



**UNIVERSIDAD DE SANTIAGO DE CHILE  
FACULTAD DE INGENIERÍA  
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA**

## **Estructura de Datos y Análisis de Algoritmos**

### **Laboratorio N° 3 Redes sociales**

Autor: Leandro Pizarro J.

Profesor: Pablo Schwarzenberg R.

Ayudantes: Sebastián Vallejos A.

Javiera Torres M.

Nicole Henríquez S.

Santiago - Chile  
2-2017



## TABLA DE CONTENIDOS

Tabla de Contenidos	3
Índice de Figuras	5
Índice de Tablas	5
CAPÍTULO 1. Introducción	6
CAPÍTULO 2. Descripción de la solución	7
2.1 Marco teórico	7
2.1.1 Puntero	7
2.1.2 Memoria Dinámica	7
2.1.3 Listas Enlazadas	7
2.1.4 Grafo	7
2.1.4.1 Clique	8
2.2 Herramientas y técnicas	9
2.2.1 Recursividad	9
2.2.2 Búsqueda en profundidad	9
2.2.3 Librerías	9
2.3 Algoritmos y estructuras de datos	10
2.3.1 Estructuras utilizadas	10
2.3.1.1 MatrizAdy	10
2.3.1.2 Lista	10
2.3.1.3 Adyacentes	10
2.3.2 Algoritmos: Funciones más importantes	12
2.3.2.1 obtenerQliquesL y obtenerQliquesM	12
2.3.2.2 obtenerVinculosL y obtenerVinculosM	13
2.3.2.3 obtenerGruposL y obtenerGruposM	14
2.3.2.4 conectarL y conectarM	16
CAPÍTULO 3. Análisis de los resultados	17
CAPÍTULO 4. Conclusiones	18
CAPÍTULO 5. referencias	19
CAPÍTULO 6. Manual de usuario	20
6.1 Introducción	20
6.2 Compilación	20

6.2.1	Linux	20
6.2.2	Windows	21
6.3	Funcionalidades	21
6.4	Posibles errores	22

## ÍNDICE DE FIGURAS

Ilustración 2.1 Ejemplos cliques	8
Ilustración 2.2 Representación gráfica de estructura Lista	11
Ilustración 6.1 Ejemplo de compilación del programa	21
Ilustración 6.2 Programa en ciclo infinito	22
Ilustración 6.3 Ejemplo error archivo no encontrado	23

## ÍNDICE DE TABLAS

Tabla 1 Tiempos de ejecución y órdenes de complejidad	17
---	----

## CAPÍTULO 1. INTRODUCCIÓN

Un algoritmo puede entenderse como “una secuencia finita de instrucciones, cada una de las cuales tiene un significado preciso y puede ejecutarse con una cantidad finita de esfuerzo en un tiempo infinito” (Aho, Hopcroft, Ullman, 1995, p.2). Desde el punto de vista de la computación, un algoritmo pasa a ser una secuencia de operaciones lógicas necesarias para resolver cierta tarea o problema en específico. En esta ocasión y siendo la razón del siguiente escrito, dicho problema es la creación de un programa que permita la representación de una red de personas por medio de un grafo y determinar sus agentes de vínculo junto con los grupos de mejores amigos.

Para desarrollar este programa se utilizará el lenguaje de programación C cuya principal característica es su velocidad de procesamiento y el control que puede llegar a otorgar al programador sobre la memoria que utiliza su programa. Además, se trabajará bajo el estándar ANSI C lo que permitirá que el programa sea compatible a casi todos los compiladores. Tal como señala Santos (1995), el lenguaje C es simple, ya que no tiene tipo de dato booleano, manejo de cadenas ni memoria. Aún así, el estándar de C define un conjunto de bibliotecas que logran satisfacer estas necesidades.

El objetivo principal del desarrollo de este programa es lograr un análisis a tal nivel que facilite la abstracción del problema para llegar así al desarrollo efectivo de los algoritmos que permiten representar la red social en cuestión e implementar las funcionalidades mencionadas anteriormente. Además, se espera que el programa pueda ser compilado y ejecutado desde cualquier distribución de Linux o Windows, utilizando el conjunto de compiladores GCC. Por otra parte, y no menos importante, lograr calcular la complejidad y el orden del programa, aplicando así los conceptos que entrega el curso de Análisis de algoritmos y estructura de datos.

El presente informe consta de una descripción de la solución con el fin exponer el material investigado, conceptos que fueron la base en la que se irguió la solución al problema, tales como memoria dinámica, listas enlazadas, entre otros. Se dan a conocer las herramientas y técnicas utilizadas en el desarrollo de los algoritmos junto con la descripción de las funciones más relevantes dentro del programa. Finalmente un análisis de los resultados obtenidos y conclusiones. Además, adjunto a este informe se encuentra el manual de usuario respectivo.

## **CAPÍTULO 2. DESCRIPCIÓN DE LA SOLUCIÓN**

### **2.1 MARCO TEÓRICO**

#### **2.1.1 Puntero**

“Un puntero es una variable que contiene una dirección de memoria de otra. Los punteros se utilizan mucho en C, en parte debido a ellos son en ocasiones la única forma de expresar operaciones y a que por lo general llevan un código más compacto y eficiente de lo que se puede obtener en otras formas” (Kernighan, Ritchie, 1988, p.103). Cuando un puntero es declarado se reserva memoria para albergar una dirección de memoria, pero no para la información o valor a la que “apunta” el puntero. El espacio de memoria que se reserva para almacenar un puntero depende de la arquitectura del sistema operativo del tipo de dato al que apunte.

#### **2.1.2 Memoria Dinámica**

Las variables definidas de forma estática no pueden variar su tamaño durante la ejecución del programa, en cambio aquellas definidas de forma dinámica pueden modificar la cantidad de memoria que utilizan dependiendo de lo que se requiera a lo largo de la ejecución, incluso liberarla si es necesario; conllevando así a un programa más eficiente en términos de la memoria que utiliza. Para trabajar con memoria dinámica en C es necesario incluir la librería *stdlib.h*.

#### **2.1.3 Listas Enlazadas**

“Una lista enlazada es un tipo de dato abstracto que permite almacenar una secuencia de datos. A diferencia de los arreglos, el espacio ocupado crece o decrece dinámicamente a medida que se inserten o se eliminen nuevos datos. El tamaño de la estructura depende exclusivamente de los datos almacenados en un momento dado” (Saavedra, 2010, p. 19). Dichos datos no necesariamente deben ser primitivos, sino que también pueden crearse listas enlazadas de estructuras de datos más complejas. Su implementación puede realizarse por medio de arreglos dinámicos, punteros o cursores.

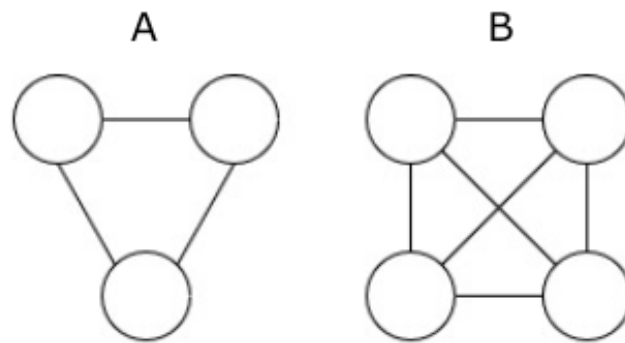
#### **2.1.4 Grafo**

Un grafo corresponde a la representación gráfica de distintos puntos denominados comúnmente nodos, los cuales se encuentran relacionados entre sí a través de flechas o líneas denominadas aristas, pudiendo estas estar orientadas o no.

Existen distintas formas de representar grafos en computación. Aquella que se utilice dependerá de las características que posea el grafo y los algoritmos que se necesiten implementar para su estudio. Dentro de las más empleadas se encuentran las listas y matrices, estas constituyen una manera sencilla de representar la relación entre los nodos y las aristas que les conectan; las primeras permiten un uso de memoria eficiente si se trata de grafos que poseen pocas aristas (grafos dispersos), mientras que las segundas aportan con un acceso rápido a la información del grafo pero, las cantidades de memoria consumidas pueden resultar demasiado elevadas debido a que estas crecen de manera cuadrática según la cantidad de nodos.

#### 2.1.4.1 *Clique*

Una clique corresponde a un conjunto de vértices en donde cada uno se encuentra relacionado por medio de una arista con los otros restantes.



*Ilustración 2.1 Ejemplos cliques*

La figura 2.1 ilustra dos grafos, A y B, que poseen cliques de 3 y 4 vértices respectivamente.

Todos los conceptos anteriores sirven de base para idear la posible solución al problema, debido a que es necesario simbolizar la red social mediante listas enlazadas y matriz de adyacencia. Si bien al momento de representar el grafo mediante una matriz no cabe otra mejor posibilidad que utilizar para ello un arreglo bidimensional, no ocurre lo mismo para las listas enlazadas; ya que estas se pueden representar al menos de tres diferentes maneras. Por ello es necesario escoger aquella que facilite su trabajo y en donde el acceso a los datos sea lo más expedito posible: arreglos dinámicos.



## **2.2 HERRAMIENTAS Y TÉCNICAS**

Se decide utilizar como técnicas para la resolución del problema recursividad y búsqueda en profundidad, además, se incluye una librería distinta a la de entrada-salida y manejo de memoria. Las razones se detallan a continuación.

### **2.2.1 Recursividad**

Esta técnica de programación simplifica la reiteración de procesos para los cuales no se conocen o han definido límites, como lo sería realizar búsquedas dentro de un grafo o que el resultado de un determinado proceso desemboque en un llamado a sí mismo. Se piensa utilizar recursividad para determinar el máximo subgrafo que se pueda obtener partiendo desde un determinado vértice.

### **2.2.2 Búsqueda en profundidad**

No se encuentra alguna característica que indique que búsqueda en anchura resulte una peor alternativa, pero puede darse el caso en que el grafo esté conectado como si se tratase de una lista enlazada (tal como el grafo A en la figura 2.1) y lo mejor en ese caso es recorrerlo en profundidad con el fin de agilizar el proceso de reconocer el máximo subgrafo.

### **2.2.3 Librerías**

Es importante señalar que se ha de incluir diferentes librerías dentro del programa. Además de `stdio.h` y `stdlib.h` cuyo uso resulta obligatorio, se requiere de `time.h`. Esta última permite definir instantes de tiempo con el fin de realizar el conteo de tiempo para los algoritmos en ambas implementaciones.

## 2.3 ALGORITMOS Y ESTRUCTURAS DE DATOS

### 2.3.1 Estructuras utilizadas

Para el manejo de la información se ha decidido definir las siguientes estructuras:

#### 2.3.1.1 *MatrizAdy*

La estructura *MatrizAdy* consiste en la representación de una matriz de adyacencia. Esta contiene un arreglo bidimensional binario y un número entero que indica el orden de la matriz. Su código se detalla a continuación.

```
typedef struct {  
    int** matriz;  
    int ordenMatriz;  
} MatrizAdy;
```

#### 2.3.1.2 *Lista*

Para la representación de listas de adyacencia se utiliza la estructura *Lista*, la cual posee un arreglo de estructuras *Adyacentes* y la cantidad de elementos de dicho arreglo. Cada elemento del arreglo contiene la información de los vértices que son adyacentes a uno en particular. Así, por ejemplo, si el grafo posee 5 nodos, la primera posición del arreglo de *Adyacentes* contendría un arreglo con todos los vértices adyacentes al primer nodo del grafo junto con su largo. Su código se detalla a continuación.

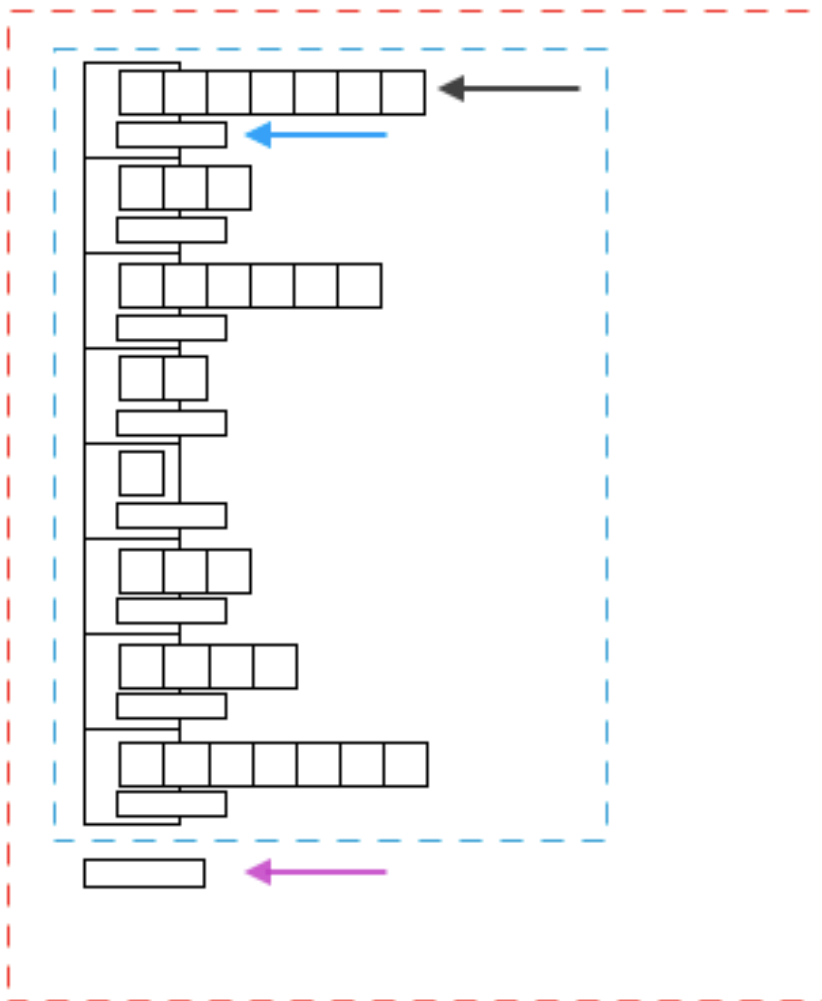
```
typedef struct {  
    Adyacentes* adyacentes;  
    int cantidad;  
} Lista;
```

#### 2.3.1.3 *Adyacentes*

Esta estructura posee un arreglo de números enteros, los cuales indican los vértices adyacentes a un determinado vértice, además de un número entero que indica la cantidad de elementos dentro del arreglo de enteros. Su código se detalla a continuación.

```
typedef struct {  
    int* vertices;  
    int largo;  
} Adyacentes;
```

El cómo se estructuran las listas de adyacencia mediante arreglos resulta más claro en la figura 2.2. En donde se distingue: con una línea punteada roja el espacio reservado para la estructura *Lista*, con una línea punteada celeste el espacio reservado para el arreglo de estructuras *Adyacentes*, con una flecha negra un arreglo que contiene los vértices adyacentes a un nodo (es la figura corresponde al primer nodo), con una flecha celeste el largo de dicho arreglo y con una flecha púrpura el largo del arreglo de *Adyacentes*.



*Ilustración 2.2 Representación gráfica de estructura Lista*

### 2.3.2 Algoritmos: Funciones más importantes

La mayor parte de las funciones definidas para resolver el problema funcionan de la misma manera tanto para la matriz de adyacencia como para las listas de adyacencia, debido a que en ambos casos la información está contenida en arreglos. Por ende, se describen a continuación las funciones más relevantes dentro del programa organizadas en pares, con el fin de explicar un funcionamiento que las describe por igual, dicho sea de paso que si el nombre de una función finaliza en “L” esta corresponde a aquella que utiliza para su operación listas de adyacencia, por el contrario, si el nombre finaliza en “M” la función utiliza una matriz de adyacencia; además junto a cada descripción se encuentra el bloque de código para ambas implementaciones.

#### 2.3.2.1 *obtenerCliquesL* y *obtenerCliquesM*

Las funciones *obtenerCliquesL* y *obtenerCliquesM*, entregan por pantalla los números de los vértices que conforman los cliques de 4 vértices, denominados “grupos de mejores amigos”. Para ello realiza cuatro ciclos *for* anidados generando todas las combinaciones posibles entre los vértices. Si en una combinación todos los vértices son distintos y al mismo cada uno es adyacente a los otros tres, dicha combinación se considera válida y debe mostrarse por pantalla. Los límites en los ciclos se reducen dependiendo del límite anterior para no imprimir todas las permutaciones que se pueden realizar con una combinación de cuatro vértices. Los bloques de código son los siguientes.

```
void obtenerCliquesL(Lista* lista) {
    int i, j, k, l;
    for (i = 0; i < lista->cantidad - 3; i++) {
        for (j = i + 1; j < lista->cantidad - 2; j++) {
            for (k = j + 1; k < lista->cantidad - 1; k++) {
                for (l = k + 1; l < lista->cantidad; l++) {
                    if (i != j && esAdyacente(i + 1, &(lista->adyacentes[j]))) {
                        if (i != k && esAdyacente(i + 1, &(lista->adyacentes[k]))) {
                            if (i != l && esAdyacente(i + 1, &(lista->adyacentes[l]))) {
                                if (j != k && esAdyacente(j + 1, &(lista->adyacentes[k]))) {
                                    if (j != l && esAdyacente(j + 1, &(lista->adyacentes[l]))) {
                                        if (k != l && esAdyacente(k + 1, &(lista->adyacentes[l]))) {
                                            printf(" %d, %d, %d, %d conforman un grupo de mejores\n", i+1, j+1, k+1, l+1);
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

void obtenerCliquesM(MatrizAdy* matriz) {

    int I, j , k , l;

    for (i = 0; i < matriz->ordenMatriz - 3; i++) {
        for (j = i+1; j < matriz->ordenMatriz - 2; j++) {
            for (k = j+1; k < matriz->ordenMatriz - 1; k++) {
                for (l = k+1; l < matriz->ordenMatriz; l++) {
                    if (i != j && matriz->matriz[i][j]) {
                        if (i != k && matriz->matriz[i][k]) {
                            if (i != l && matriz->matriz[i][l]) {
                                if (j != k && matriz->matriz[j][k]) {
                                    if (j != l && matriz->matriz[j][l]) {
                                        if (k != l && matriz->matriz[k][l]) {
                                            printf("  %d, %d, %d, %d conforman un grupo de mejores\n", i+1, j+1, k+1, l+1);
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

### 2.3.2.2 *obtenerVinculosL* y *obtenerVinculosM*

Estas funciones calculan la cantidad de grupos inicial y por medio de un ciclo *for* obtienen la cantidad de grupos (se considera grupo a aquel conjunto de vértices que forman un subgrafo) luego de eliminar cada uno de los vértices. Si el número de grupos que se encuentran luego de eliminar un vértice es mayor a la cantidad inicial, implica que dicho vértice corresponde a un agente vínculo y se imprime por pantalla el mensaje correspondiente, ya que ha dividido el grafo en dos o más partes. La cantidad de grupos iniciales se calcula y no se define con el valor 1 puesto que el grafo podría encontrarse desconexo. Los bloques de código se detallan a continuación.

```

void obtenerVinculosL(Lista* lista) {

    int i, gruposIniciales, gruposLuegoDeEliminar;

    gruposIniciales = obtenerGruposL(lista, lista->cantidad + 1);

    for (i = 0; i < lista->cantidad; i++) {
        gruposLuegoDeEliminar = obtenerGruposL(lista, i + 1);
        if (gruposIniciales < gruposLuegoDeEliminar) {
            printf("  %d es un agente de vínculo. Crea %d componente(s) conexas).\n",
                i + 1, gruposLuegoDeEliminar);
        }
    }
}

```

```

void obtenerVinculosM(MatrizAdy* matriz) {
    int i, gruposIniciales, gruposLuegoDeEliminar;

    gruposIniciales = obtenerGruposM(matriz, matriz->ordenMatriz + 1);

    for (i = 0; i < matriz->ordenMatriz; i++) {
        gruposLuegoDeEliminar = obtenerGruposM(matriz, i + 1);
        if (gruposIniciales < gruposLuegoDeEliminar) {
            printf("  %d es un agente de vínculo. Crea %d componente(s) conexas(s).\n",
                    i + 1, gruposLuegoDeEliminar);
        }
    }
}

```

### 2.3.2.3 *obtenerGruposL* y *obtenerGruposM*

Ambas funciones retornan un número entero igual a la cantidad de grupos que se encuentren en un grafo. Para realizar esta tarea definen un arreglo de enteros denominado “revisados” que en un inicio se encuentra vacío. Por medio de las funciones *conectarL* y *conectarM* recorre el grafo en profundidad desde un determinado vértice y “revisados” almacena todos los vértices que se han visitado en la búsqueda. Si luego de realizar dicha búsqueda aún hay vértices que no han sido agregados a “revisados” el proceso se repite una vez más. La cantidad de grupos será finalmente la cantidad de veces que se deba realizar la búsqueda en profundidad. Los bloques de código para ambas implementaciones se detallan a continuación.

```

int obtenerGruposL(Lista* lista, int eliminado) {

    int i, j, cantidadGrupos;
    int* revisados;
    revisados = (int *)calloc(lista->cantidad, sizeof(int));
    cantidadGrupos = 0;
    for (i = 0; i < lista->cantidad; i++) {
        if (i + 1 == eliminado) {
            continue;
        }
        if (!estaEn(revisados, i + 1, lista->cantidad)) {
            revisados[i] = i+1;
            cantidadGrupos ++;
        }
        for (j = 0; j < lista->adyacentes[i].largo; j++) {
            if (lista->adyacentes[i].vertices[j] == eliminado) {
                continue;
            }
            if (!estaEn(revisados, lista->adyacentes[i].vertices[j], lista->cantidad)) {
                revisados[lista->adyacentes[i].vertices[j] - 1] =
                    lista->adyacentes[i].vertices[j];
                conectarL(lista->adyacentes[i].vertices[j] - 1, lista, &revisados, eliminado);
            }
        }
    }
    free(revisados);
    return cantidadGrupos;
}

```

```

int obtenerGruposM(MatrizAdy* matriz, int eliminado) {

    int i, j, cantidadGrupos;
    int* revisados;

    revisados = (int *)calloc(matriz->ordenMatriz, sizeof(int));
    cantidadGrupos = 0;

    for (i = 0; i < matriz->ordenMatriz; i++) {
        if (i + 1 == eliminado) {
            continue;
        }
        if (!estaEn(revisados, i + 1, matriz->ordenMatriz)) {
            revisados[i] = i+1;
            cantidadGrupos ++;
        }
        for (j = 0; j < matriz->ordenMatriz; j++) {
            if (j + 1 == eliminado) {
                continue;
            }
            if (matriz->matriz[i][j] == 1) {
                if (!estaEn(revisados, j + 1, matriz->ordenMatriz)) {
                    revisados[j] = j + 1;
                    conectarM(j, matriz, &revisados, eliminado);
                }
            }
        }
    }
    free(revisados);
    return cantidadGrupos;
}

```

#### 2.3.2.4 conectarL y conectarM

Estas corresponden a algoritmos de búsqueda en profundidad adaptados para el problema actual. Mediante un ciclo *for* se recorren los vértices adyacentes al vértice entregado dentro de los parámetros. Si uno de estos vértices adyacentes no se encuentra en el arreglo “revisados”, es agregado y se realiza una llamada recursiva a la función utilizando como parámetro el vértice que se acaba de agregar. De esa manera una vez que ya no se pueda seguir recorriendo el grafo “revisados” contendrá los vértices que conforman un grupo. Los bloques de códigos son los siguientes:

```
void conectarM(int valor, MatrizAdy* matriz, int** revisados, int eliminado) {
    int x;
    for (x = 0; x < matriz->ordenMatriz; x++) {
        if (x + 1 == eliminado) {
            continue;
        }
        if (matriz->matriz[valor][x] == 1) {
            if (!estaEn(*revisados, x + 1, matriz->ordenMatriz)) {
                (*revisados)[x] = x + 1;
                conectarM(x, matriz, revisados, eliminado);
            }
        }
    }
}
```

```
void conectarL(int valor, Lista* lista, int** revisados, int eliminado) {
    int x;
    for (x = 0; x < lista->adyacentes[valor].largo; x++) {
        if (lista->adyacentes[valor].vertices[x] == eliminado) {
            continue;
        }
        if (!estaEn(*revisados, lista->adyacentes[valor].vertices[x], lista->cantidad)) {
            (*revisados)[lista->adyacentes[valor].vertices[x] - 1] =
                lista->adyacentes[valor].vertices[x];
            conectarL(lista->adyacentes[valor].vertices[x] - 1, lista, revisados, eliminado);
        }
    }
}
```

Para “eliminar” un vértice en cada una de las funciones que lo requiera se agregan sentencias *continue* para no considerarlo dentro de los procesos.



## CAPÍTULO 3. ANÁLISIS DE LOS RESULTADOS

El programa funciona tal y como se ha planeado, siempre y cuando se cumpla con las restricciones planteadas en el respectivo manual de usuario. Experiencias anteriores con lenguajes de programación como java o python indicaban que trabajar operatorias con arreglos de gran tamaño implicaba en un mayor tiempo de ejecución, algo que con C y en este programa no ocurre. Se realizaron pruebas con grafos de 100 o más nodos y los tiempos de ejecución son relativamente pequeños.

En cuanto a las falencias detectadas cabe mencionar que algunos algoritmos no son totalmente eficientes y podrían ser optimizados aún más, como lo es el caso de las funciones que se utilizan para saber si un elemento está dentro de un arreglo. En vez de recorrer dicho arreglo de manera lineal se podría realizar utilizando búsqueda binaria.

La implementación escogida para las listas de adyacencias resulta ser la adecuada ya que el tiempo de procesamiento que requeriría acceder a los datos por medio de punteros se deja completamente de lado utilizando arreglos dinámicos.

Por último es importante señalar los resultados de los tiempos y complejidades para las funciones más importantes del programa, los cuales se encuentran en la tabla

*Tabla 1 Tiempos de ejecución y órdenes de complejidad*

Función	Tiempo de ejecución	Orden de complejidad
obtenerQliquesL	$6n^5 - 54n^4 + 108n^3 + 324n^2 - 1458n + 1462$	$O(n^5)$
obtenerQliquesM	$13n^4 - 156n^3 + 702n^2 - 1404n + 1057$	$O(n^4)$
obtenerVinculosL	$(n + 1)T(\text{obtenerGruposL}) + n^2$	$O(n^4)$
obtenerVinculosM	$(n + 1)T(\text{obtenerGruposM}) + n^2$	$O(n^4)$
obtenerGruposL	$7n^3 + 12n^2 + T(\text{conectarL})$	$O(n^3)$
obtenerGruposM	$7n^3 + 12n^2 + T(\text{conectarM})$	$O(n^3)$
conectarL	$T(n - 1) + O(n^2)$	$O(n^3)$
conectarM	$T(n - 1) + O(n)$	$O(n^2)$

Las diferencias que se registran en los tiempos y complejidades entre las funciones aún cuando los procesos son prácticamente idénticos, se debe a que el tiempo que lleva saber si un nodo es adyacente a otro dentro de las listas de adyacencia tiende a  $n$ , lo que aumentaría el grado de complejidad en 1.

## CAPÍTULO 4. CONCLUSIONES

Los objetivos planteados para este laboratorio se cumplen de manera efectiva. Se logra desarrollar un programa que permite representar un grafo mediante listas de adyacencia y matriz de adyacencia. El funcionamiento del programa es el correcto en ambas plataformas (Windows y Linux) y los tiempos de procesamiento para ambas implementaciones no son altos, aún cuando dos de las funciones que trabajan con arreglos realizan cuatro ciclos *for* anidados.

Se está conforme con los resultados obtenidos y los conocimientos que este laboratorio a entregado, esperando mejorar en la investigación previa a la realización del código y la creación de funciones más compactas e intuitivas, que resulten más fáciles de entender.

Haber realizado este laboratorio resultó en innumerables frutos, sobretodo del punto de vista de los conocimientos asimilados. Tener idea de cómo representar un grafo y obtener la información que este posea puede llegar a ser en un futuro una herramienta muy eficaz a la hora de programar y enfrentar diversos problemas en donde la complejidad del algoritmo que se desea utilizar sea de vital importancia.

## **CAPÍTULO 5. REFERENCIAS**

Aho, A., Hopcroft, A., Ullman, J. (1995) Estructuras de datos y algoritmos. México: Pearson

Kernighan, B., Ritchie, D. (1988). El lenguaje de programación C. México: Pearson.

Santos E, J (1995-1999) Introducción al lenguaje C. España, Universidad de Las Palmas de Gran Canaria, Facultad de Informática. Recuperado de:  
[http://ecaths1.s3.amazonaws.com/laboratorio2pui/366213571.curso\\_c.pdf](http://ecaths1.s3.amazonaws.com/laboratorio2pui/366213571.curso_c.pdf)

## CAPÍTULO 6. MANUAL DE USUARIO

### 6.1 INTRODUCCIÓN

El programa descrito a continuación corresponde a la implementación de grafos mediante matriz de adyacencia y listas enlazadas, con el fin de representar una red social y permitirle al usuario obtener de ella aquellas personas que sean agentes de vínculo y los grupos de mejores amigos. Se consideran como agentes de vínculo a aquellas personas que al eliminarlas dividen la red en dos o más partes y grupo de mejores amigos a aquel grupo de cuatro personas en donde cada una está relacionada con las otras cuatro.

Este programa también señala los tiempos de ejecución utilizando matriz de adyacencia y listas enlazadas, generando la posibilidad de comparar la eficacia de ambas implementaciones.

Antes de poder utilizarlo es necesario que este sea compilado, para ello lo único necesario es tener instalado el conjunto de compiladores GCC. Todo lo relacionado a compilación y ejecución, funcionalidades del programa y posibles errores se detallan a continuación.

### 6.2 COMPILACIÓN

#### 6.2.1 Linux

Si se desea ejecutar desde un equipo con alguna distribución de Linux el primer paso es compilar el programa. Para esto es necesario abrir la terminal y posicionarse en el directorio en el cual se encuentra el archivo **RedesSociales.c**, una vez allí se debe proceder a ejecutar el siguiente comando:

```
$ gcc -o RedesSociales RedesSociales.c funciones.c -Wall
```

Una vez finalizada la compilación se generará un archivo con el nombre **RedesSociales**, que consiste en el ejecutable que permitirá utilizar el programa. Para acceder a este basta con utilizar el siguiente comando:

```
$ ./RedesSociales
```

### 6.2.2 Windows

Para compilar y ejecutar el programa en una distribución de Windows es necesario abrir la terminal y posicionarse en la carpeta que contenga el archivo **RedesSociales.c**, una vez allí se debe proceder a ejecutar el siguiente comando:

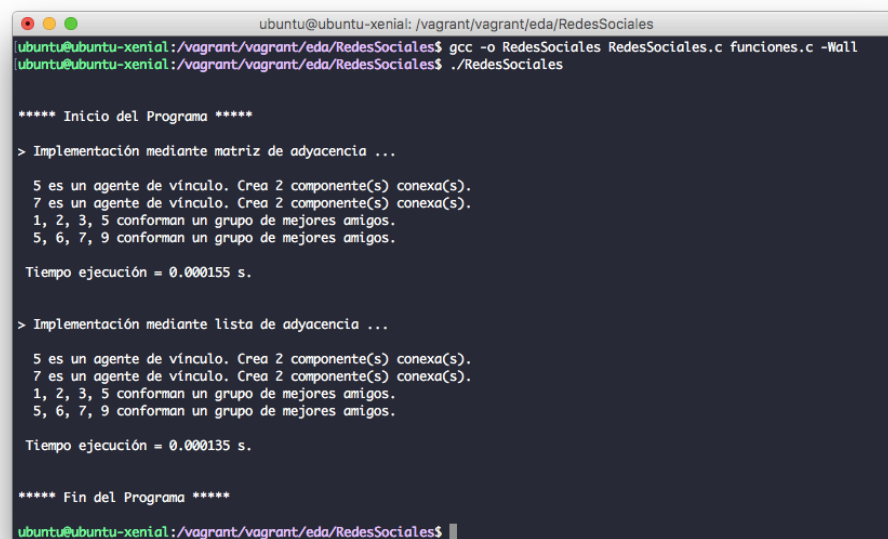
```
$ gcc -o RedesSociales.exe RedesSociales.c funciones.c -Wall
```

Una vez finalizada la compilación se generará un archivo ejecutable con el nombre **RedesSociales.exe**. Para acceder al programa basta con utilizar el siguiente comando:

```
$ RedesSociales.exe
```

## 6.3 FUNCIONALIDADES

Las funcionalidades del programa se realizan de manera automática por lo que el usuario solamente tendrá que esperar por los resultados. Estos aparecerán en la pantalla tal como en la figura 6.1



```
ubuntu@ubuntu-xenial: /vagrant/vagrant/eda/RedesSociales
ubuntu@ubuntu-xenial:/vagrant/vagrant/eda/RedesSociales$ gcc -o RedesSociales RedesSociales.c funciones.c -Wall
ubuntu@ubuntu-xenial:/vagrant/vagrant/eda/RedesSociales$ ./RedesSociales

***** Inicio del Programa *****

> Implementación mediante matriz de adyacencia ...

5 es un agente de vínculo. Crea 2 componente(s) conexas(s).
7 es un agente de vínculo. Crea 2 componente(s) conexas(s).
1, 2, 3, 5 conforman un grupo de mejores amigos.
5, 6, 7, 9 conforman un grupo de mejores amigos.

Tiempo ejecución = 0.000155 s.

> Implementación mediante lista de adyacencia ...

5 es un agente de vínculo. Crea 2 componente(s) conexas(s).
7 es un agente de vínculo. Crea 2 componente(s) conexas(s).
1, 2, 3, 5 conforman un grupo de mejores amigos.
5, 6, 7, 9 conforman un grupo de mejores amigos.

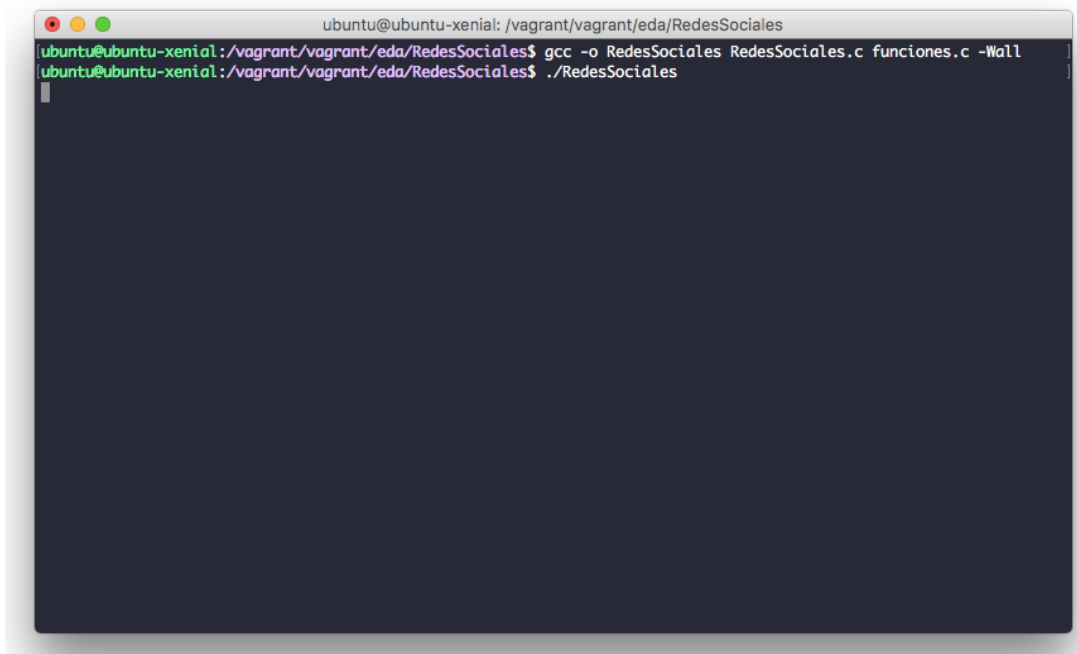
Tiempo ejecución = 0.000135 s.

***** Fin del Programa *****
ubuntu@ubuntu-xenial:/vagrant/vagrant/eda/RedesSociales$
```

Ilustración 6.1 Ejemplo de compilación del programa

## 6.4 POSIBLES ERRORES

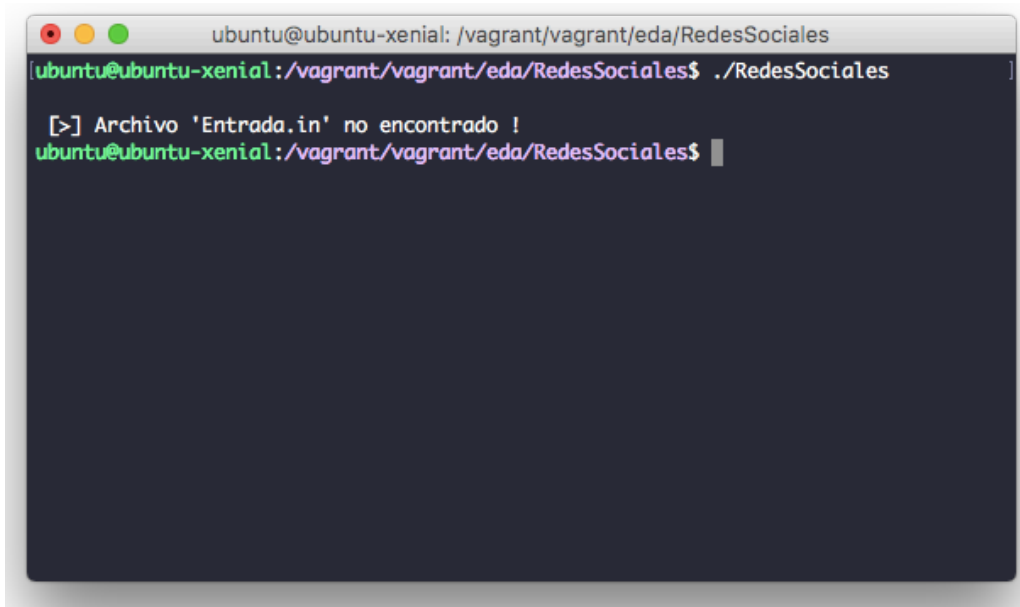
Un posible error puede ocurrir si el archivo ***Entrada.in*** que contiene el grafo posee un formato incorrecto. Este debe contener en la primera línea solo un número entero positivo que indica la cantidad de vértices en el grafo. En las líneas siguientes contiene solo las aristas, representadas por los vértices extremos separados por espacios. De ocurrir este error, el programa entraría en un ciclo infinito tal como se ve en la figura 6.2.



*Ilustración 6.2 Programa en ciclo infinito*

Para salir de ese ciclo se debe presionar la combinación de teclas Ctrl + C.

Otro posible error puede ocurrir si el archivo ***Entrada.in*** no se encuentra dentro de la carpeta del programa. De ser así, el programa entregaría mensaje por pantalla:

A terminal window with a dark background and light-colored text. The title bar at the top reads 'ubuntu@ubuntu-xenial: /vagrant/vagrant/eda/RedesSociales'. The prompt is 'ubuntu@ubuntu-xenial:/vagrant/vagrant/eda/RedesSociales\$'. The user has entered './RedesSociales'. The output is '[>] Archivo 'Entrada.in' no encontrado !'. The prompt is now 'ubuntu@ubuntu-xenial:/vagrant/vagrant/eda/RedesSociales\$' with a cursor at the end.

```
ubuntu@ubuntu-xenial: /vagrant/vagrant/eda/RedesSociales
[ubuntu@ubuntu-xenial:/vagrant/vagrant/eda/RedesSociales$ ./RedesSociales
[>] Archivo 'Entrada.in' no encontrado !
ubuntu@ubuntu-xenial:/vagrant/vagrant/eda/RedesSociales$
```

*Ilustración 6.3 Ejemplo error archivo no encontrado*

Para solucionar este error es necesario recuperar el archivo ***Entrada.in***.

Finalmente cabe recordar al usuario que no debe eliminar bajo ningún motivo alguno de los archivos dentro de la carpeta en donde se encuentra el programa, ya que lo anterior inhabilitará el programa de forma definitiva.