencias)
8:         saco el elemento que agregue al ultimo conjunto
9: devuelvo False
```
Algorithm 3: backtracking(solParcial,solFinal,numeroVertice,cantidadSubConjuntos,adyacencias, cantidadVertices)

2.3. Podas

Para mejorar los tiempos del algoritmo es necesario realizar algún tipo de validación intermedia para no probar casos que no nos van a llevar a una solución optima, por ejemplo cuando estamos insertando vértices a la solución que ya tiene mas peso que nuestra mejor solución o cuando nos quedan una cantidad de vértices es menor estricta a la cantidad de subconjuntos vacíos, estas podas son las que realizamos en nuestro algoritmo agregándolas a la función check al Backtracking de la siguiente manera:

```
1: SI peso(solParcial) ¿peso(solFinal)
2:     devolver 2
3: SI numeroVertice==cantidadDeVertices
4:     devolver 0
5: Si cantidadDeCajasVacias(solParcial) ¿cantidadVertices
6:     devolver 2
7: devolver 1
```
Algorithm 4: numero check(adyacencias, solParcial,solFinal, numeroVertice,cantidadVertices)

De esta manera el algoritmo va a realizar las podas en las llamadas recursivas intermedias antes de insertar todos los vértices descartando casos que no son útiles para llegar a una solución optima en menor tiempo.

Para improvisar una mejora en la poda al comparar los pesos de nuestras soluciones parcial y final, lo que realizamos es generar otra solución final asignando cada uno de los vértices de manera creciente a cada uno de las k subconjuntos diferentes y si esta otra solución es mejor a la solución que tiene todos los nodos en un solo conjunto la tomamos como solución final para nuestras comparaciones. De esta manera podrían tomarse otro tipo de soluciones para luego obtener la solución final de menor peso y podar mas ramas del Backtracking.

2.4. Complejidad

```
1: entero resultadoCheck = check(adyacencias,solParcial,solFinal,numeroVertice,cantidadVertices)
2: si resultadoCheck = 0
3:    Remplazo mi solFinal por solParcial
4: si resultadoCheck = 2
5:    devuelvo false porque mi solParcial es peor que mi solFinal o tengo mas cajas vacías que vertices
   para poner l
6: Si NO
7:    para cada i desde 1 hasta cantidadSubConjuntos:
8:       agrego el numeroVertice al subconjunto i en SolParcial
9:       backtracking ( solParcial , solFinal, numeroVertice+1,k,adyacencias)
10:      saco el elemento que agregue al ultimo conjunto
11: devuelvo False
```

Algorithm 5: backtracking(solParcial,solFinal,numeroVertice,cantidadSubConjuntos,adyacencias, cantidadVertices)

3. Heuristica Golosa Constructiva

3.1. Idea general

Un primer intento para conseguir una solución a este problema es utilizando un algoritmo goloso. La idea del mismo es sencilla, numeramos los nodos de 1 a n, y luego tomando de a uno los agregamos a alguno de los conjuntos de 1 a k intentando que sume a la solución el menor peso posible.

Mas formalizado el algoritmo quedará de la siguiente manera:

========= ALGORITMO ========

Claramente este algoritmo no devuelve la solución exacta, y como se verá mas adelante, la solución que devuelve puede estar tan lejos como se quiera de la optima, lo que lo hace un algoritmo no muy bueno.

Como ventajas de este algoritmo, puede verse que tiene una complegidad muy baja de $O(n^2k)$, por lo que podría usarse como una primera cota superior, aunque grosera, que puede resultar util.

3.2. Problemas del Algoritmo Goloso

Como ya adelantamos, la solución para este algoritmo no siempre da una solución exacta, y puede ser tan mala como se quiera. Esto surge principalmente de darle un orden a los nodos, como mostraremos en el siguiente ejemplo.

Supongamos que tenemos un grafo como el que se muestra en la figura y $K = 2$

A –1– B
/ 100 100
/ C

Es claro que la mejor solución posible es poner en uno de los conjuntos a A y B y en el otro a C, así la suma intrapartición es 1.

Sin embargo, supongamos que nuestro algoritmo goloso toma como primer nodo al nodo A, dado que no hay otros nodos, lo agrega en cualquiera de los dos conjuntos y se obtiene peso 0. Ahora supongamos que el algoritmo toma el nodo B, si lo pone en el mismo conjunto que el nodo A obtiene peso 1, si lo pone en el otro conjunto obtiene peso 0, asi que lo pone en el otro conjunto. Pero ahora falta agregar el nodo C, y agregandolo en cualquiera de los dos conjuntos se obtiene peso 100. Asi que la solución para k-PMP que encontrará el algoritmo goloso será 100. De ahí se desprende que cambiando ambos pesos 100 por cualquier valor puede obtenerse una solución tan mala como uno quiera.

Sin embargo, puede observarse otra cosa de este algoritmo, si se hubiese tomado el nodo C como primer nodo o como segundo nodo, se hubiera llegado a la solución optima. De aquí surge la idea de que si se eligieran diferentes ordenes para los nodos y se corriera el algoritmo para cada uno de estos ordenes, podría obtenerse diferentes cotas, y con un poco de suerte en alguna de ellas no sucederá este caso que acabamos de ver.

Esta idea la utilizaremos mas adelante para el GRASP, correremos con distintos ordenes de nodos el algoritmo goloso de manera tal de obtener en cada una de estas iteraciones una respuesta diferente y posiblemente mejor que la anterior.

4. Heuristica de Busqueda Local

5. Introducción

En esta sección se implementarán heuristicas de busqueda local para tratar de resolver el problema de $k - PMP$, intentando con diferentes metodos para alcanzar una solución y diferentes vecindades.

Para ello, partiendo de una solución generada al azar, se intentará a travez de sucesivas iteraciones analizar sierta vecindad para intentar mejorar la solución existente y aproximarse mas a una 'buena' solución en un tiempo aceptalble.

Por lo tanto debemos tener en cuenta de no hacer las vecindades ni muy grandes, ya que esto puede llevar a una perdida de performance, ni muy pequeñas, ya que esto puede llevar a que el algoritmo explore solo una pequeña cantidad de soluciones y devuelva una solución muy alejada del optimo.

Como primera idea para una vecindad tomaremos cada nodo del grafo, vemos cuanto peso agrega en la suma intraconjunto en que se encuentra, lo quitaremos de este conjunto e intentamos meterlo en todos los demas, viendo si en alguno logra minimizar esta suma. En caso afirmativo, lo sacamos de su antiguo conjunto y lo ponemos en el nuevo. Realizamos esto hasta que deja de ser posible mejorar la solución y en este punto la devolvemos.

Esta heuristica, como se verá en el apartado de testing, devuelve resultados vastante aproximados, al compararlo con la solución exacta y con las otras heuristicas.

Otra vecindad que plantearemos será buscar el nodo que mas peso esta generando en la suma intra-partición, quitarlo de la partición donde se encuentra y agregarlo a alguna otra.

Finalmente se implementará una tercera busqueda local que quite dos nodos que estan en una misma partición e intente buscar alguna otra donde los mismos sumen un menor peso intrapartición.

Los algoritmos escritos de manera mas formal serán así:

========== ALGORITMOS ==========

Ahora aquí analizaremos las complegidades de los diferentes algoritmos.

Para el primero, cada paso de busqueda local tendrá una complegidad de peor caso de $O(n(n + nk + n^2))$. Ahora si k es mayor a n quiere decir que hay mas subconjuntos disponibles que nodos, lo que llevaría a una solución trivial donde cada nodo va en un subconjunto diferente y la solución para $k - PMP$ sería 0. Luego podemos acotar a k por n con lo que se obtendría una complegidad igual a $O(n^3)$

Para el segundo algoritmo, por cada iteración del algoritmo la complegidad será $O(n^2)$ para calcular el nodo con mayor peso del grafo. $O(kn)$ para determinar si existe una mejor partición donde este nodo pueda estar y, en el caso de que exista, $O(n^2)$ para quitarlo de la partición anterior y agregarlo a la nueva. Luego, nuevamente acotando k por n, se obtiene una complegidad para cada paso de la iteración de $O(n^2)$.

Para el tercer algoritmo ==========(falta hacer que ande)==========

6. GRASP

7. Idea

Para la metaeuristica de GRASP, tomaremos el algorimto goloso previamente implementado y lo combinaremos con las diferentes busquedas locales tambien previamente implementadas.

La idea es la siguiente, para cada paso del algoritmo, corremos primero una busqueda local con un orden aleaotrio en sus nodos. Este orden aleatorio, como ya hemos demostrado en el apartado del algoritmo goloso, podrá ir produciendo diferentes respuestas, cada una con una mejor o peor aproximación a la respuesta exacta.

Luego a la solución obtenida por el algoritmo goloso se le aplicarán una o varias de las busquedas locales implementadas en un intento de aproximar mas aun la respuesta al optimo.

Como primer criterio de corte el algoritmo se correrá un numero finito de veces, que será determinado en el momento de experimentación.

Como segundo criterio de corte se realizarán estos mismos pasos hasta que luego de un numero x a determinar de intentos no haya sido posible mejorar la solución. En este punto se entrega la mejor respuesta obtenida hasta el momento.

A continuación se formaliza de manera mas precisa el algoritmo:

===== ALGORITMO =====

8. Experimentación

Para esta seccion de experimentacion compararemos los tiempos de ejecución y los resultados del algoritmo de GRASP contra la solución exacta obtenida por backtracking.

9. Apéndice

9.1. Medicion de los tiempos

Para este tp como trabajamos bajo el lenguaje de programacion C++, decidimos calcular los tiempos utilizando 'chrono' de la libreria standard de c++ (chrono.h) que nos permite calcular el tiempo al principio del algoritmo y al final, y devolver la resta en la unidad de tiempo que deseamos.

10. Aclaraciones

10.1. Medicion de los tiempos

Para este tp como trabajamos bajo el lenguaje de programacion C++, decidimos calcular los tiempos utilizando 'chrono' de la libreria standard de c++ (chrono.h) que nos permite calcular el tiempo al principio del algoritmo y al final, y devolver la resta en la unidad de tiempo que deseamos.