17
2 ej.

# UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

# CARACTERIZACION DE DISTINTAS IMPLEMENTACIONES DEL ALGORITMO DE GRADO MINIMO PARA MATRICES GRANDES



**T E S I S**

QUE PARA OBTENER EL TITULO DE

M A T E M A T I C O

P R E S E N T A

**ERIK SCHWARZ NOVOA**

DIRECTOR DE TESIS: ING. JORGE GOMENDIETA

MEXICO, D. F. 1999

VNIVERSIDAD NACIONAL
AVTÓNOMA DE
MEXICO

**MAT. MARGARITA ELVIRA CHÁVEZ CANO**
**Jefa de la División de Estudios Profesionales de la**
**Facultad de Ciencias**
**Presente**

Comunicamos a usted que hemos revisado el trabajo de Tesis: Caracterización de distintas implementaciones del algoritmo de grado mínimo para matrices grandes

realizado por Erik Schwarz Novoa,

con número de cuenta 9450373-3, pasante de la carrera de matemáticas

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

| | | |
|---|---|---|
| Director de Tesis Propietario | ING. JORGE GIL MENDIETA | Jorge Gil |
| Propietario | DR. MAXIMILIANO ANTONIO DIAZ DE LA PEÑA | |
| Propietario | DR. FERNANDO BRAMBILA PAZ | |
| Suplente | ACT. CLAUDIA CARRILLO QUIROZ | Claudia Carrillo Q. |
| Suplente | DR. JAVIER PAEZ CARDENAS | |

**Consejo Departamental de** MATEMATICAS

MAT. CESAR GUEVARA BRAVO DIAZ

CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

*A mi familia le dedico el tiempo, el esfuerzo y los frutos de mi educación, de la cual este esfuerzo forma parte.*

## AGRADECIMIENTOS

Quiero agradecer a varias personas que me ayudaron mucho a lo largo de mi carrera y que lo continuaron haciendo a lo largo de la tesis. En primer lugar al director de esta tesis, el ingeniero Jorge Gil Mendieta, quien confió en mí durante los proyectos que hicimos juntos. También el Dr. Javier Páez Cárdenas con su consejo y compañía. A Paola Aguilar-Alvarez Zerecero y familia les debo infinitamente el tiempo y espacio que me dieron; así como la paciencia que me tuvieron mientras contaba mis frustraciones durante la codificación.

A Adolfo Cortés Cárdenas y Adrián Farías les agradezco los consejos que guiaron este trabajo. A Katy le agradezco muchísimo el apoyo en el último esfuerzo.

A los sinodales que le dedicaron el tiempo y esfuerzo para tratar de entender y desenmarañar mi texto (así como a quienes no lo hicieron). A Alejandro Ruiz y Jorge Castro del Laboratorio de Redes.

Agradezco a la FUNDACION TELMEX por prestarme sus instalaciones durante mi participación como becario.

Se agradece a la Dirección General de Asuntos del Personal Académico de la UNAM, DGAPA, que patrocinó parcialmente el Proyecto IN-310296 "Sistema Automático para el Análisis de Redes Sociales" del cual formé parte como becario.

A los miembros de ATL-INFINITA por tenerme paciencia mientras me distraía de mis deberes por terminar este trabajo; además del tiempo de su gente y de su material: en especial a José Luis Perea, Rafael Muñoz y Jorge Pérez.

Y en general a mis amigos y familiares.

## CONTENIDO

## LISTA DE ILUSTRACIONES

## LISTA DE TABLAS

## MOTIVACIÓN

En el Laboratorio de Redes del Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas, el Ing. Jorge Gil Mendieta y su grupo trabajan en la aplicación de la teoría de gráficas al estudio de estructuras sociales. En varios artículos se han reportado los diversos avances en esta área. Un problema al que se han enfrentado son las redes grandes con más de mil nodos y densidades alrededor de 50%. Por lo mismo, se ha visto la necesidad de desarrollar un sistema que analice redes con un número grande de nodos (ie. superior a mil). En la actualidad existen algunos sistemas comerciales mediante los cuales se pueden analizar las principales características de las redes Un sistema muy usado es UCINET IV Versión 1.04 [GS]. Sin embargo, hacer un sistema que incorpore todos los algoritmos que se necesitan en el caso particular del análisis de redes sociales, así como permitirle al usuario aprovechar la facilidad de manejo que es posible a través de WINDOWS, fue, en gran parte, lo que motivó el desarrollo de este software.

En la investigación sobre la red política en México, se han llegado a presentar matrices de más de 10,000 vértices[GS]. Si se piensa en buscar rutas mínimas entre vértices, separación en cliques, vértices centrales, entre otros elementos de análisis, se ve que los algoritmos de fuerza bruta pueden extenderse

demasiado y consumir recursos excesivos.

Buscando algoritmos que resolvieran los problemas planteados por J. Gil y su grupo, caímos en la cuenta de que convendría, en primer lugar, hacer un preprocesamiento en la gráfica y ordenar los vértices[ADD]. Encontramos varios artículos que sugerían cómo hacerlo. Cada uno difería en algo con respecto a los anteriores, y decidimos que podríamos instrumentar algunos para compararlos. Variamos la representación del orden de los vértices, así como la estrategia de selección y la interacción de la selección con el resto de la gráfica; esto dio como resultado las seis distintas opciones que se presentan en el software. En los siguientes capítulos se describirá el algoritmo, la estructura del código, las observaciones que hemos hecho a lo largo de este tiempo, y el desempeño de nuestras propuestas.

## DEFINICIÓN

Cuando se tienen enormes volúmenes de información, hacer búsquedas se facilitaría mucho si la información estuviera ordenada de alguna forma, por ejemplo el orden alfabético en un directorio telefónico. De una forma similar, tratamos de ordenar la base de la matriz con el criterio del grado mínimo. Pero como nuestras búsquedas no son lineales (como sería buscar en una lista), también tenemos que considerar que se minimicen otros parámetros, como sería la búsqueda dentro del renglón particular. Por eso nuestro ordenamiento ordena con el algoritmo de grado mínimo, buscando reducir el ancho de banda.

Ordenar una matriz significa encontrar una permutación de la base, que permita obtener una matriz nueva formada tomando como primer renglón, el primer elemento de la permutación, el segundo renglón será el segundo elemento de la permutación, y así aplicándole la permutación a cada renglón. Evidentemente como la matriz original expresaba relaciones entre los vértices (representadas por las entradas de la matriz), al mover los renglones, se tienen que mover las columnas mediante el mismo ordenamiento, de forma que, aunque el ordenamiento se aplica sobre los vértices, representados por los renglones y columnas, el ordenamiento se aplica a las posiciones de la matriz. El ordenamiento se aplica con el fin de que si las entradas no-

**cero** de la matriz original parecían arbitrariamente colocadas, después del ordenamiento se encuentren con algún patrón, o al menos concentradas.

Si pensamos en una gráfica muy pequeña como la de la *ilustración 1*, veremos que la forma "natural" de numerar los vértices es como se hizo ("ABCD"). Sin embargo, ¿habrá algo más formal que justifique, por qué es más natural esa forma que "BADC"? La respuesta es sí, como se puede observar al analizar las matrices derivadas de los distintos ordenamientos.

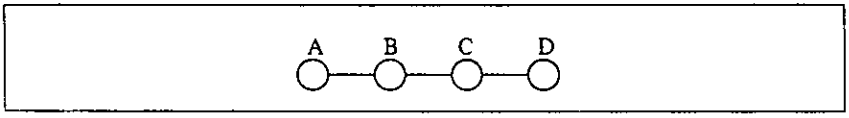



Ilustración 1: Ejemplo de una gráfica simple y pequeña

| | | | |
|---|---|---|---|
| 1 | 1 | | 1 |
| 1 | 1 | | |
| | | 1 | 1 |
| 1 | | 1 | 1 |

Tabla 1: Representación de la gráfica de la *Ilustración 1* con el ordenamiento "BADC"

| | | | |
|---|---|---|---|
| 1 | 1 | | |
| 1 | 1 | 1 | |
| | 1 | 1 | 1 |
| | | 1 | 1 |

Tabla 2: Representación de la gráfica de la *Ilustración 1* con el ordenamiento "ABCD"

Como se puede observar, la información en la segunda matriz está más condensada; mientras que en que en la primera aparentemente podrían aparecer **1's** dondequiera. En este ejemplo queda claro como con un ordenamiento puede modificar la estructura de una matriz, simplemente con ordenar los vértices y como esto puede reducir el ancho de banda.

En esta etapa buscamos principalmente evaluar dos distintas representaciones del orden entre vértices, para ir construyendo el sistema sobre bases sólidas. Teóricamente esta comprobación no es de mucha importancia por el orden de los algoritmos, sin embargo existen variaciones entre la teoría y la práctica. Por eso mismo, incluimos ambas formas para compararlas. Es claro que cada una tiene diferentes virtudes y defectos, y lo importante es determinar sus cualidades para emplearlas en futuras situaciones. Por ejemplo, en el caso de la versión final del sistema de análisis de gráficas, podría resultar que alguna representación fuera excesivamente superior a la otra; en ese caso sólo se le incluiría a ella. O quizás una

sobresale en los casos pequeños o medianos y la otra en los grandes (entendemos, por ejemplo, que pequeño es menor a cien vértices, y que mediano es hasta mil); entonces, el programa podría decidir cuál representación utilizar según el caso. Incluiremos los resultados de las comparaciones en una sección posterior.

Este programa quiere proporcionarle al sistema, que habrá de desarrollarse en proyectos futuros, una permutación de los vértices que haga que la matriz de adyacencia tenga menor ancho de banda. Dicho programa ha de leer una matriz de un archivo, procesarla y entregar un ordenamiento. El resultado del ordenamiento puede copiarse a disco en un archivo de texto que contendrá cada número de vértice en una línea, y el tiempo hasta el final, todo tal y como aparece en la ventana de resultados. Para lograr esto, basta con seleccionar la opción "Guardar a disco" que aparece apretando, sobre la ventana de resultados, el botón secundario del mouse. Este archivo quedará listo para utilizarse en otras aplicaciones y se guardará con el nombre y ubicación que se escoja en la pantalla que se abrirá para tal fin. Cuando se tengan más módulos dentro del software, este ordenamiento podrá aplicarse directamente en la matriz, según se vea si conviene más adelante. Es probable que no sea necesario reorganizar la matriz si se considera que a cada llamada a una celda, se le puede aplicar la permutación en ambas coordenadas para recalcular la celda buscada. El criterio principal para determinar si vale la pena desde un principio permutar las columnas y renglones, es el estimado de cuántas veces se irán a acceder las celdas de la matriz. Por ejemplo, si se va a considerar el problema de encontrar el árbol de costo mínimo que cubra una gráfica, conviene ya tener ordenada la matriz (ya que éstos pueden tardar $O(|E| \log |V|)$ [PS], que considerando una densidad de 50%, tenemos

que se puede tardar $O(n^2 \log n)$, pero si se van a hacer, por decir algo, $O(n)$ búsquedas de tamaño $O(\log n)$ pues el costo resultaría $O(n \log n)$ que comparado con el ordenamiento en $O(n^2)$ no valdría la pena.

El funcionamiento del programa es muy sencillo. El usuario deberá escoger únicamente la representación del orden sobre los vértices, y la estrategia de selección y actualización en la gráfica, seleccionar el archivo de donde se tomará la matriz (ésta será la matriz de adyacencia, cuando se quiera ordenar una gráfica ), y determinar el tamaño de la matriz. Existe la opción de calcular una matriz al azar y ver la matriz actual para facilitar las pruebas y familiarización del usuario con el sistema. En el caso de que el usuario especifique el archivo que contenga la matriz, si el archivo es demasiado pequeño con respecto al número de columnas indicado, la matriz se rellenará con **0's**. Si por el contrario fuera demasiado grande, se tomará la porción inicial del archivo que sea suficiente para llenarla. Una vez inicializada la matriz, el usuario dará la orden de comienzo con el botón "Ordenar" y esperará el resultado.

El tiempo variará dependiendo de la matriz y también dependiendo de las opciones seleccionadas. Además también está sujeto a las capacidades del sistema y de los recursos disponibles en ese momento. El espacio que se utilice también habrá de cambiar según las opciones escogidas, y principalmente por el tamaño indicado de matriz. Más adelante se mostrará de qué forma las opciones harán depender el tiempo y el espacio consumidos por el programa.

## ESTRATEGIAS DE SELECCIÓN

El problema de encontrar el ordenamiento que minimice el *Fill-in* es un problema de los que se conoce como **NP-COMPLETOS**, como se ha documentado en la literatura [Y]. Esto quiere decir que (suponiendo la conjetura **P =/= NP**), a medida que el problema crece, en cantidad de vértices, el tiempo necesario para calcular una solución crece en forma no-polinomial. En estos casos lo que se hace es utilizar funciones heurísticas para resolver el problema, implicando un sacrificio de exactitud por rapidez [PS].

El algoritmo de grado mínimo se ha usado frecuentemente para problemas de este tipo; principalmente porque se ha visto en problemas de prueba que minimizando el *Fill-in* en cada paso, se va a reducir el *Fill-in* en su totalidad [A,DR]. A este tipo de algoritmos se les conoce con el nombre de *greedy algorithms*, *algoritmos glotones*. El principio detrás de ellos es minimizar localmente, con la esperanza de alcanzar un mínimo global. Evidentemente, como el lector puede imaginarse, es fácil encontrar situaciones en las que esta estrategia falle [DR]. En este

algoritmo, se escoge en cada paso el vértice de grado mínimo. El orden en el que se escogen los vértices, es el ordenamiento ofrecido como mejora para reducir el *Fill-in*.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|
| X | X | X | X | | | | | | | |
| X | X | X | | X | | | | | | |
| X | X | | X | X | | | | | | |
| X | | X | X | X | | | | | | |
| | X | X | X | X | X | | | | | |
| | | | | X | X | X | | | | |
| | | | | | X | X | X | X | X | |
| | | | | | | X | X | X | | X |
| | | | | | | X | X | | X | X |
| | | | | | | X | | X | X | X |
| | | | | | | | X | X | X | X |

Tabla 3: Contraejemplo de optimalidad. El mejor ordenamiento sería **[11, 10, 9, 8, 1, 2, 3, 4, 5, 6, 7]**... pero según el Algoritmo de Grado Mínimo, sería **[6, 11, 10, 9, 8, 1, 2, 3, 4, 5, 7]**

En este programa, como ya había mencionado antes, se utilizan tres estrategias para seleccionar al vértice de grado mínimo: *sin indistinguibilidad, con indistinguibilidad* y *con indistinguibilidad y actualización*.

*INDISTINGUIBILIDAD:* Se dice que dos vértices, **x** y **y**, son indistinguibles cuando **ady(x)+{x} = ady(y)+{y}** (nota: el "más +" aquí representa unión entre conjuntos). En nuestro caso, al considerar a los vértices auto-adyacentes, la fórmula

se reduce a **ady(x) = ady(y)**. De aquí reconocemos fácilmente dos condiciones necesarias para la indistinguibilidad: *ambos vértices deben tener el mismo grado* y *ambos vértices deben ser adyacentes entre sí*. La primera idea la empleamos para verificar la *indistinguibilidad*.

La importancia de poder determinar cuáles vértices son indistinguibles, es, que sin mayor problema, podemos eliminarlos todos juntos, sin tener que hacer más cálculos o modificaciones que los que se necesitarían si los eliminaramos de uno en uno. Pero desgraciadamente verificar esto es costoso. Por lo que recurrimos a una función heurística.

*FUNCIÓN HEURÍSTICA:* Notemos que la suma: $\mathbf{h(x)} = \sum_{i \in ady(x)} i$ es igual para vértices indistinguibles, pero si dos vértices tienen la misma suma no necesariamente son indistinguibles. Esta función **h** es nuestra función *hash*, con ésta identificamos una condición más, indispensable para la *indistinguibilidad*. Cuando los dos primeros criterios se satisfagan, al igual que ambos vértices compartan el mismo valor de **h( )**, los consideraremos indistinguibles. Pero desgraciadamente no son suficientes.

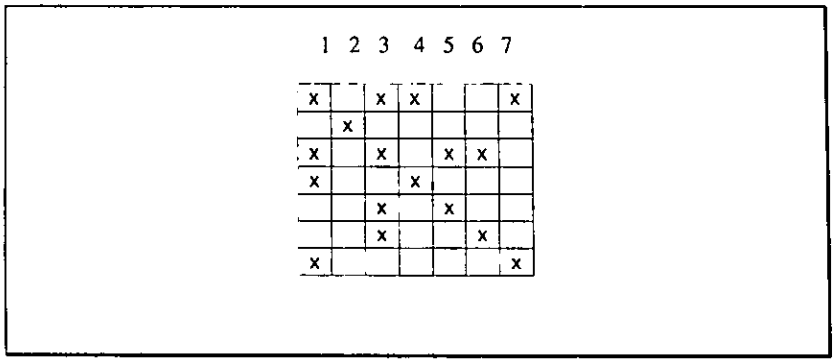



Tabla 4: Contraejemplo de suficiencia para la *indistinguibilidad* Muestra una matriz donde **h(1) = h(3) = 15, grado(1) = grado(3) = 4, 1** y **3** son adyacentes, pero no son *indistinguibles*.

Recordándolos:

- deben tener el mismo grado,
- deben tener la misma función **h**.

Tabla 5: Condiciones necesarias de *indistinguibilidad*

*ACTUALIZACIÓN*: La *actualización* se refiere a modificar la matriz de adyacencias una vez escogido un vértice. La modificación que se hace es siguiendo el modelo de la gráfica de eliminación, que dice que al eliminar el vértice **x**, sin pérdida de generalidad, $\forall y, z \in ady_G(x), y \in ady_{G'}(z)$. Es decir, que los vértices que estaban a distancia **2** en **G** y que pasaban en ese camino mínimo por **x**, se van a encontrar a distancia **1** en **G'** al eliminar **x** de **G**. Como vemos, la *actualización* y la *indistinguibilidad* son conceptos independientes que se complementan.

Utilizamos los tres criterios, ya que producen diferentes ordenamientos. En algunos casos se llegan a producir mejores ordenamientos con un criterio que con otro. Como se ve en la literatura, no se puede demostrar que el ordenamiento mejore el ancho de banda, pero empíricamente se ha visto que sí ayuda en la mayoría de los casos[A,DR].

## ESTRUCTURAS DE DATOS

En este programa se han usado dos formas distintas de representar el orden entre los vértices, *la cola binaria de prioridad* y *los listados dinámicos*. La instrumentación de las funciones básicas de cada una de dichas estructuras de datos se tomó de Data Structures and Algorithm Analysis [W]. Aunque para el problema en particular que nosotros tratamos, tuvimos que agregar nuevas funciones. Esto lo hicimos heredando las propiedades y métodos básicos además de incluir algunos nuevos.

Traté, tal cual se recomienda en los libros consultados sobre la programación orientada a objetos, que hubiera los menos casos de *amistad* ("*friend function, friend class*") entre clases, y que cada una se hiciera cargo de sí misma. En el caso de *la cola binaria de prioridad,* fue mediante la herencia que se le incorporó la funcionalidad deseada a nuestra clase. Pero en el caso de *los listados dinámicos*,

utilizamos la clase principal como un contenedor. Permitimos así, que fuera a través de las funciones propias de la clase, que se modificaran y revisaran las listas.

La idea detrás de *la cola binaria de prioridad* es un árbol binario completo (de ahí su nombre). Pero afortunadamente no se tienen que instrumentar las aristas y los nodos padre e hijos, como tradicionalmente se haría al representar dicho árbol como una gráfica. En cambio, se puede identificar dicho árbol mediante un arreglo [W]. Las aristas quedan representadas implícitamente gracias a la estructura existente entre los números naturales, que por cierto, son los que sirven de índices para el arreglo.

| [nivel 1] | [nivel | 2] | [ni | ve | l | 3] | ... |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |

Tabla 6: Aplanamiento de un árbol binario respetando niveles

La estructura queda establecida mediante las operaciones de multiplicar y dividir por 2. El nodo principal es el nodo en la posición **1**. Sus nodos hijos son los **2** y 3. A su vez, los nodos hijos de **2** son **4** y **5**; mientras que los de **3** son **6** y **7**. Como se puede ver fácilmente, para saber quién es el padre del nodo **x** se tiene que hacer **x div 2.** Y sus hijos son **2x** y **2x + 1**. En nuestro caso particular, el nodo padre es menor o igual que sus hijos. Por lo mismo, *el mínimo*, en todo momento, es el nodo **1**.

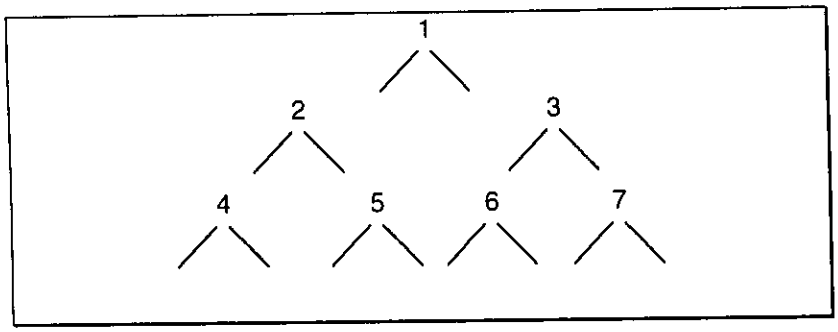



Ilustración 2: Indexación de los nodos de un árbol binario. Forma en que se asignan las posiciones del arreglo a un árbol binario para representarse en una cola binaria de prioridad. Nótese que los hijos de cada nodo **n** son **2n** y **2n+1**.

La instrumentación en listados dinámicos utilizó todo un arreglo de los mismos. Se construyó un arreglo de tamaño igual al máximo número de vértices permitidos. En cada entrada del arreglo se "encajó" un listado dinámico, para que guardara todos los vértices que coincidieran en grado con el índice de la posición en el arreglo. Así, se ve que habrá muchas casillas vacías, pero la inserción de vértices es sumamente rápida y no es demasiado el espacio perdido ya que cada lista vacía no ocupa mucho espacio.

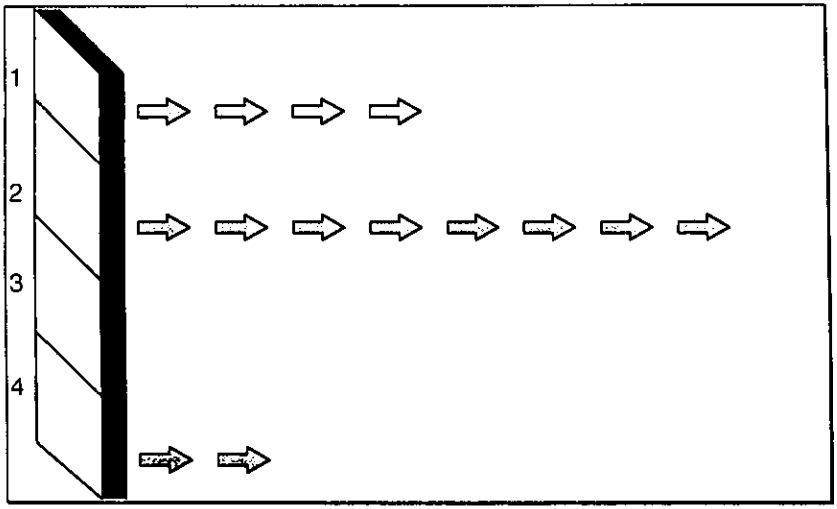



Ilustración 3: Orden de los vértices en listas dinámicas. Gráfica que ilustra la representación del orden parcial sobre los vértices. En este caso resulta el orden ser total. La figura de la izquierda representa el arreglo donde cada casilla contiene una lista con todos los vértices cuyo grado coincide con el del índice de la lista en el arreglo. Las flechas indican los nodos, cada uno indicando qué nodo le sigue. Se ve que no existen vértices de grado 3, y hay 2 vértices de grado 4.

Considerando las posibles instrumentaciones de la matriz de adyacencia, llegamos a la conclusión de que convendría utilizar una matriz de bytes. El problema es de suma importancia, como lo hizo notar el Dr. Díaz, ya que en el caso de mil vértices, estamos hablando de almacenar $O(n^2)$ bits. Según la recomendación encontrada en los libros de consulta, la matriz instrumentada es de *bytes*; ya que, aunque con el uso de bits se reduce entre una constante el espacio de almacén, el tiempo de acceso se incrementa de forma considerable, así como el tamaño del código.

Otra posibilidad hubiera sido empleando listados dinámicos de las aristas. Esto traería los beneficios esperados al complicar la estrategia de búsqueda, pero implicaría una capacidad inicial superior de memoria para almacenar la matriz. Por ejemplo, con la matriz se utilizarían *ca.* $n^2/2$ *bytes*, mientras que utilizando listados

dinámicos serían *ca.* **(p+i) * |E|** (**p** que son el tamaño de los apuntadores, **i** por el tamaño del entero que expresa a qué vértice va la arista, y **|E|** que en nuestro caso es $n^2$*.50). Así que notamos que el ahorro en el tamaño es de **(p+i)**- veces (las constantes **p** e **i** dependen del sistema, en este caso **p = 8, i = 4**).

## ALGORITMOS

Los tiempos necesarios en cada operación cambian mucho de una instrumentación a la otra. *Los listados dinámicos* son mucho más eficientes en los casos grandes, ya que la inserción se hace en tiempo constante ***O(1)***, al igual que la extracción. Por lo mismo, la construcción es en tiempo ***O(n)***. Sin embargo, todas las operaciones ocupan a los apuntadores. Estas operaciones son costosas comparadas con el acceso a un arreglo, como en el caso de *la cola binaria de prioridad*. Sin embargo, esta última tiene inserciones y extracciones en tiempo ***O(log n)***. Y por lo mismo, la creación toma tiempo ***O(n log n)***. Básicamente instrumentamos ambas formas, para comparar su desempeño en problemas de distintas dimensiones.

En nuestra instrumentación de la cola binaria, se necesita **O( log N )** para insertar un elemento. Porque tiene que subir a lo más **log N** niveles y subir cada uno toma **O( 1 )**, porque es comparar el grado con el de arriba y permutarlos (si fuera necesario). Crear la cola de prioridad toma **O( N * log N )**. Porque son **N** elementos que tienen que insertarse. Quiero hacer notar que la creación del *binary-heap* no

siguió este patrón obvio, sino que se metieron todos los vértices arbitrariamente y luego se procedió a ordenarlos desde adentro. Así nos ahorramos el tiempo en que cada vértice se inserta desde abajo. Esto, aunque no baja la complejidad, sí reduce las constantes. Quitar el mínimo ocupa también **O( log N )**, ya que consiste en tomar el mínimo, **O( 1 )**, y reinsertar el último elemento, **O( log N )**.

La instrumentación de los listados dinámicos es muy sencilla, ya que se elimina e inserta cada elemento (independientemente de su ubicación) en tiempo **O(1)**, debido a que consiste solamente modificar valores de los apuntadores. El resultado final de la creación es **O( N )**.

| Tipo | Creación | Inserción | Extracción |
|---|---|---|---|
| Cola Binaria de Prioridad | O( n log n ) | O( log n ) | O( log n ) |
| Listados Dinámicos | O( n ) | O( 1 ) | O( 1 ) |

Tabla 7: Orden de algunas operaciones importantes (selección)

Crear e ir actualizando la tabla de indistinguibilidad podría tomar hasta **O(N*N)**. Así como revisar los sacados con el mismo grado, porque en el peor de los casos todos los vértices tendrán el mismo grado, y no habrá dos indistinguibles. Por lo

mismo, aunque estas dos operaciones son tardadas, es difícil que se den problemas que exijan tanto tiempo, ya que se reducirá esta cota tan pronto como aparezcan vértices con distintos grados, que es lo más frecuente.

Guardar los elementos en el arreglo tomará **O( 1 )**; así que el algoritmo tiene como cota **O( N*N )**, aunque en la mayoría de los casos será una cota con mucha holgura, como se explicó en el párrafo anterior.

| | **Cálculo de grados** | **Creación** | **Actualización** |
|---|---|---|---|
| Matriz Simétrica | O(n*n) | O(n*n) | O(n*n*n) |

Tabla 8: Orden de algunas operaciones importantes ( matriz )

INTERFASE ENTRE EL USUARIO Y EL PROGRAMA

El programa está escrito en C++ para **C++ Builder** de Borland, el cual permite escribir programas para WINDOWS95. Fue instrumentado bajo el esquema de la *programación orientada a objetos*. Lo que, por cierto, facilitó la escritura del programa; básicamente porque se codificaron tres algoritmos distintos ya que los restantes tres son copia de éstos, modificando la instrumentación de la estructura de búsqueda y algunos detalles mínimos. Salvo los detalles, la variación quedó encapsulada dentro del objeto que contenía el ordenamiento de los vértices.

Las dos formas en que se representaron los vértices y sus grados son: una cola binaria de prioridad, y un arreglo de listas dinámicas. Esta última se puede entender como una escalera, donde en el escalón **n** se encuentran los vértices con el grado **n**. Mientras que la primera es un arreglo que simula un árbol binario (*ver sección de Fundamentos*).

Las tres estrategias de búsqueda ( *sin indistinguibilidad, con*

*indistinguibilidad, con indistinguibilidad y actualización*) se basan en la idea del algoritmo de grado mínimo. Sin embargo, solamente en la última de las opciones, se instrumenta la idea de la gráfica de eliminación[ADD]. La *indistinguibilidad* se refiere a eliminar consecutivamente vértices con las mismas adyacencias; mientras que la *actualización*, a modificar el conjunto de aristas incidentes en los vértices adyacentes al que se elimina en cada paso. Estas nociones se desarrollarán en el capítulo de *Fundamentos*.

El programa se compone de una pantalla principal, de una ventana de diálogo para abrir o seleccionar archivos, y una más para cerrar y guardar el ordenamiento. En la primera se presenta un cuadro de texto, donde se muestran los resultados, así como dos grupos de opciones, a saber: uno para escoger la representación del orden, y otro para determinar la estrategia de selección. Dentro de cada uno, las opciones son mutuamente excluyentes. Hay también un botón para iniciar la operación del algoritmo seleccionado y otro para salir. Además hay un pequeño cuadro de edición, que recibirá el número de columnas que se desea contemplar en la matriz, y otro que recibirá un número **d** que será el porcentaje de **1**'s que contenga la matriz, cuando ésta se calcule al azar. Como se puede ver, la operación de esta etapa es muy sencilla, gracias al ambiente WINDOWS y a la sencillez de la interfaz. Así mismo, el ambiente de programación visual de C++BUILDER facilitó su programación.

Quisiera comentar que en un principio se había hecho el programa para

DOS. Sin embargo, decidimos presentarlo bajo WINDOWS por la transparencia del manejo de la memoria. Estuvimos tentados a llevarlo a una plataforma UNIX, pero no nos pareció apropiado, dados los intereses en el proyecto. Gracias a la modularidad alcanzada a través de la programación orientada a objetos, los cambios no se dieron en los algoritmos, sino que se incorporaron funciones a los objetos que aprovechaban las capacidades de Windows.

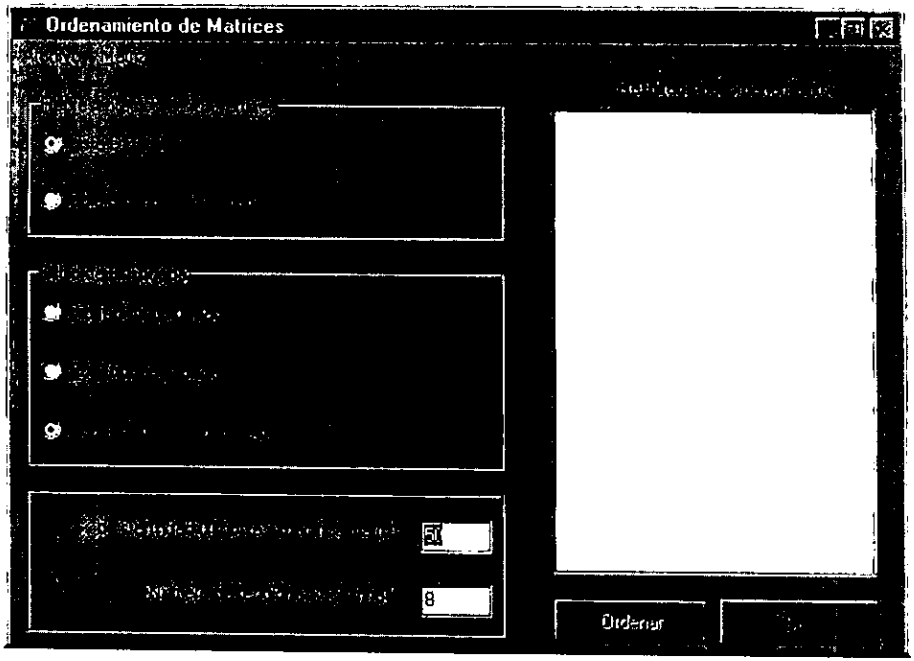



Ilustración 4: Pantalla principal. Muestra que el usuario solicita el ordenamiento con criterio de indistinguibilidad y siguiendo el modelo de la gráfica de eliminación, con una representación en listas dinámicas. Además parece que creará una matriz al azar ya que indica que quiere que tenga una densidad aproximada al 50% (ie. tantos **0**'s como **1**'s, previo a que se conviertan en **1** los elementos de la diagonal principal), en una matriz de tamaño **8x8**.

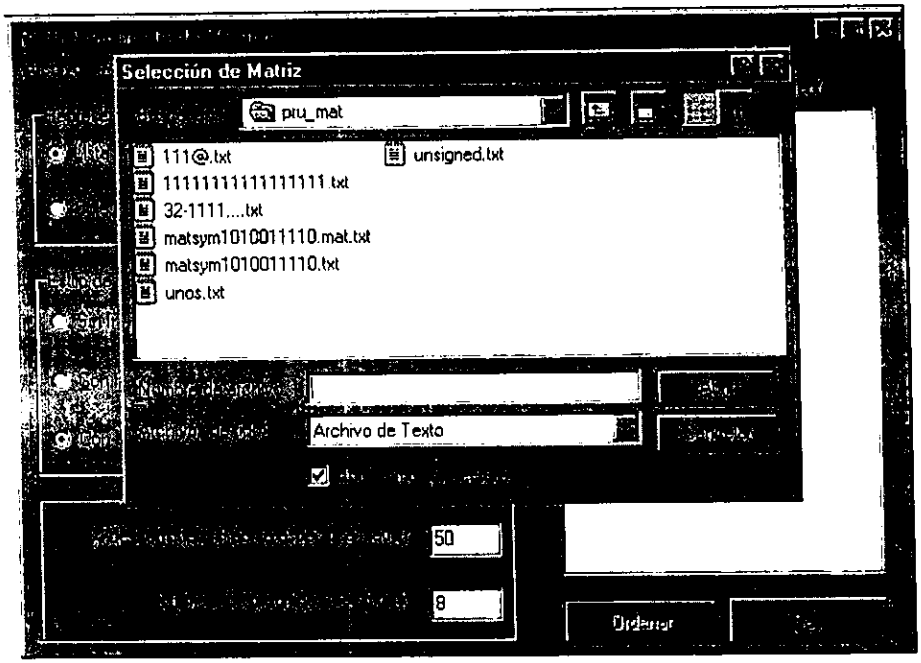



Ilustración 5: Pantalla de Selección de Matriz (para su ordenamiento). Indica el subdirectorio actual, los archivos que tienen extensión "txt", como se escogió en la opción "Archivos de tipo". Dicho archivo se abrirá "como de "sólo lectura". El primer botón en la esquina superior derecha que contiene una flecha, sirve para subir un nivel en el árbol de subdirectorios. El inmediato a la derecha se utiliza para crear una subdirectorio nuevo; los últimos dos indican la presentación de los archivos en la ventana interior, para su selección.

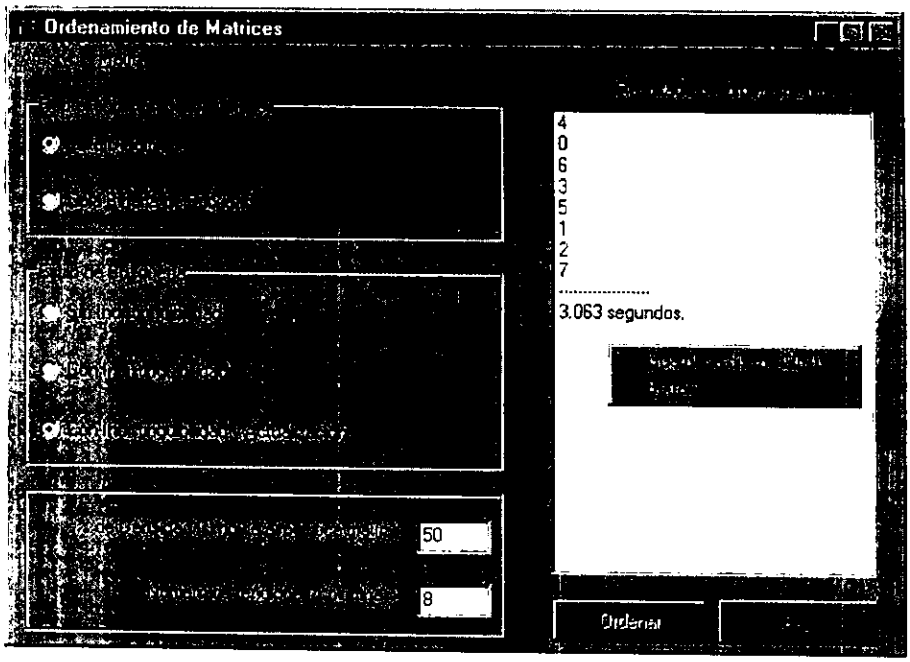



Ilustración 6: Pantalla principal con menú Pop-up. Este menú ofrece la posibilidad de vaciar a un archivo el contenido del *ListBox*- sobre el que se apretó el botón secundario del mouse para que saliera dicho menú, así como de borrar todo su contenido.

ESTRUCTURA Y CÓDIGO

El algoritmo que se llama con la opción "Sin indistinguibilidad" implica confiar totalmente en el ordenamiento construido al inicializar el objeto que contenga a los vértices. Se describe como sigue:

- Ordenar los vértices dentro de la estructura escogida
- Inicializar el arreglo de almacenamiento de los vértices seleccionados: apuntando a la primera casilla.
- Mientras no esté vacía la estructura escogida sacar de ella y guardar, en el ordenamiento, el vértice mínimo.

El algoritmo que se llama con la opción "Con indistinguibilidad" es muy parecido al "Sin indistinguibilidad", salvo que se aplica una búsqueda de indistinguibles sobre los vértices restantes.

- Inicializar el orden y la estructura escogida
- Inicializar la función hash para cada vértice calculando su valor
- (1) Mientras queden vértices en la estructura:
- Sacar el vértice minimo, y guardarlo en *minimo*
- Guardar *minimo* en el ordenamiento

- (2) Mientras el grado del mínimo actual coincida con el grado de *minimo*:

- Separar los vértices: si es indistinguible, guardarlo en el ordenamiento, si no, almacenarlo.

Ya sabemos que no hay más inidistinguibles al mínimo incial (de esta iteración).

- Revisar indistinguibilidad entre los vértices almacenados.

Ya sabemos que tienen el mismo grado.

- A todos los vértices guardados en el ordenamiento se les hace su función hash igual a **0**.

Algo que conviene hacer notar es que el almacenamiento de los vértices, en la lista de espera para ser revisados por su indistinguibilidad, se hace de manera distinta según la estructura escogida. En el caso de los listados dinámicos, se utilizó una lista, llamada *hashlist*, que los guardaba; mientras que en el caso de la cola binaria, el espacio que iba quedando vacío al final del arreglo que almacenaba a la cola se iba ocupando con estos vértices. No ocurren problemas de utilización de espacio en el arreglo, ya que de menos queda un espacio vacío al quitar el vértice mínimo. Al quitar el siguiente ya quedan dos, se guarda el vértice en el mínimo y entonces queda una casilla vacía entre la colección de posibles indistinguibles y los todavía no seleccionados. La ventaja de re-utilizar el espacio es obvia. Sin embargo, al ir quitando vértices indistinguibles en esta segunda revisión, como no se mantienen contiguos los vértices no eliminados( para evitar operaciones de copiado), se hacen búsquedas en espacios innecesarios. Posiblemente una mejora (aunque habría que instrumentarla para compararla) sería la de tener una lista dinámica temporal.

El algoritmo que se llama con la opción "Con indistinguibilidad y Actualización" es muy

parecido al de "Con indistinguibilidad", salvo porque se modifica la matriz cada vez que se guarda un vértice en el ordenamiento:

- Inicializar el orden y la estructura escogida
- Inicializar la función hash para cada vértice calculando su valor
- (1) Mientras queden vértices en la estructura:
- sacar el vértice minimo, y guardarlo en *minimo*
- guardar *minimo* en el ordenamiento
- (2) Mientras el grado del mínimo actual coincida con el grado de *minimo*:
- Separar los vértices: si es indistinguible, guardarlo en el ordenamiento, si no, almacenarlo.
- Actualizar la matriz, ie. si **x,y** son adyacentes a *minimo*, entonces *mi_matriz(x,y)* **:= 1**

Ya sabemos que no hay más inidistinguibles al mínimo inicial (de esta iteración).

- Revisar indistinguibilidad entre los vértices almacenados. Igual que en el paso anterior, por cada grupo de indistinguibles se actualiza *mi_matriz* antes de pasar al siguiente grupo.

Ya sabemos que tienen el mismo grado.

- A todos los vértices guardados en el ordenamiento hacerles su función hash igual a **0**.

Desgraciadamente la idea de llevar la gráfica de eliminación, dada mi instrumentación directa de la matriz (que fue promovida porque las matrices que reportan Gil, J. *et al* tienen muy frecuentemente densidad superior al 50% [GS]), toma operaciones de orden $\mathbf{n}^3$ ($\mathbf{n}^2$ por la doble revisión de adyacentes y $\mathbf{n}$ por los vértices que se eliminan). Aunque se explota el hecho de que la matriz es simétrica, y que además se lleva registro de los vértices ya eliminados, de todas formas todas estas operaciones son las que consumen más tiempo de proceso, como se verá en los resultados. Quiero notar, sin embargo, que la cota $\mathbf{n}^3$ es con frecuencia muy holgada, ya que se omiten las dobles iteraciones en los vértices ya eliminados; y aún cuando sí se efectúa la doble iteración, se omiten los vértices ya no adyacentes.

## APLICACIONES

Prácticamente las aplicaciones que tiene un algoritmo como éste son en cualquier situación donde haya una matriz grande. Nosotros las utilizamos en el proyecto IN-310296, Sistema Automático para el Análisis de Redes Sociales, *SAPARES*, patrocinado en parte por la Dirección General de Asuntos del Personal Académico de la UNAM. Los grafos que se emplean ahí son superposiciones de *cliques*. Por lo tanto el grado de vértice es uno más la suma sobre los *cliques* a los que pertenece, de **-1** más el número de vértices en el *clique*. La idea para aplicar este algoritmo en esta investigación es encontrar el grado del grafo, encontrar el vértice central, encontrar rutas mínimas, etc. para después aplicarlo al problema particular del grupo social que se esté estudiando.

Otra aplicación es linearizar los problemas de ecuaciones diferenciales. Aquí también aparecen grandes matrices y puede ser, dependiendo del grado de aproximación que se quiera obtener, que se tengan que resolver muchos.

En ingeniería lo utilizan mucho para problemas de resistencia de materiales. El escenario típico es que discretizan la superficie del material que van a estudiar y los consideran como unidades de materia que interaccionan con las de alrededor. Luego le aplican una fuerza externa y ven las interacciones. En este problema, como será en la mayoría de los casos, la matriz inicial no es de **0, 1**.

Ahora que está tomando tanta importancia el *DATA-WAREHOUSE*, las bases de datos se pueden beneficiar al utilizar una modificación muy discreta al código. Es decir, las aplicaciones de esto son infinitas y, como ocurre frecuentemente en matemáticas, basta solamente tener la representación adecuada de los datos.

Para mostrar la utilidad de este ordenamiento, sería conveniente observar el siguiente ejemplo, donde se consideran **577** vértices y una red dada por cuatro propiedades. La propiedad $\mathbf{P_1}$, la tienen **400** vértices (círculo); la $\mathbf{P_2}$, **80** (octágono); la $\mathbf{P_3}$, **60** (hexágono); finalmente la $\mathbf{P_4}$, **40** (rectángulo). Para construir la red total, se considera que todos los nodos que cumplen la misma propiedad están relacionados entre sí, formando un *clique*. De forma que cuando un vértice goza de varias propiedades estos *cliques* se unen a través de él.

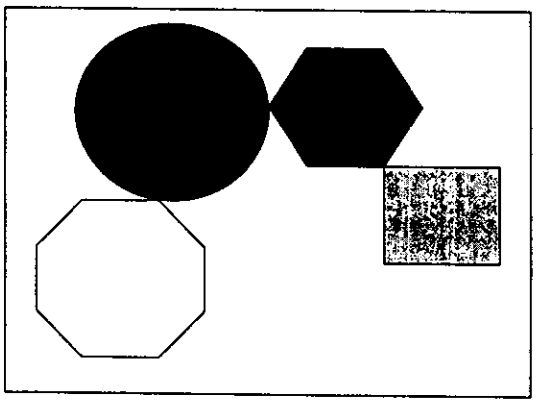


Ilustración 7: Esquema de una posible red. Se ven los *cliques* generados por cuatro propiedades.

Si observamos que el grado de cada vértice es igual a la suma sobre las propiedades que comparte de la cantidad de vértices que tiene cada propiedad (menos las repeticiones), nos será más fácil seguir el siguiente razonamiento. Poniendo ejemplos, un vértice que esté en el interior del cuadrado tendrá grado **40** ya que sólo pertenece a una propiedad y ésta tiene **40** vértices. Si en cambio nos fijamos en el mismo cuadrado, pero en el vértice que también comparte la propiedad hexágono, notaremos que tendrá grado **99**, **40** del cuadrado más **60** del hexágono, menos **1** de los vértices que estén en ambos (en este caso, solamente él mismo).

Si aplicamos el algoritmo de grado mínimo, nos daremos cuenta que el ordenamiento será:

- **39** vértices de grado **40** que pertenecen a la constelación *cuadrado* (jugando un poquito con las imágenes)
- **58** vértices de grado **60** que pertenecen a la constelación *hexágono*
- **78** vértices de grado **80** que pertenecen a la constelación *octágono*
- **398** vértices de grado **400** que pertenecen a la constelación *octágono*
- **1** vértice de grado **99** que pertenece tanto a la constelación *cuadrado* como a la *hexágono*
- **1** vértice de grado **459** que pertenece tanto a la constelación *rectángulo* como a la *hexágono*
- **1** vértice de grado **479** que pertenece tanto a la constelación *rectángulo* como a la *octágono*

Si consideramos los vértices que están "mal" ordenados notaremos que son sólo tres. Lo cual nos dio un ordenamiento casi perfecto; pero si nos fijamos aún más detalladamente, podemos observar que se logró dividir la gráfica en *cliques*, lo cual es uno más de los problemas *NP-completos*[PS, p.360].

Una última observación, que levanto sin formalidad tratando de inspirar al lector a pensar en este problema, es que este problema también se puede transformar en una versión aproximada a los problemas *0-1-Knapsack* e *Integer-Knapsack*. Estos problemas se pueden pensar como: *dada una lista finita de enteros y un entero a, ¿es posible asignarles coeficientes 0 ó 1 (o enteros, en* Integer-Knapsack*) a los elementos de la lista, para que la suma total sea el entero* a? [PS, p.374] En este caso los elementos de la lista serían las cardinalidades de los *cliques* y habría un término negativo atribuible a compensar la posible repetición de vértices que existan en la intersección entre *cliques*. Este planteamiento podría responder a la pregunta: ¿a qué cliques debería un vértice pertenecer, para tener exactamente a vértices adyacentes?

Es claro que este ejemplo es muy sencillo (por las pocas intersecciones que hay entre *cliques*), pero nos ha dado elementos para pensar que las aplicaciones del algoritmo de grado mínimo pueden ser mucho más importantes que ser solamente un preprocesamiento. Las implicaciones esbozadas en la discusión del párrafo anterior ameritan un estudio más detallado, ya que pueden ayudar a comprenderse mejor estos importantes y difíciles problemas y a encontrarles soluciones alternativas y, posiblemente, mejores a las conocidas.

ESPACIO DE PRUEBAS

Los experimentos se hicieron, en todos los casos, cinco veces y lo que se reportó es su promedio. En el caso con 8192 vértices, se hicieron algunas pruebas que no se reportan, ya que los resultados no eran significativos. Esto es debido a que para esa magnitud, el sistema operativo WINDOWS98 hacía acceso a disco por falta de memoria RAM, lo cual aumentaba el tiempo y desvirtuaba la posible comparación entre el desempeño de los distintos algoritmos presentados.

Las matrices de prueba fueron calculadas al azar en el momento de cada ejecución; por eso, se hicieron varias pruebas para diluir la variación en la estructura fundamental y densidad de la matriz.

El espacio de muestras se tomó:

- en la dimensión de la densidad (dentro de un rango del cero al cien), en **0, 20, 40, 60, 80, 100%** .
- en la dimensión del tamaño (dentro de un rango de dos a 8600) en **16, 64, 128, 256, 512, 1024, 2048, 4096**
- y cinco veces cada creación de la matriz al azar, promediando los cinco resultados.

La máquina de prueba contaba con un procesador Intel Pentium II/266. Su memoria RAM era de 32Mb; se permitió que WINDOWS controlara la memoria virtual. Durante el tiempo de ejecución las máquinas no fueron utilizadas para no "distraer" al sistema.

## TENDENCIAS

Como se puede ver en las gráficas tridimensionales, la variación no es importante a lo largo del eje **densidad**. Es cierto que existen unos picos muy marcados en algunos puntos, sin embargo la regularidad general nos lleva a pensar que se trató de situaciones particulares fuera de nuestro programa, posiblemente atribuibles a WINDOWS98, como por ejemplo algún cambio debido al monitoreo de ahorro del uso de energía por desuso del disco duro, o algún proceso que no se hubiera terminado antes de empezar las pruebas. Al notar esta muestra tan particular, se repitieron las mediciones en esos puntos, dando, esta vez, valores dentro del rango normal que se esperaba. Como se verá más adelante, los resultados finales fueron

bastante robustos aún frente a estas variaciones tan inesperadas.

El crecimiento en el eje **dimensión**, concuerda con las expectativas hechas *a priori*. Las gráficas que llevan por título los mecanismos de selección muestran cómo, a pesar de las variaciones extrañas que se comentaron anteriormente, la diferencia en comportamiento entre la representación en colas binarias y las listas dinámicas fue desapareciendo rápidamente conforme crecía la dimensión. Al grado de tener comportamientos casi indistinguibles en las matrices grandes.

Debido al enriquecimiento en los criterios de selección que comparamos los resultados pueden parecer lógicos. Sin embargo, según la literatura, estas variaciones que se le hicieron al ALGORITMO DE GRADO MÍNIMO tendían a hacer el proceso más rápido. Nosotros creemos que el hecho de que hayamos utilizado una matriz en lugar de un conjunto dinámico de aristas es lo que hizo que el ahorro que resulta al eliminar vértices y aristas de una gráfica se diluyera entre las operaciones de mantenimiento de la lista hash y control de la gráfica de eliminación.

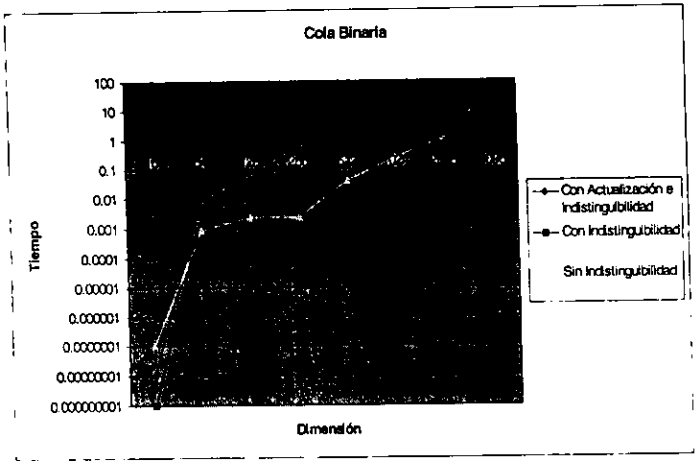
## CONCLUSIONES

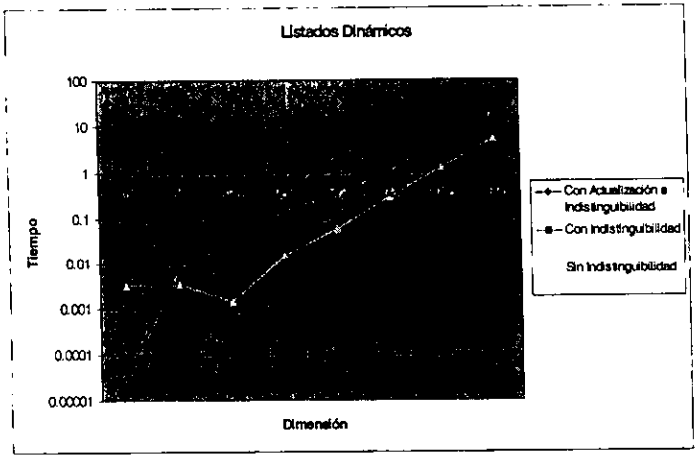
Tanto el Ing. Gil como yo quedamos muy complacidos con los resultados. Sentimos que los resultados apoyan el hecho de que las listas dinámicas y las colas binarias son soluciones equivalentes desde el punto de vista del tiempo que toman para el ordenamiento. Así alcanzamos el objetivo principal que nos propusimos al

inicio de este trabajo.

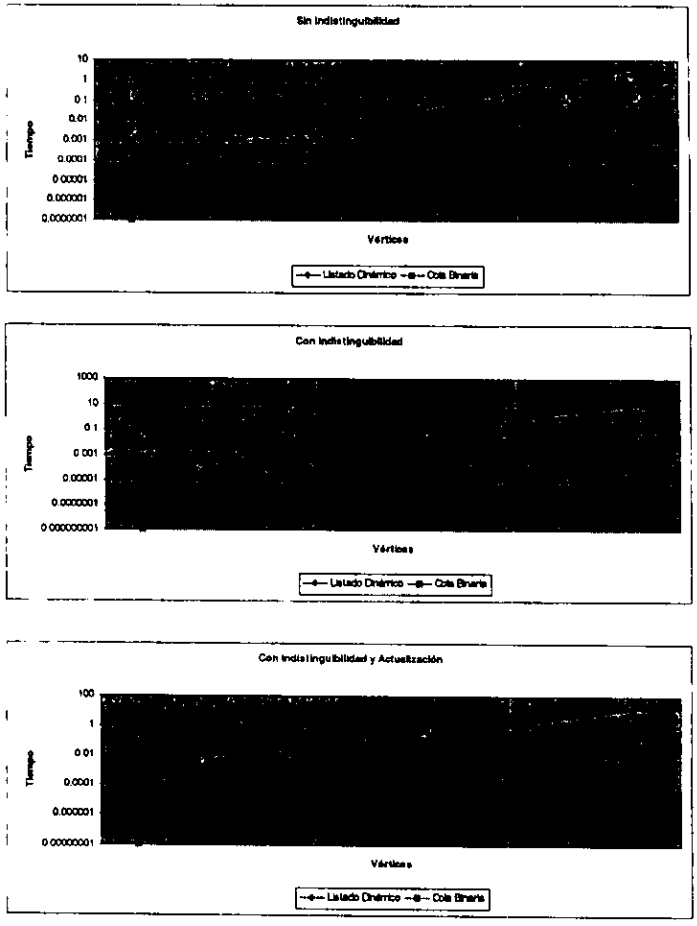
Encontramos, además, que la representación que se utilice de la gráfica es fundamental en el desempeño final del algoritmo. Creemos, por consiguiente, que posiblemente convenga probar otras representaciones antes de desarrollar el sistema final.

Por último, y de la mano del objetivo principal, queda demostrado que aunque en teoría la complejidad de un algoritmo es superior al de otro, las constantes, que usualmente se desprecian, pueden ser muy importantes.




Ilustraciones 8 y 9 Desempeño según almacenamiento de los vértices (Cola Binaria y Listados Dinámicos). Relación de Tiempo x Dimensión.




Ilustraciones 10, 11 y 12: Desempeño según estrategia de selección (Sin Indistinguibilidad, Con Indistinguibilidad, Con Indistinguibilidad y Actualización). Relación de Tiempo x Dimensión.

## APÉNDICE A

Clave del Proyecto: IN310296

Nombre del Proyecto: **SAPARES** SISTEMA AUTOMATICO PARA EL ANALISIS DE REDES SOCIALES.
Nota: Proyecto multidisciplinario: *computación, matemáticas, sociología, antropología, ciencia política.*

1. Comentario General

El Proyecto presentado es de naturaleza interdisciplinaria involucrando áreas del conocimiento que tradicionalmente carecen de puentes, entre las que destacan: matemáticas, computación, sociología y ciencia política. De hecho el concepto de redes sociales fue aplicado por primera vez por los antropólogos que buscaban explicaciones para relaciones sociales complejas en las comunidades que los marcos explicativos tradicionales, no alcanzaban a descifrar, como la explicación de las clases sociales.

El director de este proyecto es un pionero en la aplicación de la teoría de redes a fenómenos políticos, tanto en México como en el extranjero. Esto se ha comprobado por la presentación de diversas ponencias en congresos internacionales. Este es un proyecto de computación en cuanto a que hay que diseñar un sistema automático para el análisis sistemático de redes grandes. Este sistema lo visualizamos constituido por una base de datos orientada a objetos, codificación de un conjunto de algoritmos de teoría de gráficas, aplicables al análisis de redes sociales, sistemas de representación gráfica apropiados a los problemas bajo estudio. Analizará la conveniencia de integrar todas las partes bajo el concepto de un sistema experto. Todo esto enmarcado en un sistema orientado al usuario, o sea, un sistema de uso "amistoso". Para el proyecto es necesaria la identificación de teoremas de teoría de gráficas aplicables al análisis de redes y construcción de los correspondientes programas para computadora utilizando programación orientada a objetos.

El sistema debe ser universal, porque su desarrollo tiene aplicaciones múltiples para el análisis de diversos tipos de redes siendo, consecuentemente, aplicable a diversas disciplinas como sociología, ciencia política, antropología, administración de empresas, mercadotecnia, administración pública, etc. En pocas palabras, este sistema es una herramienta útil para estudiosos de las relaciones entre individuos y sistemas en general. Los programas de ayuda para el análisis de redes que existen actualmente son muy limitados y prácticamente incapaces de analizar redes grandes. Estudios estándar de redes cubren un promedio de 20 nodos, nuestra intención es tener un sistema capaz de analizar redes de más de 1000 nodos.

En México si bien hay una literatura muy amplia sobre estudios de distintos grupos sociales, existen muy pocos expertos en el análisis de redes sociales utilizando este tipo de herramientas, entre los más destacados se encuentran: Dra. Larissa Lomnitz, Dr. Raúl Carvajal, Ing. Jorge Gil Mendieta y sus colaboradores, Jorge Castro y Alejandro Ruiz. En el extranjero tres académicos se ocupan del análisis de la elite política mexicana. Peter Smith, el que después de publicar su libro El Laberinto de Poder prácticamente ha abandonado este tipo de estudios, aunque ha reconocido en un par de entrevistas con el director de este proyecto, que mi trabajo es algo que él decidió no intentar. Roderic Camp, tal vez el mas importante biógrafo de la elite política mexicana no sigue esta metodología. El tercer académico es Samuel Schmidt (Universidad de Texas en El Paso) quien ha colaborado con Camp durante los últimos cuatro años, realizando entre ambos varias publicaciones y ponencias en congresos. Entre ambos han estudiado la Red de Poder de México, y nos hemos enfrentado al problema de analizar una red que tiene más de 3000 nodos. Hasta la fecha J. Gil, S. Schmidt, Jorge Castro y Alejandro Ruiz han investigado sobre partes de esta red que se ha traducido en varios artículos de investigación original y múltiples conferencias. El financiamiento de la DGAPA servirá para ampliar este proyecto y consolidar la creación del Laboratorio de Redes del IIMAS, el cual estará orientado a apoyar a estudiantes de posgrado que hagan investigación sobre redes sociales, algoritmos y nuevas aplicaciones, desarrollo de nuevos algoritmos y sistemas que sirvan a investigadores de diversas disciplinas.

2. Comentarios de innovación tecnológica

Actualmente no existe en el mercado un sistema experto de análisis de redes sociales, ni un sistema de análisis de redes sociales para grandes redes (del orden de 1000 ó más nodos). El Laboratorio de Redes utiliza una base de datos comercial, que se adecua al manejo de bases de datos que incluyen grandes cantidades de

información de tipo subjetivo o se diseñará un sistema apropiado a las necesidades del usuario. De hecho se construirá un sistema inteligente que facilite el trazo de redes sociales complejas cuyos actores se mueven con frecuencia.

3. Componente de Ciencias Sociales

En las ciencias sociales hay una gran resistencia a los análisis matemáticos, en parte porque muchos estudios han utilizado a la estadística como un medio de validación de hipótesis y no como un instrumento para la formulación de hipótesis. Adicionalmente, las ciencias sociales han tenido dificultad para incorporar información subjetiva, lo que ha limitado el desarrollo de formulaciones no convencionales tal y como lo plantea el politólogo israelí Yehezkel Dror. En este proyecto se trabajará en una herramienta matemática orientada a formular hipótesis respecto a un fenómeno social y a su comprobación. El hecho que el estudio de redes sociales sea de por si de carácter interdisciplinario dentro de las mismas ciencias sociales hace que este proyecto sea atractivo para académicos de varias disciplinas. Adicionalmente, el proyecto lograra atraer a matemáticos, estadísticos y computólogos al estudio de fenómenos sociales y de sistemas en general. Se pretende hacer más atractivo el uso de herramientas matemáticas por los estudiosos de las ciencias sociales.

4. Consideraciones finales

Estamos en una época en la cual las disciplinas tradicionales se encuentran con múltiples barreras para analizar fenómenos complejos. El desarrollo científico reclama cada vez más la creación de puentes interdisciplinarios y la comunicación interdisciplinaria. La humanidad se encuentra ahora ante problemas inéditos y las herramientas tradicionales son insuficientes para explicarlos. Uno de estos problemas es la estabilidad política mexicana, la que hasta ahora ha sido insuficientemente explicada por medio de los métodos tradicionales, sin embargo, hemos encontrado que el análisis de redes proporciona una nueva dimensión para explicar esta estabilidad eventualmente para re-escribir la historia política del México contemporáneo.

Este proyecto incide en desarrollo tecnológico, desarrollo científico y crea puentes interdisciplinarios cuya importancia es difícil de valorar. La complejidad de este proyecto obliga a que su lectura y evaluación se haga desde una perspectiva interdisciplinaria. Visto desde una sola disciplina se correría el peligro que los principios paradigmáticos de esa sola disciplina obstruyan la riqueza del mismo. No es un proyecto de computación pura ni de ciencia política pura, aunque su desarrollo ayudará a responder preguntas en ambos campos y en otros, como por ejemplo para los antropólogos, ingenieros y matemáticos.

El proyecto también tiene un componente de desarrollo de tecnología, porque puede llevar a la elaboración de aplicaciones de computación en el área de análisis estructural de sistemas automáticos de análisis de estructuras complejas. Esta línea de trabajo se ha visto impulsada por la inauguración, el pasado mes de septiembre, del "Laboratorio de Redes" del IIMAS cuya misión está orientada al estudio de las aplicaciones de la teoría de gráficas.

OBJETIVOS:

Diseñar y construir un Sistema Automatizado de Análisis de Sistemas y Organizaciones Complejas, mediante teoría de gráficas.

El objetivo principal es construir un "Sistema Experto" para analizar información de bases de datos sobre cualquier tema en particular y construir la red de relaciones de los actores involucrados dentro de una base, así como sus subredes, conexiones, distancias, influencia, poder, centralidad, cliques, clanes, facciones, bloques, etc.

La red que planeamos analizar, además de la Red Política de México, es la Red de Científicos que trabajan en medio ambiente en el ámbito nacional e internacional.

La codificación de los diversos algoritmos de teoría de gráficas que sean aplicables al análisis de redes y su prueba experimental utilizando estas bases es otro de los objetivos de este trabajo.

Algunas de las preguntas que queremos contestarnos son, entre otras:
¿Cuál es el nodo central de una red?
¿ Cómo podemos determinar las diversas subredes que actúan dentro de una organización, p. ej. la política?
¿ Se puede determinar algún índice que nos describa el "poder" de un nodo? ¿De un clique? ¿De una subred?

¿Cómo podemos separar de una red las subredes que constituyen el total?

Difundir los resultados en eventos nacionales e internacionales, así como a través de revistas especializadas.

# APÉNDICE B

## GLOSARIO

ADYACENCIA: Se dice que el vértice **v** es adyacente al vértice **u** en una gráfica **G**, si y sólo si **(u,v)** pertenece a **A**.Si el contexto lo permite sin ambigüedad, se omite el nombre de la gráfica, y sólo se dice "**v** es adyacente a **u**".

ANCHO DE BANDA: La distancia máxima sobre los renglones que existe entre el primero y último elemento no-cero de cada renglón. [NOTA: En la literatura consultada no se encontró la definición de este término; así que, por completud, sugiero esta definición.]

ÁRBOL: Gráfica conexa sin ciclos.

ÁRBOL BINARIO: Un árbol cuyos vértices tienen grado **3**, excepto los vértices terminales que son de grado **1** y el nodo raíz que tiene grado **2**.

ÁRBOL BINARIO COMPLETO: Es un árbol binario, de forma que todos los nodos terminales estén a la misma distancia del nodo raíz (es decir, que estén al mismo nivel).

CLIQUE: Se dice que "**H** es un clique de **G**" cuando **H** es una subgráfica completa de **G** y maximal. También se conoce como "clan".

FILL-IN: (*rellenar*) se refiere a las aristas que se crean al eliminar un vértice en la gráfica de eliminación. Se puede considerar como *transitividad*, en el sentido en que si **a** y **b** son adyacentes a **x** en la gráfica original, entonces **a** es adyacente a **b** en la gráfica resultante al eliminar **x**.

GRÁFICA: Una gráfica está constituida por dos conjuntos: el conjunto de *vértices* **V** y un subconjunto **A**, de *aristas*, de **VxV**. Usualmente **A** representa una propiedad binaria sobre parejas de vértices. Cuando **A = VxV**, se dice que es una "gráfica completa".

GRAFO: Sinónimo de *gráfica*. Éste es el término sugerido por el Dr. Frank Harary, con el fin de mantener el contraste existente entre "graph" y "graphic".

LISTADO DINAMICO: Estructura que representa una gráfica en forma de cadena (cíclica o no) que permite la anexión o separación de vértices o eslabones mediante la simulación, a través de apuntadores, de las uniones entre dichos eslabones.

LISTBOX: Es el nombre de un componente visual de C++ Builder. En el programa es donde aparecen el ordenamiento y la matriz.

MATRIZ DE ADYACENCIA: Es una matriz creada en el espacio **VxV**, donde para cada pareja se le asocia un número: **0** si no son adyacentes y **1** si son adyacentes. En ciertos casos, cuando la propiedad que define **A** no es binaria, las parejas pueden tomar distintos valores. Si este es el caso, se dice que "las aristas tienen pesos".

MENU POP-UP: Es el tipo de menú de opciones que aparece en WINDOWS cuando se oprime el botón secundario del mouse.

Como se ha dicho antes, el programa fue escrito bajo el paradigma de la *Programación Orientada a Objetos*. Casi todos los objetos se escribieron en una unidad aislada (las unidades se indican con *itálicas* y extensión ".h"), excepto los vértices que están en ***vertfin.h*** y las listas que están en *listfin.h* . El orden, en el que aparecen en la tabla siguiente, es por inclusión para el compilador. La inclusión es obviamente a través del comando "#include". La unidad *tesis* es la unidad de la aplicación, *tesis1.h* de la forma(ventana), *heapchek.h* contiene los algoritmos para la cola binaria, *listchek.h* es análogo a *heapchek.h* pero para listas, *heap.h* amplía los métodos y propiedades de *colavert.h* que contiene la instrumentación genérica de una cola binaria de prioridad, *listfin.h* contiene la instrumentación genérica de una lista dinámica, así como la clase que contiene al arreglo de listas. Directamente se utiliza esta última clase en el programa, dado que las listas son protegidas. *Ordenfin.h* encapsula el comportamiento del ordenamiento. Se encarga de almacenar el orden de selección de los vértices y de mostrarlo al final. *Hashfin.h* se encarga de controlar lo referente a la función hash **h**( ). *Vertfin.h* tiene los vértices que se usan en ambas representaciones; *matrix.h* se encarga de todo lo referente a la matriz. Finalmente *dfmxsize.h* sólo contiene la definición de la constante ***def_max_size*** que representa el valor máximo permitido para la dimensión de la matriz.

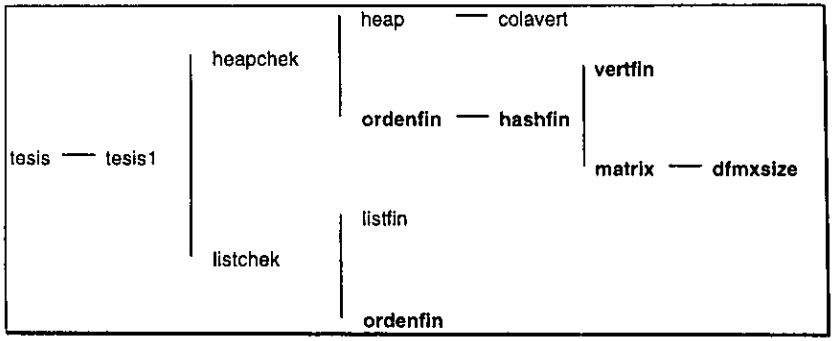



Ilustración 13: Estructura del código. Unidades programadas por mí que componen la aplicación. En **negrita** están aquellas que se comparten tanto en listas como en colas binarias.

## DESCRIPCIÓN DEL CÓDIGO

El código está dividido en varias librerías. Empezaré describiendo las que se utilizan en los dos paradigmas principales de la instrumentación. Si se define un símbolo como "vertices_hpp" basta cambiar "_" por "." para saber el nombre del archivo.

Todos los elementos principales que utilizamos están instrumentados como objetos. Cada objeto lo implementamos en su propio archivo. Enlisto aquí abajo las declaraciones de los objetos.

Declaramos al objeto matriz como se ve en el listado 1. Tiene dos variables que guardan el tamaño de los renglones y de las columnas, así como un apuntador a la primera casilla de la matriz. Las entradas son del tipo ***char***. Al operador ( ) lo definimos ( incluí aquí su código ) para que reciba dos entradas que se toman como los índices de la matriz y entonces regresa esa posición. La función ***pon*** asigna el valor ***w*** a las entradas ***i,j***; mientras que la función ***asign_sym*** lo hace simétricamente.

Listado 1: MATRXCLS.HPP

```
class matrix
{
protected:
        unsigned c_size, r_size;
        char **p;
public:
```

```
        matrix(unsigned c, unsigned r);
        ~matrix();
        char& operator() (unsigned i, unsigned j) const { return p[i][j];}
        void pon(unsigned i, unsigned j, char w)
        void asign_sym( unsigned i, unsigned j, char w)
};
```

En el listado 2 se declaran tres objetos. Cada uno puede ser una variación del anterior, pero ***minivert*** y ***vertice*** son diferentes pero ***vertice_heap*** sí se hereda de ***vertice***. Ya que son prácticamente las mismas variables que se utilizan en cada objeto, y como también se le asignan valores *default* a vértice, es que instrumenté *typecasting* entre ellos. No se pierde más información que la que se perdería en una instanciación sin parámetros.

Por lo sencilla que resultó ***minivert*** es que decidí no hacer ***etiqueta*** privada. Sin embargo en las otras dos clases sí, pudiendo acceder y asignar através de las funciones ***ver_grado(), ver_etiqueta(), asigna()***. Tratamos de huir de las funciones amigas, pero se creyó conveniente definir a ***separa_vert( const vertice_heap& , const vertice_heap& )*** amiga de ***vertice_heap***.

***Vertice_heap*** se empleó, como su nombre indica, para la sección de la cola binaria; ***minivert*** se utilizó para la sección del listado dinámico, ya que no se necesitaba guardar su grado porque éste se podía determinar al saber en cuál de los niveles estaba ( ver más adelante la descripción de ***list*** ).

Listado 2: VERTICES.HPP

```
class vertice;
class vertice_heap;

class minivert
{
        public:  unsigned etiqueta;
        minivert ( unsigned u = 0 )
        minivert( const vertice &v)
        minivert operator = (const vertice&);
        int operator == ( const minivert& m)
};


class vertice
{
 protected:
        unsigned grado, etiqueta;
 public:
        vertice( const vertice_heap &v)
        vertice operator = (const vertice_heap&);
        vertice( const minivert& mini )
        vertice operator = (const minivert& m )
```

```
        vertice() {grado = etiqueta = 0;}
        void asigna( unsigned a = 0, unsigned b = 0)
        unsigned ver_grado() ;
        unsigned ver_etiqueta();
};

class vertice_heap: public vertice
{
protected:
        friend void      separa_vert( const vertice_heap& , const vertice_heap& );
public:
        vertice_heap(){}
        vertice_heap(vertice_heap & value)
        const vertice_heap&  operator=(const vertice_heap& );
        vertice_heap   operator=(const vertice &);
};
```

***Orden*** es el objeto encargado de recibir el ordenamiento de los vértices. Consta de un arreglo y de un marcador. Solamente se puede inicializar, imprimir todo, y guardar un nuevo vértice en la próxima posición libre ( ver listado 3 ).

Listado 3: ORDEN.HPP

```
class arreglo_orden
{
private:
        unsigned        arreglo[def_max_size];
        friend void separa_vert( const  vertice_heap& , const  vertice_heap& );
        unsigned marca;
public:
        arreglo_orden();
        void guarda ( unsigned cual, hash_indist& hashy )
        void imprime();
        void init()
};
```

El Listado 4 exhibe la declaración de la clase ***hash_indist***. Ésta se encarga de mantener el registro de los vértices que ya han sido eliminados, distinguiendo a éstos en los que se eliminaron en esta iteración y los que ya se eliminaron antes, así como del valor de la función de comparación de indistinguibilidad entre los vértices. Para la primera tarea, se hace uso de los valores negativos que permite el tipo ***long***. De esta forma usamos la función ***negativo()***. ***Convierte0( unsigned = def_max_size - 1)*** tiene la tarea de transformar todo valor negativo en **cero**. Ésta es la marca de que un vértice "ya no existe", a diferencia de un vértice que ya fue eliminado, pero que todavía puede tener vértices indistinguibles aún no eliminados, y por lo tanto conviene considerarlo todavía. ***Test_indist (const vertice&, const vertice& )*** entrega el valor de la comparación de los valores de hash. ***Actualiza_hash( unsigned, unsigned = def_max_size )*** se encarga de revisar en ese momento la matriz generando el valor hash para el vértice dado como

parámetro. En esta función se consideran solamente a los vértices que no han sido eliminados totalmente, es decir a aquellos que tienen **h() <> 0**. ***Suma_hash(unsigned cual, long cuanto )*** se encarga de actualizar el hash, pero sin revisar todo. Esta función se utiliza cuando se le añade una arista a un vértice.
***Ver_hash ( unsigned )*** hace lo que su nombre dice. ***Init()*** convierte todos los valores del hash a **cero**. A diferencia de ***init_total()*** que calcula el hash para todos los vértices.

Listado 4: HASH.HPP

```
class hash_indist
{
private:
        long    hash[def_max_size];
        friend void separa_vert( const vertice_heap& ,const vertice_heap& );
public:
        hash_indist(){}
        int test_indist (const vertice& , const vertice& );
        void negativo( unsigned cual )
        void actualiza_hash( unsigned, unsigned  = def_max_size );
        void convierte0( unsigned  = def_max_size - 1);
        void suma_hash ( unsigned cual, long cuanto )
        long ver_hash ( unsigned cual )
        void init();
        void init_total();
};
```

La instrumentación de la cola binaria de prioridad se muestra en el Listado 5. El código está tomado de [W]. Los pequeños cambios son que ***def_max_size*** la declaré global y además es el valor que se usa todo el tiempo, y no representa el tope de recursos del sistma. En el cuerpo del programa principal se hereda esta clase a una nueva que básicamente hace funciones propias del proyecto y por eso decidimos no incluirlas en esta clase base.

Listado 5: COLAGRAL.HPP

```
static const unsigned           initial_size = 1, min_val = 0;

template <class element_type>
class binary_heap
{
protected:
        unsigned max_size;
        unsigned size;
        element_type *elements;
public:
        binary_heap(const unsigned initial_size = def_max_size);
```

```
        binary_heap( binary_heap &value )
        ~binary_heap()
        const binary_heap &operator =(const binary_heap &value);
        void make_empty()
        int is_empty()
        int is_full()
        void insert(const element_type &x);
        element_type delete_min();
        element_type ver_element(unsigned q)
        element_type find_min()
};
```

La lista expuesta en el listado 6 es tomada también de [W]. La única adición que le hice fue la función ***am_last()*** que permite, al ir recorriendo la lista, saber si existe un elemento próximo.

Listado 6: LISTGRAL.HPP

```
template <class etype>
class list
{
protected:
        struct node
        {
        etype element;
        node *next;
        node ( etype e = 0,  node * n = NULL): element(e), next(n) { }
        };
        node *listhead, *current_pos;
        void delete_list();
public:
        list():listhead( new node ), current_pos( listhead ) { }
        list( list & value )
        virtual ~list() { delete_list(); }
        const list & operator = ( list & value );
        const list & operator ++ ();
        int operator !() const;
        etype operator ()() const;
        int is_empty()
        int  am_last()
        virtual int find ( const etype & x );
        virtual int find_previous ( const etype & x );
        virtual void insert( const etype &x );
        virtual void insert_as_first_element( const etype &x );
        void first
        void header()
        int remove( const etype & x );
};
```

La instrumentación final en nuestro caso se hace ya en el código principal, y sigue el consejo de Adolfo Cortés. Consiste en mantener un arreglo donde el índice indica el grado de los vértices que le siguen. Para dar una imágen, es como si fuera un peine. La columna es el arreglo *list* -as. Cada posición del arreglo es un *list*, sin ninguna modificación. Las inserciones a las listas se hacen a través de las funciones públicas de esta clase grande, que a su vez sólo usa las funciones públicas de *list*.

## CONSULTA

### *Artículos*

[ADD] AMESTOY Patrick, Timothy Davis, Iain Duff, **An Approximate Minimum Degree Ordering Algorithm,** SIAM J. Disc. Math. and Matr.Anal.(1996), pp. 886-905

[A] ASHCRAFT Cleve, **Compressed Graphs and the Minimum Degree Algorithm,** SIAM J. Sci. Comput. 16 (1995) pp. 1404-1411.

[DR] DUFF Iain, J.K. Reid, **A Comparison of Sparsity Orderings for Obtaining a Pivotal Sequence in Gaussian Elimination,** J. Inst. Math. Appl., 14 (1974) pp. 281-291.

[GS] GIL Jorge, S.Schmidt, **The Political Network in Mexico,** Social Networks, 18 pp.355-381.

[L] LIU Joseph W. H., **Modification of the Minimum-Degree Algorithm by Multiple Elimination,** ACM Trans. Math. Software, 11 (1985) pp. 141-153.

[Y] YANNAKAKIS Mihalis, **Computing the Minimum Fill-in is NP-Complete,** SIAM J. Alg. Disc. Meth., 2 (1981) pp. 77-79

*Libros*

[PS] Papadimitriou Christos H., K. Steiglitz, (1982). *Combinatorial Optimization: Algorithms and Complexity*, Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

[W] WEISS Mark Allen, (1993). *Data Structures and Algorithm Analysis in C++*, Redwood City, California: The Benjamin/Cummings Publishing Company, Inc.

## REFERENCIA

*Artículos*

GEORGE Alan, J.W.H. Liu, **A Minimal Storage instrumentation of the Minimum Degree Algorithm,** SIAM J. Numer. Anal., 17 (1980) pp. 282-299.

GIL Jorge, S.Schmidt, J. Castro, **Red de un ex-presidente mexicano,** Revista Mexicana de Sociología, (1993).

SCHMIDT Samuel, Jorge Gil, J. Castro, Alejandro Ruiz, **Un análisis dinámico de las Redes de Poder: El caso de México,** Metapolítica, Vol. 2, Num.7 (1998) pp.409-432.

*Libros*

DIESTEL, Reinhard, (1990) *Graph Decomposition:. A Study in infinite Graph Theory*, Oxford, Clarendon Press.

NAGLER Eric, (1993). *Learning C++: A Hands-On Approach,* St. Paul, MN: West Publishing Co.

POHL Ira, (1993). *Object Oriented Programming Using C++,* Redwood City, California: The Benjamin/Cummings Publishing Company Inc.

SHAMMAS Namir Clement, (1993). *Moving from Turbo Pascal to Turbo C++: The Ins and Outs of Object-Oriented Programming,* Carmel, IN: SAMS Publishing Co.

STROUSTRUP Bjarne, (1986). *The C++ Programming Language,* Murray Hill, NJ: Addison-Wesley Publishing Co.

## SOFTWARE

*Borland C++ v.4.5* (1994). [Programa de computadora]. Scotts Valley, CA: Borland International Inc.

*Borland C++ Builder Professional* (1997). [Programa de computadora]. Scotts Valley, CA: Borland International Inc.

*UCINET IV Version 1.04,* Borgatti, Everett, Freeman. (1992). [Programa de computadora]. Columbia: Analytic Technologies.