Algoritmos y estructuras de datos III

Trabajo Práctico Nº3

De Sousa Bispo Mariano	389/08	marian_sabianaa@hotmail.com
Grosso Daniel	694/08	dgrosso@gmail.com
Livorno Carla	424/08	carlalivorno@hotmail.com
Raffo Diego	423/08	enanodr@hotmail.com

Junio 2010

Índice

Introducción

Este trabajo tiene como objetivo la aplicación de diferentes técnicas algorítmicas para la resolución de tres problemas particulares, el cálculo de complejidad teórica en el peor caso de cada algoritmo implementado, y la posterior verificación empírica.

El lenguaje utilizado para implementar los algoritmos de todos los problemas fue C/C++

1. Situaciones de la vida real

El problema del Clique Máximo puede usarse como modelo para diversas situaciones de la vida real en ambitos muy variados. Por un lado tenemos los problemas que involucren personas (como nodos) y las relaciones entre ellos (los ejes) en distintas materias. El ejemplo mas cotidiano (al menos para todos nosotros) es el de las redes sociales, y las .ªmistades.enˆtre las distintas personas. En este caso, puede ser útil para hacer pruebas de mercado, como dar productos gratis para promocionarlos, y dado que el costo de cada producto puede ser elevado se trata de entregar la menor cantidad, asegurandose la máxima promocion posible, entonces se busca el grupo de .ªmigos"mas grande intentando que todos se enteren del producto en cuestion.

Podemos suponer que el Doctor Malito, némesis del conocido y carismático agente ingles Austin Powers, quiere infectar a la población con un virus. Supongamos que el virus es de transmición aérea, y dado el enorme costo de fabricacion del virus, solo se pudieron fabricar un par de cientos de ejemplare. El Dr. Malito utilizaría una modificacion de Max_Clique para elejir sus blancos para que la probabilidad de contagio sea mayor.

2. Algoritmo exacto

2.1. Explicación

El Algoritmo busca todas las formas de armar una clique utilizando la técnica de backtracking. Para esto, inicia la clique una vez desde cada vértice probando todas las combinaciones que lo incluyan, agregando vértices tal que forman un completo con los ya incluídos. Se necesita empezar una vez por cada nodo ya que la solución final podría no incluir el nodo inicial. De esta

forma, se genera un árbol de backtracking teórico para cada vértice inicial. Mediante podas, evita recorrer el árbol por completo, siendo la solución final la máxima de las cliques encontradas. Como el algoritmo busca todas las cliques del grafo, la solución es la clique máxima del grafo.

2.1.1. Optimizaciones

Dado que se trata de un algoritmo de backtracking, la optimzación se basa en podar las ramas en las que estamos seguros que no va a aparecer el óptimo. Para esto tenemos que poder predecir, dado un estado actual, si es posible mejorar el óptimo encontrado hasta el momento.

Por un lado, podamos las ramas que no forman un grafo completo, ya que no es solución.

Por otro lado, evaluamos en cada paso del algoritmo la cantidad de vértices que falta explorar. Es decir, calculamos el tamaño de la clique máxima que podríamos formar considerando los vértices que ya estan incluídos en la solución actual. Si la cantidad de vértices que todavía no fueron evaluados más la cantidad de vértices ya pertenecientes a la clique actual es menor a la cantidad de vértices de la clique máxima encontrada hasta el momento, no tiene sentido seguir explorando esa rama ya que el tamaño de la clique máxima que se puede encontrar por ese camino es menor al tamaño de la máxima encontrada. Por este motivo, podamos esta rama.

Además, para cada vértice que inicia la clique se intenta agregar los de mayor numeración tal que forman un completo. Por lo tanto, se evita repetir combinaciones. Supongamos que tenemos una clique de tres vértices, siendo estos el $< 1, 2, 3 >$. Con esta optimización, nunca han de analizarse los casos $< 2, 3, 1 >$; $< 2, 1, 3 >$; $< 3, 1, 2 >$ y $< 3, 2, 1 >$.

2.2. Detalles de la implementación

Almacenamos las relaciones entre los vértices en una matriz de $n \times n$, donde n es la cantidad de vértices. Cada posición (i, j) de la matriz contiene un *uno* si existe la arista (i, j) y un *cero* en caso contrario. De esta forma se le asigna un número a cada vértice.

A continuación, se muestra el pseudocódigo del algoritmo exacto.

```
exacto(matriz_adyacencia,n)
    solucion ← ∅
    for i to n
        buscar clique máxima desde el vértice i
        if  tiene más vértices que solucion
            solucion ← clique encontrada
return #(solucion)
```

El algoritmo de backtracking recorre todos los vértices. Para cada uno de estos verifica si es adyacente con todos los vértices de la solución actual y si todavía no pertenece a la misma. Si es así lo agrega y repite este procedimiento (avanza). Si no, significa que recorrió todos los vértices y no logró formar una clique mayor a la encontrada, por lo que comienza a retroceder.

Cuando retrocede, saca el último vértice v que agregó a la solución y prosigue la búsqueda desde el vértice siguiente a v en numeración. Si no hay un vértice que se pueda agregar, es decir, si ninguno de los siguientes forma una clique, el algoritmo sigue retrocediendo.

2.3. Complejidad temporal

El algoritmo de Max_Clique exacto utiliza la técnica de backtracking, con lo que genera un árbol donde cada rama es una posible solución, y esta se desarrolla hasta que encuentra una solucion o el algoritmo mediante una optimizacion "se da cuenta"que no hay forma de encontrar el clique óptimo y la "poda". Ya que las optimizaciones podan constantemente las ramas donde los cliques que no lleguen a ser el óptimo, se hace muy difícil encontrar el *peor caso* del algoritmo, que sería el caso donde las podas sean las mínimas posibles, con una cantidad máxima de cliques máximos, con lo que decidimos plantear un caso hipotético donde el árbol de backtracking se genere sin podas, verifícando para cada nodo i las posibles cliques que contengan a i, y todos los nodos posibles entre i y n. Como dijimos, esto es un caso hipotético, donde el algoritmo tiene un comportamiento menos eficiente, y de esta forma lo que nos da es una cota un tanto *gruesa* con respecto a la real, pero que a fines prácticos nos da una idea de que aún en el peor caso, el algoritmo será un tanto mas eficiente respecto a la cota calculada.

Sea G un grafo con conexo, con n nodos y relaciones máximas, es decir, siendo m la cantidad de ejes del grafo, $m = n(n-1)$. $Esdecir, G = K_n$ Al tratarse de un caso hipotético asumiremos que el algoritmo no efectua podas ni utiliza las optimizaciones que ayudan a terminar el algoritmo antes que recorra todos los nodos, es decir, que solo verificará para cada nodo i, con $0 < i < n$, los cliques posibles que contengan a i, junto con todos los nodos posibles entre i y n, y terminara cuando i sea igual a n

De esta forma, para cada nodo i generará su arbol de cliques correspondiente,y así tendremos un arbol de $n - i$ factorial nodos, que son todas las posibles cliques máximas de n-i nodos, con i como raiz.

Como este procedimiento se repite con i desde 1 a n nos queda planteada una sumatoria: $\sum_{i=1}^{n} i! = 1 + 2 + 3! + ... + n! \leq n.n!$

3. Heurística constructiva

3.1. Explicación

Como Primera Heurística, en este caso constructiva, desarrollamos un algoritmo goloso para resolver el problema `MAX-CLIQUE` de manera aproximada.

El mismo funciona de la siguiente manera:

Sea *grados* un arreglo de tamaño n, donde n es la cantidad de vértices del grafo. En cada posición j $\forall 1 \leq j \leq n$ del arreglo está el grado correspondiente al vértice j. Para construir una clique ordenamos *grados* en forma decreciente. El primer vértice del arreglo, es decir, el de mayor grado del grafo se considera parte de la solución final del algoritmo. Para completar la clique recorremos *grados* en forma completa, y cada vértice que forma un completo con la solución parcial se agrega a la misma. Al terminar de recorrer *grados* el algoritmo termina siendo la solución parcial, el resultado final.

Al ser un algoritmo goloso, en este problema como en tantos otros, no devuelve necesariamente el óptimo. Particularmente, la clique está condicionada al vértice de mayor grado, y no necesariamente la solución óptima lo contiene.

3.2. Detalles de la implementación

A continuación, se muestra el pseudocódigo del algoritmo de heurística constructiva.

```
constructivo(matriz_adyacencia,n)
    grados[n] ← ordenar_grados(matriz_adyacencia)
    solucion[n]
    solucion[0] ← grados[0]
    tamanyo ← 1
    for i to n
        if ¬solucion[i]
            completo ← forma_completo(solucion, i, matriz_adyacencia)
            if completo
                solucion[i] ← true
                tamanyo ← tamanyo + 1
return tamanyo
```

- `ordenar_grados:` En la implementación, el arreglo *grados* es de tipo tupla donde la primer componente representa el vértice y la segunda el grado. Dicho arreglo está ordenado según la segunda componente en forma decreciente. Lo ordenamos con el algoritmo de Quick Sort de *STL*.

  Para setear el grado de un vértice i tenemos un contador inicializado en *cero*. Recorremos la columna de la matriz de adyacencia correspondiente a dicho vértice e incrementamos el contador por cada posición (i, j) igual *uno*.

- `forma_completo:` Para saber si agregar un vértice v determina una solución al problema debemos verificar que forme un completo con los vértices ya incluídos. Para esto recorremos todos los vértices del grafo y para cada uno que pertenezca a la solución parcial chequeamos que sea adyacente a v. Si esto ocurre podemos agregar el vértice v y agrandar la clique.

3.3. Complejidad temporal

El algoritmo empieza inicializando el arreglo *grados* lo que tiene un costo de n^2 ya que para cada vértice recorre la columna correspondiente en la matriz de adyacencia (diseñada como un arreglo de arreglos).

Como mencionamos anteriormente, *grados* es ordenado con un algoritmo de QuickSort dado por la librería estándar de C++. El costo del algoritmo es CUADRATICO APARENTEMENTE, OJO AHI LA VIDA FIJATE Q DICE LA STL.

El algoritmo constructivo, una vez realizadas las operaciones antes mencionadas, entra a un ciclo for que itera desde 0 hasta n. En cada iteración, debe en primera instancia, analizar una guarda if. De ser $falsa$, procederá a la siguiente iteración del ciclo (teniendo así costo constante). De ser *verdadera* (es decir, el vértice i no forma parte de la solución parcial), analizará si puede formar una nueva clique mayor, ahora con el vértice i (en nuestro pseudo-código, la función $forma\_completo$ es la encargada de analizarlo). Con el resultado de $forma\_completo$ la iteración del for principal analiza una guarda if más de costo constante (en el peor caso realiza dos asignaciones y una suma). Sabemos entonces que el costo de este ciclo será como mínimo n. Analizaremos a continuación el costo de $forma\_completo$, para concluir el costo total del algoritmo.

$forma\_completo$, como ya dijimos antes, recorre todos los vértices del grafo analizando que, los vértices que pertenecen a la solución parcial, sean adyacentes al que queremos agregar. Esta función itera por todos los vértices del grafo, teniendo así un costo lineal a la cantidad de vértices.

Como el ciclo for del algoritmo constructivo, en el peor caso, llamaría una vez por cada una de las n iteraciones a la función $forma\_completo$, el costo del ciclo entonces es cuadrático a la cantidad de vértices.

Tenemos entonces en el algoritmo constructivo, la inicialización del arreglo con costo de n^2, el QuickSort de STL, costo AOISJDOAISJDN^2 y el ciclo for también con costo a lo sumo cuadrático. La complejidad asintótica del problema viene dada entonces por la suma de las complejidades anteriormente descriptas. Podemos afirmar que el costo es O(()n^2), ya que O(()$n^2 + n^2 + n^2$) = O(()$3 * n^2$) = O(()n^2).

4. Busqueda local

4.1. Explicación

La heurística de busqueda local actua a partir de una solución inicial S, en este caso, a partir de la solución dada por la heurística constructiva. El algoritmo busca en la vecindad de la solución dada, $N(S)$, una solución mejor que ésta. Si no encuentra ninguna mejor, nos encontramos en un óptimo local (de la vecindad) que tomamos como solución del algoritmo. La vecindad $N(S)$ que elegimos en este problema es el conjunto de soluciones tales que no tienen uno y sólo uno de los vértices pertenecientes a S, es decir, $S \setminus \{v\} \cup L$ donde L es un conjunto de vértices tal que $u \in L \iff S \setminus \{v\} \cup \{u\}$ forma un completo.

Para revisar la vecindad, sacamos un nodo de la solución óptima actual e intentamos agreagar los vértices que no pertenecen a la clique. Luego, comparamos el tamaño de la clique que logramos contruir de esta manera con el tamaño de la clique correspondiente a la mejor solución vista de la vecindad. Si esta nueva solución es mejor, es decir, el tamaño de la clique es mayor al tamaño de la actual (mejor de la vecindad), dicha solución pasa a ser la mejor de la vecindad. Una vez revisada toda la vecindad comparamos la mejor solución encontrada en dicha vecindad con la mejor solución encontrada hasta el momento. Si es mejor, actualizamos el óptimo actual.

4.2. Detalles de la implementación

A continuación, se muestra el pseudocódigo del algoritmo de heurística de búsqueda local.

```
busqueda_local(solucion,tamanyo,matriz_adyacencia,n)
    grados[n] ← ordenar_grados(matriz_adyacencia)
    tam_actual ← tamanyo
    tam_mejor_vecindad ← tamanyo
    actual[1..n] ← solucion[1..n]
    mejor_vecindad[1..n] ← solucion[1..n]
    mejore ← true
    while i < n and mejore
            mejore ← false
            for j to n
                sacar_de_clique(actual, i)
                agrandar_clique(actual, matriz_adyacencia)
                if tam_actual > tam_mejor_vecindad
                    tam_mejor_vecindad ← tam_actual
                    mejor_vecindad[1..n] ← actual[1..n]
                else
                    reconstruir(actual)
            if tam_mejor_vecindad > tamanyo
                mejore ← true
                tamanyo ← tam_mejor_vecindad
                solucion[1..n] ← mejor_vecindad[1..n]
return tamanyo
```

- `sacar_de_clique`: Esta función setea en $falso$ la posición i del arreglo *actual*, es decir, excluye el vértice i de la solución actual. Además, decrementa el tamaño de la clique, variable *tam_actual* y resetea una variable *nodo* que indica si despues logra agregar algún vértice, esto sirve para reconstruir la solución en caso de ser necesario (conseguir un tamaño de clique igual al tamaño de la clique desde la que partió).

- `agrandar_clique`: Esta función se encarga de buscar entre los vértices que actualmente no pertenecen a la clique e intenta agregarlos (agrega todo vértice que forma un completo con los ya pertenecientes), con el objeto de conseguir una de tamaño mayor. Por otro lado, setea *nodo* con el valor del último vértice que agrega.

- `reconstruir`: En el caso donde el tamaño de la clique que logro contruir es igual al tamaño de la clique inicial, como queremos obtener el mejor de la vecindad, reconstruimos la clique anterior y continuamos, es decir, eliminamos *nodo* y volvemos a agregar i.

4.3. Complejidad temporal

En un pricipio el algoritmo inicializa el arreglo de los *grados* y lo ordena, lo que tiene un costo de n^2. Luego, obtiene la solución inicial mediante la heurística constructiva que como ya vimos tiene también un costo de n^2 e inicializa los arreglos *actual* y *mejor_vecindad* con costo lineal.

La función *sacar_de_clique* y *reconstruir* tienen costo constante ya que consta sólo de indexaciones, asignaciones y otras operaciones elementales.

La función *agrandar_clique* tiene costo n^2 porque recorre todos los vértices y para cada uno de los que todavía no pertenece a la clique, verifica si puede agregarlo. Para esto, recorre nuevamente todos los vértices y corrobora que sea adyacente a cada uno de los pertenecientes a la clique.

La complejidad final del algoritmo es n^4 porque el ciclo `while` itera a lo sumo n veces y por cada una de estas el ciclo `for` itera exactamente n veces. En cada iteración del `for` hay una llamada a la función *sacar_de_clique* y *agrandar_clique* las cuales tienen un costo de n^2. Eventualmente hay una llamada a *reconstruir* lo que no altera la complejidad, ya que su costo es constante.

5. Tabu-Search

5.1. Explicación

Finalmente implementamos una metaheuristica, es decir, una heurística que guía otra heurística, en este caso la búsqueda local del ejercicio anterior.

El `Tabu-Search` permite evitar que la heurística de búsqueda local se estanque en óptimos locales cuando en realidad fuera de la vecindad existía una solución óptima global (mejor que la local). Para lograrlo, permite al algoritmo perseguir una solución peor que la mejor obtenida mediante búsqueda local, por una cantidad máxima de iteraciones. Pasada esta cantidad, consideramos que el algoritmo ya buscó lo suficiente y la mejor solución encontrada hasta el momento debe ser la óptima.

Para no revisar las vecindades que se revisaron anteriormente (que son muy cercanas a la vecindad actual), cada vez que decidimos movernos a otra vecindad porque tenemos un nuevo aspirante a óptimo (más allá de que sea peor que la mejor solución que encontramos hasta el momento, pero lo llamamos así por su similitud con el mismo en la búsqueda local) prohibimos revertir el cambio que hicimos para llegar del aspirante anterior al nuevo, o sea, prohibimos volver a agregar el vértice que sacamos.

Esto lo implementamos mediante un arreglo de vértices (los índices representan los vértices), donde para cada uno guardamos la iteración en el que el mismo deja de ser tabú (el algoritmo lleva la cuenta de las iteraciones). Luego, cuando revisamos las vecindades de un aspirante, evitamos aquellas donde la modificación implica agregar un vértice tabú.

Los parámetros que indican cuanto tiempo una variable queda tabú y cuantas veces se puede iterar sin encontrar un nuevo óptimo antes de cortar el algoritmo, los buscamos empíricamente viendo los resultados obtenidos con distintos parámetros, y son bla y bla respectivamente.

CAMBIAR LOS PARAMETROS!!!

5.2. Detalles de la implementación

A continuación, se muestra el pseudocódigo del algoritmo de heurística de búsqueda tabú.

```
busqueda_tabu(solucion,tamanyo,matriz_adyacencia,n)
    grados[n] ← ordenar_grados(matriz_adyacencia)
    tam_actual ← tamanyo
    actual[1..n] ← solucion[1..n]
    tabu[1..n] ← 0
    lista_tabu ← vacia
    cant_iter ← max(tamanyo − 2, 3);
    mejore ← true
    while i < n and mejore
            mejore ← false
            for n − 1 to 0
                v1 ← grados[i]
                sacar_de_clique(actual, i)
                agrandar_clique(actual, matriz_adyacencia)
            for nodo ∈ lista_tabu
                formar_completo(actual, matriz_adyacencia, nodo)
                if actual[nodo]
                    eliminar(lista_tabu, nodo)
                    tabu[nodo] = 0
    agregar(lista_tabu, v1)
    restar_tabu(tabu, lista_tabu)
            if tam_actual > tamanyo
                mejore ← true
                tamanyo ← tam_actual
                solucion[1..n] ← actual[1..n]
return tamanyo
```

- `sacar_de_clique:` Esta función setea en $falso$ la posición i del arreglo *actual*, es decir, excluye el vértice i de la solución actual. Además, decrementa el tamaño de la clique, variable *tam_actual* y resetea una variable *nodo* que indica si despues logra agregar algún vértice, esto sirve para reconstruir la solución en caso de ser necesario (conseguir un tamaño de clique igual al tamaño de la clique desde la que partió).

- `agrandar_clique:` Esta función se encarga de buscar entre los vértices que actualmente no pertenecen a la clique e intenta agregarlos (agrega

todo vértice que forma un completo con los ya pertenecientes), con el objeto de conseguir una de tamaño mayor. Por otro lado, setea *nodo* con el valor del último vértice que agrega.

- `forma_completo:` Para saber si agregar un vértice v determina una solución al problema debemos verificar que forme un completo con los vértices ya incluídos. Para esto recorremos todos los vértices del grafo y para cada uno que pertenezca a la solución parcial chequeamos que sea adyacente a v. Si esto ocurre podemos agregar el vértice v y agrandar la clique.

- `restar_tabu:` Para cada vértice tal que tiene tabú mayor a *cero* decrementa la cantidad de iteraciones que va a permanecer tabú. Si al decrementarlo deja de ser tabú lo elimina de *lista_tabu*.

5.3. Complejidad temporal

En un pricipio el algoritmo inicializa el arreglo de los *grados* y lo ordena, lo que tiene un costo de n^2. Luego, obtiene la solución inicial mediante la heurística de busqueda local que como ya vimos tiene también un costo de n^4 e inicializa los arreglos *actual* y *tabu* con costo lineal.

La función *sacar_de_clique* tiene costo constante ya que consta sólo de indexaciones, asignaciones y otras operaciones elementales.

La función *agrandar_clique* tiene costo n^2 porque recorre todos los vértices y para cada uno de los que todavía no pertenece a la clique, verifica si puede agregarlo. Para esto, recorre nuevamente todos los vértices y corrobora que sea adyacente a cada uno de los pertenecientes a la clique.

El primer ciclo `for` itera exactamente n veces y en cada una de estas iteraciones hay una llamada a la función *sacar_de_clique* y *agrandar_clique* las cuales tienen un costo de n^2, por lo que el costo del ciclo es n^3. El otro ciclo `for` itera tantas veces como elementos tenga la lista tabú, es decir, a lo sumo n veces. Cada iteración de este ciclo tiene un costo lineal por la llamada a la función $formar\_completo$, por lo que este ciclo `for` cuesta n^2. Finalmente, se deduce que la complejidad final del algoritmo es n^4 porque el ciclo `while` itera a lo sumo n veces y en cada una de estas iteraciones ejecuta ambos ciclos `for` (con complejidades n^3 y n^2). Cabe destacar que además del ciclo `while` la heurística de tabú search llama a búsqueda local una vez, la que tiene la misma complejidad que el `while`, sin modificar así, el costo asíntotico del análisis de peor caso.

6. Observación

EXPLICAR PORQUE TABU VA POR GRADOS Y LOCAL NO

7. Resultados

7.1. Parametros de la heurística tabú

7.2. Comparacion de tiempos

7.3. Comparacion de calidad

8. Mediciones

- Para contar la cantidad aproximada de operaciones definimos una variable inicializada en *cero* la cual incrementamos luego de cada operación. Preferimos contar operaciones en vez de medir tiempo porque a pesar de que es aproximado el resultado, el error es siempre el mismo y así podemos hacer una mejor comparación entre las instancias.
  Midiendo tiempo, el error para cada instancia varía, ya que es el sistema operativo el que ejecuta nuestro programa, al "mismo tiempo"que otras tareas.

9. Compilación y ejecución de los programas

Para compilar los programas se puede usar el comando `make` (Requiere el compilador `g++`). Se pueden correr los programas de cada ejercicio ejecutando `./secuencia_unimodal`, `./ciudad` y `./prision` respectivamente.

Los programas leen la entrada de stdin y escriben la respuesta en stdout. Para leer la entrada de un archivo `Tp1EjX.in` y escribir la respuesta en un archivo `Tp1EjX.out` ses puede usar:
`./(ejecutable) <Tp1EjX.in >Tp1EjX.out`

Para contar la cantidad de operaciones: `./(ejecutable) count`. Devuelve para cada instancia el tamaño seguido de la cantidad de operaciones de cada instancia. En el ejercicio 3 también se puede contar la cantidad de operaciones en función de la cantidad de llaves de la siguiente manera:

`./prision count_llaves`. Devuelve para cada instancia la cantidad de llaves/puertas seguido de la cantidad de operaciones correspondientes.