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. De esta forma, se genera un árbol de backtracking teórico para cada vértice inicial.

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.

2.2. Optimización

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 $< 3, 2, 1 >$.

2.3. Complejidad temporal

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.

4. Busqueda local

4.1. Explicación

La heuristica de busqueda local actua a partir de una solución inicial S, en este caso, a partir de la solución dada por la heuristica constructiva. El algoritmo busca en la vecindad de la soluición 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.

4.2. Complejidad temporal

5. Tabu-Search

5.1. Complejidad temporal

6. Resultados

6.1. Parametros de la heurística tabú

6.2. Comparacion de tiempos

6.3. Comparacion de calidad

7. 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.

8. 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.