



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

Estructura de Datos y Análisis de Algoritmos

Laboratorio 2

Kevin Arévalo Fernández

Profesor: Pablo Schwazenbergriveros

Ayudantes: Diego Opazo
Javiera Saez
Javiera Torres

CONTENIDO

CAPÍTULO 1.	INTRODUCCIÓN	4
1.1	OBJETIVOS	4
1.1.1	Objetivo general	4
1.1.2	Objetivos específicos.....	4
CAPÍTULO 2.	DESCRIPCIÓN DE LA SOLUCIÓN	5
2.1	MARCO TEÓRICO.....	5
2.1.1	Entradas.....	5
2.1.2	Sobre estructuras de datos	5
2.1.2.1	Arreglos	5
2.1.2.2	Listas enlazadas	6
2.1.2.3	Grafos	7
2.2	HERRAMIENTAS Y TÉCNICAS	8
2.2.1	Herramientas.....	8
2.2.2	Técnicas	8
2.2.2.1	Initialize-Single-Source(G, s).....	9
2.2.2.2	Relax(u, v, w)	9
2.2.2.3	Dijkstra(G, w, s)	10
2.3	ESTRUCTURAS DE DATOS	11
2.3.1	Representación de un consultorio	11
2.3.2	Representación de la red de consultorios.....	11
2.3.3	Funciones y algoritmos.....	13
2.3.3.1	leerGrafo	14
2.3.3.2	ingresarPaciente.....	15
2.3.3.3	escribirRuta	16
CAPÍTULO 3.	ANÁLISIS DE RESULTADOS OBTENIDOS.....	17
CAPÍTULO 4.	CONCLUSIONES	18
CAPÍTULO 5.	REFERENCIAS	19

ILUSTRACIONES

Ilustración 2-1: Representación gráfica de un arreglo	6
Ilustración 2-2: Representación gráfica de una lista enlazada.....	6
Ilustración 2-3: Ejemplo de grafo no dirigido y sin pesos.	7
Ilustración 2-4: Ejemplo de lista de adyacencia de un grafo no dirigido.	7
Ilustración 2-5: Función Initialize-Single-Source.	9
Ilustración 2-6: Función Relax.	9
Ilustración 2-7: Algoritmo de Dijkstra.	10
Ilustración 2-8: TDA Nodo.	11
Ilustración 2-9: TDA Grafo.....	11
Ilustración 2-10: TDA ListaAdyacencia.	12
Ilustración 2-11: TDA NodoAdyacente.	12
Ilustración 2-12: Representación gráfica de la red de consultorios.....	12
Ilustración 2-13: Función leerGrafo.	14
Ilustración 2-14: Funcón ingresarPaciente.....	15
Ilustración 2-15: Función escribirRuta.	16
Ilustración 3-1: Menú principal del programa.	17

TABLAS

Tabla 2-1: Funciones del programa.....	13
--	----

CAPÍTULO 1. INTRODUCCIÓN

El Bulto Feliz, empresa de clínicas veterinarias conocida anteriormente, ha modernizado su sistema de registro de clientes mediante el software propuesto como solución y explicado en el Informe 1. Sin embargo, ha surgido un nuevo problema, esta vez relacionado con optimización de tiempo de traslado de pacientes a sus diversos consultorios distribuidos en distintos puntos de la región. El problema principal consiste en qué ruta escoger para derivar a un paciente desde un consultorio (que no posee la especialidad necesaria para realizar tratamiento) a otro que sí posea las herramientas, y como el tiempo es vital, por supuesto, esta ruta debe ser la más rápida. Además, el consultorio destino debe contar con un médico y espacio suficiente para atender al paciente.

Para solucionar el problema, se propone un software que sea capaz de leer la información de la red de consultorios, identificando todas las rutas disponibles entre ellos, especialidades y cupos, y mediante ésta, sea posible calcular la ruta más corta entre un consultorio origen (donde se encuentra el paciente) ingresado por el usuario, y un consultorio destino, que tenga la especialidad requerida (también especificada por el usuario) y el cupo suficiente.

1.1 OBJETIVOS

1.1.1 Objetivo general

- Diseñar y programar el software propuesto como solución al problema.

1.1.2 Objetivos específicos

- Desarrollar la solución utilizando grafos como estructura de datos principal.
- Lograr que el programa sea eficiente, en cuanto al tiempo de ejecución.
- Adquirir y/o mejorar habilidades de programación en el lenguaje C

CAPÍTULO 2. DESCRIPCIÓN DE LA SOLUCIÓN

2.1 MARCO TEÓRICO

Para resolver el problema de encontrar la ruta más corta entre dos consultorios dentro de una red, evidentemente, se necesita proporcionar al programa la información necesaria sobre la cual sea posible aplicar algoritmos resolutivos. Además de leer dicha información, es importante escoger con detalle, de qué forma ésta es almacenada y representada y luego, escoger una técnica eficiente para hallar la ruta más corta. Todos estos aspectos son abordados en el presente sub-capítulo y el siguiente.

2.1.1 Entradas

Como entradas al problema se consideran dos **archivos de texto plano**, que se definen como “aquellos formados exclusivamente por texto (sólo caracteres), sin ningún formato; es decir, no requieren ser interpretados para leerse (aunque pueden ser procesados en algunos casos). También son llamados archivos de texto llano, simple o sin formato”¹. Uno de ellos posee información acerca de los consultorios (Nombre, especialidad, cupos) y el otro proporciona información acerca de cómo están conectados entre sí, especificando el tiempo que demora el trayecto entre uno y otro.

Para resolver el sub-problema de cómo representar adecuadamente una red de consultorios, cabe como anillo al dedo la utilización de la **estructura de datos** conocida como **grafo**.

2.1.2 Sobre estructuras de datos

También llamada TDA, es un tipo de dato compuesto en que cada elemento no es necesariamente del mismo tipo que los demás. Son útiles a la hora de tratar como un único dato a un conjunto de ellos que guardan cierta relación determinada.² Ejemplos de ellos son los arreglos, las listas enlazadas y los grafos, que son, justamente las estructuras principales utilizadas para representar la información en el programa desarrollado. Antes de entrar en la definición de grafo, es necesario conocer las primeras dos estructuras mencionadas.

2.1.2.1 Arreglos

Un arreglo es una colección de variables del mismo tipo que se referencian por un nombre común.³ Los datos de un arreglo son fácilmente accesibles en el lenguaje C, ya que estos se referencian mediante un índice, que representa su posición en el arreglo. Existen arreglos estáticos: su tamaño no varía durante la ejecución del programa. Y también los hay dinámicos: su tamaño puede variar durante la ejecución del programa (aun así, el largo está explícitamente definido en todo momento).

¹ (Alegsa, 2010)

² (Köhler, 2017)

³ (Schildt, 1999)

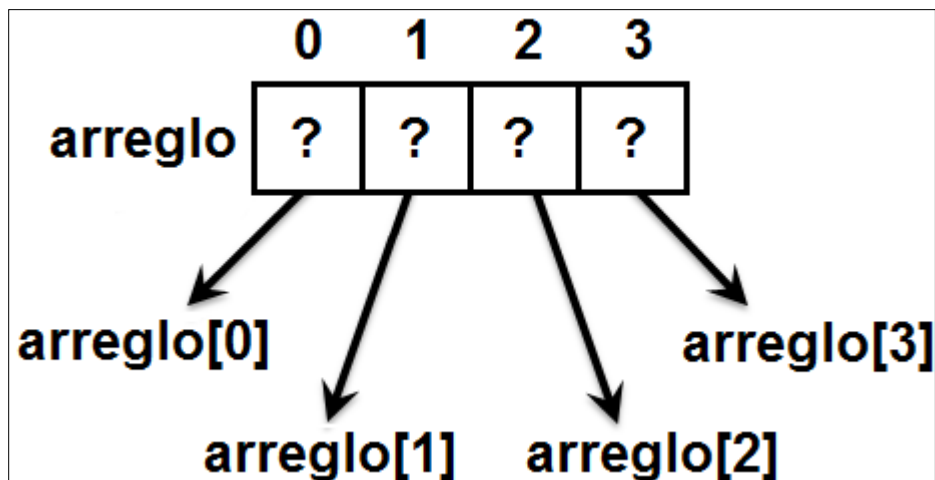


Ilustración 2-1: Representación gráfica de un arreglo

2.1.2.2 Listas enlazadas

Una lista enlazada es un TDA similar a un arreglo, pero con la ventaja de que no es necesario saber, a priori, el número de elementos que va a tener la lista, ya que estos se van agregando dinámicamente según se necesite. A diferencia de los arreglos, sus elementos no están explícitamente indexados, por lo que es más difícil acceder a ellos, y es preciso crear funciones auxiliares con dicho propósito.

Una lista enlazada se compone básicamente de un nodo que contenga los datos que se desea almacenar (que puede ser cualquier TDA), y además posee necesariamente un puntero al siguiente nodo de la lista. El último elemento posee un puntero a NULL, para indicar que es el fin de la lista.

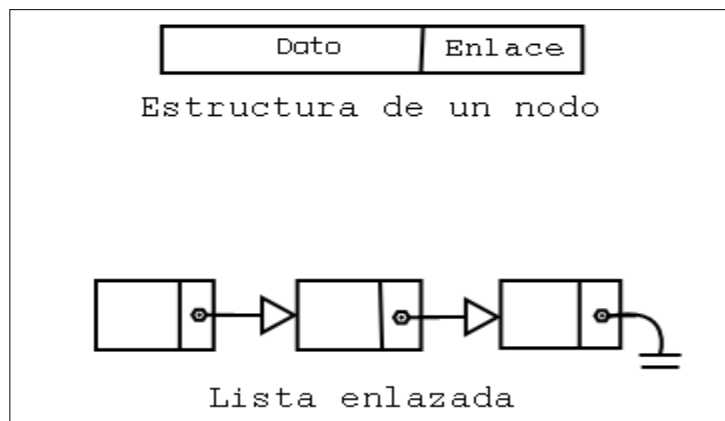


Ilustración 2-2: Representación gráfica de una lista enlazada.⁴

⁴ Imagen extraída de <http://www.calcifer.org/documentos/librognome/glib-lists-queues.html>

2.1.2.3 Grafos

Para las ciencias de la computación y la matemática, un grafo es una representación gráfica de diversos puntos que se conocen como nodos o vértices, los cuales se encuentran unidos a través de líneas o flechas que reciben el nombre de aristas.⁵ Existen grafos dirigidos y no dirigidos. En los grafos dirigidos, se especifica el sentido de las conexiones entre nodos, en cambio, en los grafos no dirigidos, esto no es así y se asume que, dada una conexión entre dos nodos, es posible llegar desde un nodo hacia el otro y viceversa. Además, un grafo puede poseer pesos en sus aristas, que son considerados como costos de llegar de un nodo a otro.

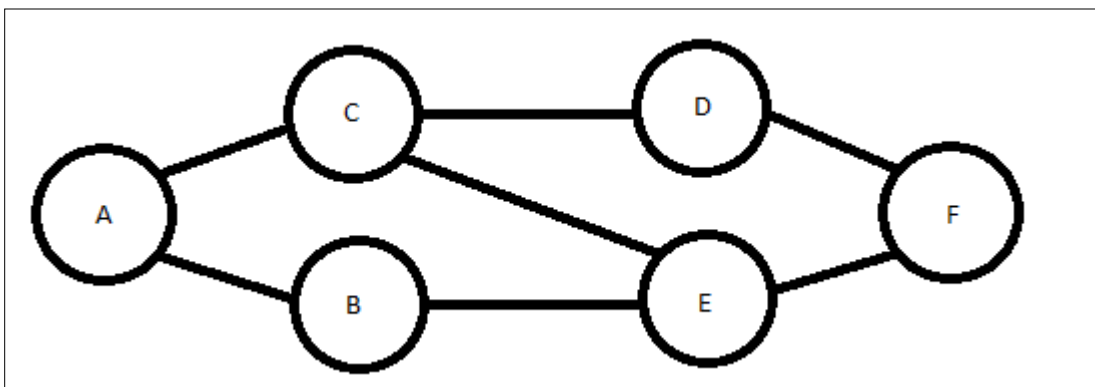


Ilustración 2-3: Ejemplo de grafo no dirigido y sin pesos.

Para representar un grafo, existen dos formas muy utilizadas, que son mediante una matriz de adyacencia o una lista de adyacencia. En este caso se utiliza la lista de adyacencia, que consiste en una lista de todos los nodos del grafo, y cada nodo posee otra lista con todos los nodos adyacentes a él (de ahí el nombre).

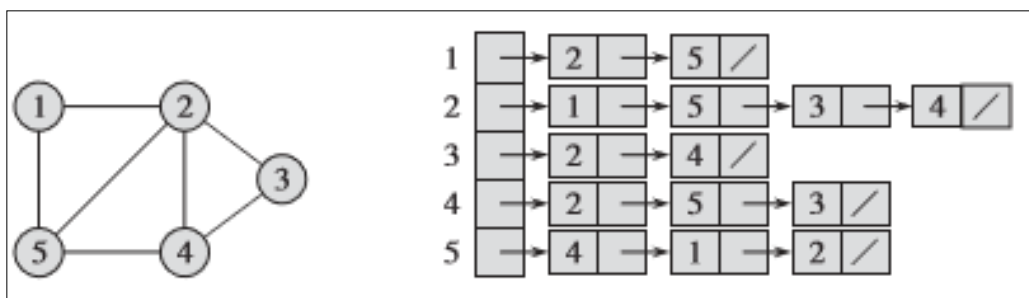


Ilustración 2-4: Ejemplo de lista de adyacencia de un grafo no dirigido.⁶

⁵ (Köhler, 2017)

⁶ (Cormen, 2009)

2.2 HERRAMIENTAS Y TÉCNICAS

2.2.1 Herramientas

Para llevar a cabo el proyecto, se utiliza el lenguaje de programación C, siguiendo el estándar ANSI-C para garantizar el correcto funcionamiento del programa en los sistemas operativos más utilizados. El compilador utilizado es GCC.

Se prioriza este lenguaje por sobre otros, debido al gran control que otorga a la hora de manejar la memoria del computador, lo que posibilita el desarrollo de un programa eficiente. Además, al ser tan estricto con los tipos de datos, se garantiza la correcta comprensión y aplicación de TDA's por parte del programador.

2.2.2 Técnicas

La técnica principal en la que se basa el algoritmo que resuelve el problema de hallar la ruta más corta (entendiéndose la más corta como aquella que demande menos tiempo, no menos distancia) entre dos consultorios, es el Algoritmo de Dijkstra, que resuelve el problema de encontrar el camino más corto entre un nodo origen y todos los nodos de un grafo con pesos asociados entre sus aristas.

Antes de explicar el funcionamiento de dicho algoritmo, es necesario realizar una serie de definiciones de conceptos utilizados por este algoritmo para representar a un grafo y los caminos más cortos entre sus nodos.

Dado un grafo $G = (V, E)$, en donde V es un conjunto de vértices o nodos, y E es un conjunto de aristas (del inglés, edge) que poseen pesos o costos asociados entre ellas. Entonces, para cada vértice $v \in V$ se mantiene un atributo $v.\pi$, que representa al antecesor de v , este atributo puede ser otro vértice $u \in V$ o NULL. El atributo π de cada vértice se establece de manera tal que la cadena de predecesores que se forma a partir de un vértice $\varphi \in V$ hasta llegar a un nodo $s \in V$, representa el camino más corto entre s y φ .

El algoritmo de Dijkstra utiliza la técnica de **relajación**, que consiste en establecer para cada vértice $v \in V$ un atributo $v.d$, que mantiene el peso o costo total acumulado de llegar desde un nodo $s \in V$ hasta el nodo v , siguiendo además el camino más corto entre ellos. Se define $v.d$ como una **estimación del camino más corto**.

A continuación, se describen las funciones necesarias para aplicar el algoritmo.

2.2.2.1 Initialize-Single-Source(G, s)

Inicialización de los atributos $v.d = \infty \forall v \in V - \{s\}$, $s.d = 0$ y $v.\pi = NULL \forall v \in V$.
En donde s es el nodo source o fuente

```

INITIALIZE-SINGLE-SOURCE( $G, s$ )
1  for each vertex  $v \in G.V$ 
2       $v.d = \infty$ 
3       $v.\pi = \text{NIL}$ 
4   $s.d = 0$ 

```

Ilustración 2-5: Función Initialize-Single-Source.⁷

2.2.2.2 Relax(u, v, w)

El proceso de relajar una arista (u, v) consiste en determinar si es posible mejorar el camino más corto encontrado hasta el momento para llegar a v , llegando a través de u . Si este es el caso, se actualizan los atributos de v : $v.\pi = u$, $v.d = u.d + w$. En donde w es el peso asociado entre u y v .

```

RELAX( $u, v, w$ )
1  if  $v.d > u.d + w(u, v)$ 
2       $v.d = u.d + w(u, v)$ 
3       $v.\pi = u$ 

```

Ilustración 2-6: Función Relax.⁸

⁷ (Cormen, 2009)

⁸ (Cormen, 2009)

2.2.2.3 Dijkstra(G, w, s)

Finalmente, con todas las definiciones previas, es posible indicar cómo funciona el algoritmo:

```

DIJKSTRA( $G, w, s$ )
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2   $S = \emptyset$ 
3   $Q = G.V$ 
4  while  $Q \neq \emptyset$ 
5       $u = \text{EXTRACT-MIN}(Q)$ 
6       $S = S \cup \{u\}$ 
7      for each vertex  $v \in G.Adj[u]$ 
8          RELAX( $u, v, w$ )

```

Ilustración 2-7: Algoritmo de Dijkstra.⁹

A medida que avanza el algoritmo, aquellos nodos a los que ya fue hallado su camino más corto, partiendo desde el nodo source, son agregados a la lista S . El algoritmo, originalmente termina cuando se ha encontrado los caminos más cortos para todos los nodos pertenecientes al grafo. Eso será modificado para el caso particular de resolver el problema en tratamiento, pues el objetivo es encontrar el camino más corto hacia un solo consultorio destino, no a todos los de la red.

⁹ (Cormen, 2009)

2.3 ESTRUCTURAS DE DATOS

En esta sección, se detallan las estructuras de datos utilizadas para formar el grafo que, eventualmente, represente a la red de consultorios de El Bulto Feliz.

2.3.1 Representación de un consultorio

Un consultorio se representa como un nodo, que formará parte de un grafo. El campo *padre* representa al atributo π y *tiempoAcumulado* representa a la estimación del camino más corto. Además, debido a las características del problema, es necesario almacenar el nombre del consultorio, cantidad de pacientes atendidos actualmente en él, y su capacidad máxima de pacientes por atender.

```
struct nodo{
    struct nodo* padre;
    char* nombreConsultorio;
    char* especialidad;
    int pacientesMaximos;
    int pacientesActuales;
    int tiempoAcumulado;
};
typedef struct nodo Nodo;
```

Ilustración 2-8: TDA Nodo.

2.3.2 Representación de la red de consultorios

La red de consultorios es representada mediante un grafo no dirigido utilizando una lista de adyacencia. Para dicho efecto, se definen las siguientes estructuras.

```
struct grafo{
    ListaAdyacencia** matrizAdyacencia;
    int numNodos;
};
typedef struct grafo Grafo;
```

Ilustración 2-9: TDA Grafo.

En donde *matrizAdyacencia* es una lista de *ListaAdyacencia*, estructura encargada de representar a un nodo (denominado origen) y contener una lista enlazada de nodos adyacentes a él, mediante otra estructura llamada *NodoAdyacente*, que se encarga de almacenar el nodo adyacente en cuestión y el costo (tiempo) entre él y el nodo origen.

```

struct listaAdyacencia{
    Nodo* origen;
    NodoAdyacente* inicio;
    NodoAdyacente* final;
    struct listaAdyacencia* siguiente;
    int numNodosAdyacentes;
};
typedef struct listaAdyacencia ListaAdyacencia;

```

Ilustración 2-10: TDA ListaAdyacencia.

```

struct nodoAdyacente{
    int tiempo;
    Nodo* consultorio;
    struct nodoAdyacente* siguiente;
};
typedef struct nodoAdyacente NodoAdyacente;

```

Ilustración 2-11: TDA NodoAdyacente.

A continuación, se expone una representación gráfica de cómo están unidas estas estructuras, para una mejor comprensión de la abstracción del problema.

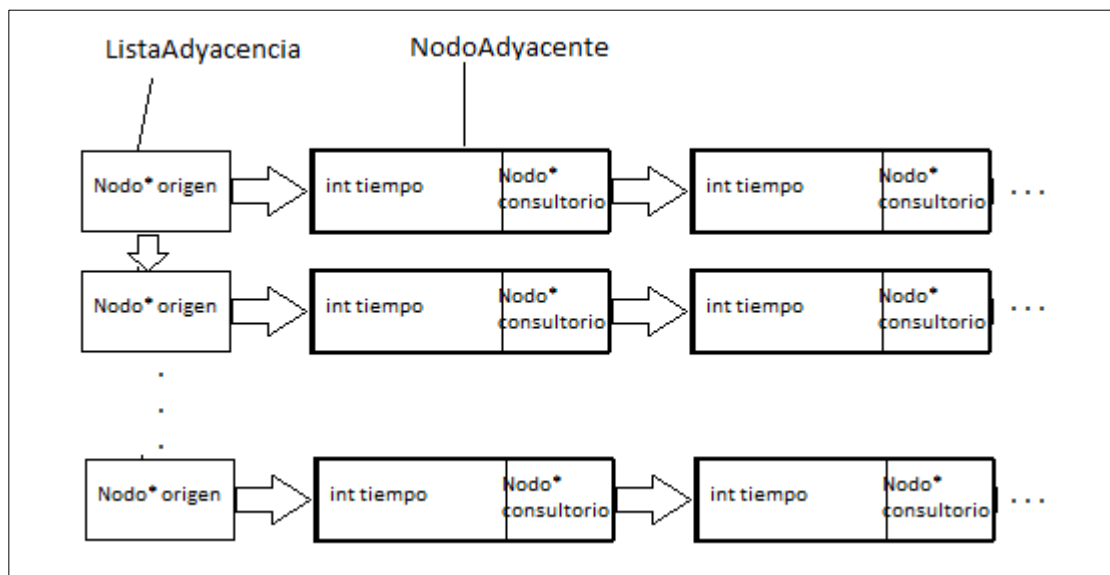


Ilustración 2-12: Representación gráfica de la red de consultorios.

Cabe aclarar que el campo *consultorio* de cada *NodoAdyacente* es un puntero al consultorio adyacente en cuestión (permitiendo obtener las características propias del mismo), y no hacia el siguiente *NodoAdyacente* de *origen*. La estructura *Grafo* encapsula toda la información interpretada en la ilustración, y, además, almacena el número total de consultorios de la red (que es la misma cantidad de Listas de Adyacencia que posee *matrizAdyacencia*).

2.3.3 Funciones y algoritmos

A continuación, se muestran todas las funciones del programa, considerando sus respectivas entradas, salidas y complejidad.

Función	Entradas	Salida	T(n)	O(n)
CrearNodo	<ul style="list-style-type: none"> Nodo* padre char* nombreConsultorio char* especialidad 	Nodo*	10	1
BuscarNodo	<ul style="list-style-type: none"> char* nombreNodo Grafo* g 	Nodo*	2n+1	n
AgregarAdyacente	<ul style="list-style-type: none"> Grafo* g char* consultorioOrigen char* consultorioAdyacente int tiempoTrayecto 	-	n^2+2n+5	n^2
leerGrafo	<ul style="list-style-type: none"> char* path char* pathAristas 	Grafo*	$n^4+n^3+10n^2+9n+32$	n^4
imprimirAdyacentes	<ul style="list-style-type: none"> ListaAdyacencia* lista 	-	3n+2	n
imprimirGrafo	<ul style="list-style-type: none"> Grafo* g 	-	n^2+n+4	n^2
relax	<ul style="list-style-type: none"> Nodo* u Nodo* v int w 	-	4	1
extractMin	<ul style="list-style-type: none"> ListaAdyacencia** matrizAdyacencia int largo 	ListaAdyacencia*	6n+6	n
ingresarPaciente	<ul style="list-style-type: none"> Grafo* g Nodo* s char* especialidad 	Nodo*	$14n^2+3n+2$	n^2
escribirRuta	<ul style="list-style-type: none"> Grafo* g Nodo* destino char* path 	-	11n+12	n
darDeAlta	<ul style="list-style-type: none"> Grafo* g char* consultorio 	-	5n+1	n
guardarConsultorios	<ul style="list-style-type: none"> Grafo* g char* path 	-	5n+3	n
liberarGrafo	<ul style="list-style-type: none"> Grafo* g 	-	$4n^2+n+1$	n^2

Tabla 2-1: Funciones del programa.

Las funciones más importantes, correspondientes a la lectura de los archivos y a la aplicación del algoritmo de Dijkstra, son detalladas a continuación.

2.3.3.1 leerGrafo

```

82 Grafo* leerGrafo(char* path, char* pathAristas){
83     #pragma region Grafo y Variables auxiliares ...
96     printf("##### LEYENDO GRAFO #####\n\n");
97     #pragma region Lectura de archivo y validacion de memoria...
105     g->numNodos = atoi(buffer);
106     printf("Num nodos: %d\n", g->numNodos);
107     g->matrizAdyacencia = (ListaAdyacencia**)malloc(sizeof(ListaAdyacencia)*(g->numNodos));
108     // Leer el salto de línea pendiente
109     fgets(buffer, sizeof(buffer), archivo);
110     // Este ciclo es para leer el archivo Consultorios.in y agregarlos al grafo
111     while (i < g->numNodos){
112         #pragma region Leer datos, crear un nodo y agregarlo al grafo...
140     }
141     fclose(archivo);
142     #pragma region Lectura segundo archivo y validacion de memoria...
150     numAdyacentes = atoi(valor);
151     fgets(valor, sizeof(valor), archivoAdj);
152     i = 0;
153     // Este ciclo es para agregar los consultorios adyacentes a cada consultorio
154     while (i < numAdyacentes){
155         #pragma region Leer datos, crear nodo adyacente y agregarlo al grafo...
169     }
170     #pragma region Liberando memoria...
178     return g;
179 }

```

Ilustración 2-13: Función leerGrafo.

Como se indica en los comentarios, el primer ciclo *while* es utilizado para cargar la información del archivo que contiene los nombres y características de todos los consultorios; dentro se utilizan instrucciones para leer la línea actual del stream y se utiliza la función *crearNodo*, por lo tanto, esta acción produce una complejidad del orden de n .

Para el caso del segundo ciclo *while*, utilizado para leer el archivo con la lista de todas las conexiones entre consultorios existentes y registrar dicha información en el grafo, se puede deducir que, en el peor caso (cuando todos los consultorios están conectados entre sí), y debido al formato de entrada del archivo, esta instrucción se repetirá $n(n - 1)$ ($O(n) = n^2$) veces. Dentro del ciclo se ejecutan un par de operaciones con complejidad constante y además se utiliza la función *agregarNodoAdyacente* (complejidad n^2).

Es por eso que $O(n) = n + (n^2)(n^2)$, de allí se deduce que la complejidad de esta función es $O(n) = n^4$.

2.3.3.2 ingresarPaciente

```

278 Nodo* ingresarPaciente(Grafo* g, Nodo* s, char* especialidad){
279     #pragma region Inicializar nodos como inaccesibles a excepción del origen...
296
297     // Copia del arreglo de ListaAdyacencia* del grafo
298     ListaAdyacencia** nodosSinOptimizar = (ListaAdyacencia**)malloc(sizeof(ListaAdyacencia*)*g->numNodos);
299     // Asignando los punteros del grafo al arreglo
300     for (int k = 0; k < g->numNodos; k++){ ...
302     }
303     int largoArreglo = g->numNodos;
304     // Mientras queden elementos en el arreglo
305     while (largoArreglo > 0) {
306         ListaAdyacencia* minim = extractMin(nodosSinOptimizar, largoArreglo);
307         largoArreglo -= 1;
308         // Comprobamos si el minimo es un consultorio de la especialidad
309         // buscada y ademas tiene cupo
310         if (strcmp(minim->origen->especialidad, especialidad) == 0 &&...
317         // Sino seguimos buscando
318         } else {
319             // En este caso el minimo no es el destino o no tenia cupo
320             NodoAdyacente* cursor = minim->inicio;
321             while (cursor != NULL){ // Para cada nodo adyacente al minimo
322                 Nodo* u = minim->origen;
323                 Nodo* v = cursor->consultorio;
324                 int w = cursor->tiempo;
325                 // Comprobamos si el camino para llegar a v puede ser mejorado
326                 relax(u, v, w);
327                 cursor = cursor->siguiente;
328             }
329         }
330     }
331     return NULL;
332 }

```

Ilustración 2-14: Función ingresarPaciente.

Esta es la función principal del programa, pues es la encargada de ejecutar el algoritmo de Dijkstra, acomodándolo al caso particular del problema.

Primero se crea una copia del campo *matrizAdyacencia* para poder trabajar sin alterar el grafo original. Esta operación se realiza en el primer ciclo *for* y claramente conlleva una complejidad de n , siendo n el número de consultorios.

Luego se comienza a aplicar el algoritmo de Dijkstra explicado anteriormente, pero con la diferencia de que el ciclo *while* encargado de este proceso se detiene cuando el nodo optimizado es un consultorio con la especialidad buscada. Este ciclo se ejecuta a lo más n veces.

En el interior del primer ciclo *while* se extrae el nodo con *tiempoAcumulado* mínimo, cuya complejidad es de n , y luego se comprueba la condición de término explicada. De no cumplirse la comprobación, se procede a aplicar la función *relax*, característica del algoritmo de Dijkstra, a todos los nodos adyacentes al mínimo extraído. Esta operación se repetirá a lo más n veces, es por eso que al anidar los dos ciclos, la complejidad de la función *ingresarPaciente* es de orden n^2 .

2.3.3.3 escribirRuta

```

342 void escribirRuta(Grafo* g, Nodo* destino, char* path){
343     Nodo* cursor = destino;
344     int contador = 0; // Aqui contaremos la longitud del camino
345     while (cursor != NULL){
346         contador++;
347         cursor = cursor->padre;
348     }
349     // Definiremos un arreglo de nodos
350     Nodo** camino = (Nodo**)malloc(sizeof(Nodo*)*contador);
351     // Agregamos los nodos del camino de forma inversa
352     cursor = destino;
353     int aux = contador-1;
354     while(cursor != NULL){
355         camino[aux] = cursor;
356         cursor = cursor->padre;
357         aux -= 1;
358     }
359
360     #pragma region Escribir camino en archivo...
381     // Liberar el camino
382     free(camino);
383 }

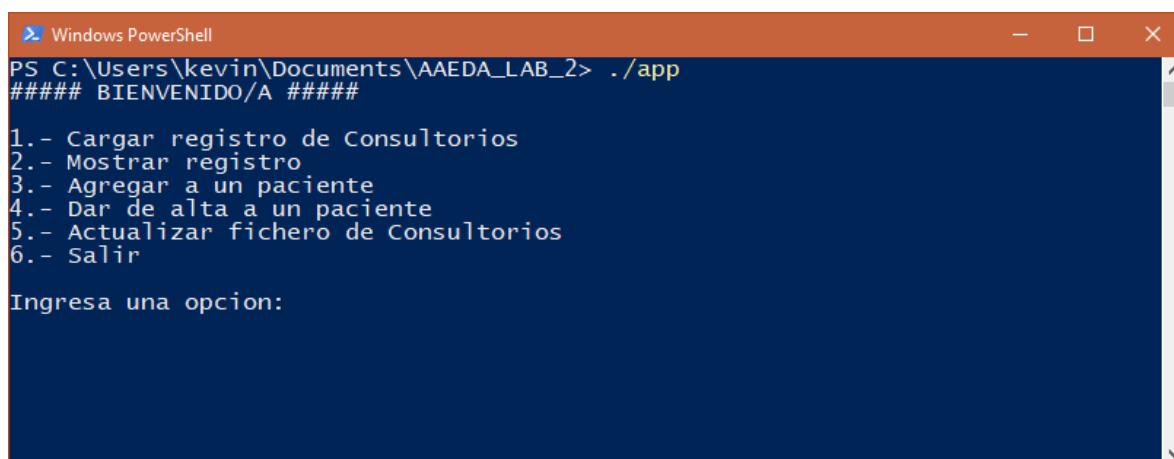
```

Ilustración 2-15: Función escribirRuta.

Para indicar el camino más óptimo encontrado (en términos de tiempo), se crea la función *escribirRuta*, que consiste básicamente en recorrer iterativamente los padres del nodo encontrado y retornado por *ingresarPaciente*, y hasta llegar a la fuente (aquel nodo con padre nulo). Todos estos nodos que representan el camino, son almacenados en un arreglo, que después se utilizará para escribir el resultado en el archivo de texto. Se aprecia claramente que la complejidad de esta función es de orden n .

CAPÍTULO 3. ANÁLISIS DE RESULTADOS OBTENIDOS

El resultado obtenido es un programa capaz de resolver el problema planteado, a través de la aplicación del algoritmo de Dijkstra adaptado a las necesidades específicas propias del problema.



```
Windows PowerShell
PS C:\Users\kevin\Documents\AAEDA_LAB_2> ./app
##### BIENVENIDO/A #####
1.- Cargar registro de Consultorios
2.- Mostrar registro
3.- Agregar a un paciente
4.- Dar de alta a un paciente
5.- Actualizar fichero de Consultorios
6.- Salir
Ingresa una opcion:
```

Ilustración 3-1: Menú principal del programa.

Con respecto a la eficiencia del programa, se aprecia que la función con mayor complejidad es de orden n^4 y corresponde al proceso de abrir y almacenar los datos de los archivos de texto en memoria. Esta situación resulta un poco inquietante, ya que se trata de un proceso secundario del programa que consume más recursos en términos de capacidad de procesamiento que la función principal *ingresarPaciente*. A pesar de ello, también es cierto que, para usar el programa, se necesita cargar la información una sola vez en su ejecución, pero es posible ingresar a más de un paciente.

El programa fue probado con distintos archivos, y falla cuando no se cumple con el formato especificado, o se ingresan tipos de datos distintos por parte del usuario en el menú. Exceptuando estos casos, si se dan otras situaciones, como, por ejemplo, solicitar el ingreso de un paciente desde un consultorio que no existe, o si no hay ningún consultorio con la especialidad requerida, el programa no falla y es capaz de informar al usuario que no es posible realizar el proceso solicitado.

Estos resultados reflejan que el correcto uso de grafos puede ser de gran utilidad. Además, cabe destacar que, para implementar grafos, es necesario poseer conocimientos acerca de otras estructuras como listas enlazadas principalmente, por lo tanto, se nota la correcta planificación de lo aprendido en la cátedra del curso.

CAPÍTULO 4. CONCLUSIONES

El objetivo principal: “Diseñar y programar el software propuesto como solución al problema”, se cumple, pues el programa obtenido posee las funcionalidades necesarias para cumplir con dicho propósito, y, además, se prueba con diferentes archivos, obteniendo los resultados deseados para cada uno de ellos.

Los objetivos secundarios también son cumplidos: se logra implementar la solución de utilizando grafos como estructura de datos principal.

Los grafos son esenciales y muy útiles para resolver problemas que se pueden abstraer como relaciones entre nodos. Además del algoritmo de Dijkstra, existen muchos otros vistos en clases, que utilizan representaciones similares para nodos y aristas, por lo tanto, con este proyecto se adquiere un aprendizaje con proyección al resto de la carrera y futuros problemas.

Indudablemente, el programa logrado no es perfecto, se puede agregar un sistema de verificación de entradas, tanto del archivo de texto, como de las entradas que ingresa el usuario por teclado. Además, en una situación real, sería necesario implementar una interfaz gráfica para una interacción más cómoda y eficiente con el usuario.

CAPÍTULO 5. REFERENCIAS

Calcifer.org. (2018). Estructuras de datos: listas enlazadas, pilas y colas. [online] Disponible en: <http://www.calcifer.org/documentos/librognome/glib-lists-queues.html> [Fecha de acceso: 20 de abril de 2018].

Köhler, J. (n.d.). APUNTES DE LA ASIGNATURA ANÁLISIS DE ALGORITMOS Y ESTRUCTURA DE DATOS. Santiago: Universidad de Santiago de Chile.

Alegsa, L. (2010). Definición de texto plano (archivo). Retrieved from http://www.alegsa.com.ar/Dic/texto_plano.php

Cormen, T. (2009). *Introduction to Algorithms* (3rd ed.). Massachusetts: Massachusetts Institute of Technology.