Informe >> Planificador

Geoffroy Nicolás, Iglesias Araceli

Planificador de viajes

09.10.2019

Organización de Computadoras

Tabla de Contenidos

1. Aspectos relevantes

1.1. Introduccion

Bienvenido a la documentación de Planificador, se dispone está presente para que en un futuro la aplicación pueda ser ampliada, por lo que se presenta información tanto de diseño como de implementación, completamente orientado a desarrolladores.

1.2. Invocación

El programa implementado, denominado *planificador* debe conformar la siguiente especificación al ser invocado desde la línea de comandos:

$ planificador <archivo_texto>

El parámetro archivo_texto, indica el archivo a partir del cual se conocerá la ubicación actual del usuario, y la lista de ciudades que desea visitar. En caso de que la invocación no sea la indicada, se mostrará un error y finaliza la ejecución.

2. Definiciones y especificaciones de requerimiento

2.1. Definición general

Planificador>> un programa desarrollado en C creado para ayudar a viajeros a calcular distancias entre ciudades y hallar caminos de conveniencia.

3. Especificaciones de requerimientos del proyecto

3.1. Requisitos generales

Este proyecto ha sido creado sobre la base de lo solicitado por la cátedra de Organización de Computadoras, con el objetivo de cumplir los requerimientos del proyecto presentado.

Las pautas seguidas fueron:

3.1.1. Uso cuidadoso y eficiente de la memoria, tanto para reservar (malloc), como para liberar (free) en el espacio asociado a las variables y estructuras

3.1.2. Almacenamiento de elementos del tipo genérico en la lista

3.1.3. Almacenamiento de entradas con clave y valor del tipo genérico, ordenadas en función a su prioridad. La prioridad de las entradas es determinada por la función de prioridad elegida por el usuario.

3.1.4. El programa principal toma como argumento por línea de comando el nombre de un archivo de texto y a partir de este determina la ubicación actual y las ciudades a visitar por el usuario, permitiendo luego realizar operaciones sobre estos datos.

3.2. Requisitos funcionales

El programa ofrece un menú de operaciones, con las que el usuario luego puede operar sobre los datos ingresados

3.2.1. Mostrar ascendente: permite visualizar el listado de todas las ciudades a visitar, ordenadas de forma ascendente en función de la distancia que existe entre la ubicación de estas ciudades y la ubicación actual del usuario.

3.2.2. Mostrar descendente: permite visualizar el listado de todas las ciudades a visitar, ordenadas de forma descendente en función a la distancia que existe entre la ubicación de estas ciudades y la ubicación actual del usuario.

3.2.3. Reducir horas manejo: permite visualizar un listado con el orden en el que todas las ciudades a visitar deben ser visitadas, de forma tal que el usuario ubicado en la ciudad de origen conduzca siempre a la próxima ciudad más cercana al origen, reduciendo las horas de manejo entre las ciudades visitadas, finalmente se debe listar la distancia total recorrida con esta planificación.

3.2.4. Salir: permite finalizar el programa liberando toda la memoria utilizada para su funcionamiento.

3.3. Información de autoría y Legacy del proyecto

Este proyecto forma parte de un desarrollo original. Ha sido desarrollado y testeado en el sistema operativo Linux SMP Debian 4.6.3.1, i686. Se asegura el correcto funcionamiento en esta plataforma. No ha sido testeado en otros sistemas, por lo cual, su correcta ejecución se ve ligada a la retro-compatibilidad que tenga dicha versión de Linux con la plataforma sobre la cual se esté ejecutando.

4. Especificaciones de procedimientos

4.1. Herramientas utilizadas

Este sistema fue desarrollado en el lenguaje de programación C, a través del entorno de desarrollo integrado Code::Blocks. Se trabajó en el sistema Operativo Linux SMP Debian 4.6.3.1, i686. Se usó la terminal integrada del sistema para la verificación de programa. La documentación técnica fue creada con la herramienta DoxyGen.

4.2. Planificación

Este proyecto establece dos estructuras de datos de almacenamiento y un programa principal. Se desarrollaron dos archivos de implementación y dos archivos de encabezado para modular lo desarrollado. Cada módulo funciona independientemente, por lo que fueron desarrollados *a priori* de manera paralela, exceptuando el planificador que requería de las estructuras de datos para funcionar. Se establecieron los encabezados (lista.h y colacp.h) de acuerdo a lo requerido y fue analizada la forma de resolver correctamente cada operación del mismo, contemplando el abanico de casos que presentaba cada función y así formar el algoritmo adecuado que incluye cada uno, testeando cada estructura inicialmente en borrador tratando de representar gráficamente las situaciones y luego procediendo a testearlo a nivel código. Esta situación fue repetida hasta el último momento para asegurar su correcto funcionamiento.

Una vez asegurado el adecuado funcionamiento de cada operación de las estructuras de datos, se continuó con la implementación del módulo planificador. Se desarrolló un menú de opciones para proveer una forma de ejecutar las operaciones brindadas por este módulo y se estudió cómo encarar la resolución de dichas operaciones. Se crearon en el proceso algoritmos y se implementaron los métodos necesarios. Luego se ejecutaron diversas pruebas de testeo hasta que todo resultó en un estado final y correcto.

4.3. Proceso de instalación y prueba

4.3.1. Obtención e instalación

Para realizar la ejecución del programa se debe ingresar por línea de comando. Se comienza posicionándose en la carpeta que contiene a los archivos que conforman el sistema. Una vez que se haya ingresado la carpeta se debe ejecutar la sentencia “gcc -Wall planificador.c lista.c colacp.c -o Planificador”. Se detalla como ejemplo esta sentencia, la cual ha sido ejecutada sobre el sistema operativo mencionado en *Herramientas utilizadas* a través del compilador gcc, que hace referencia a la creación del archivo ejecutable del programa denominado *planificador*. Para otros sistemas operativos y/o compiladores se debe ejecutar la sentencia adecuada al entorno. De esta forma se obtendrá el archivo ejecutable del programa.

4.3.2. Especificaciones de prueba y ejecución

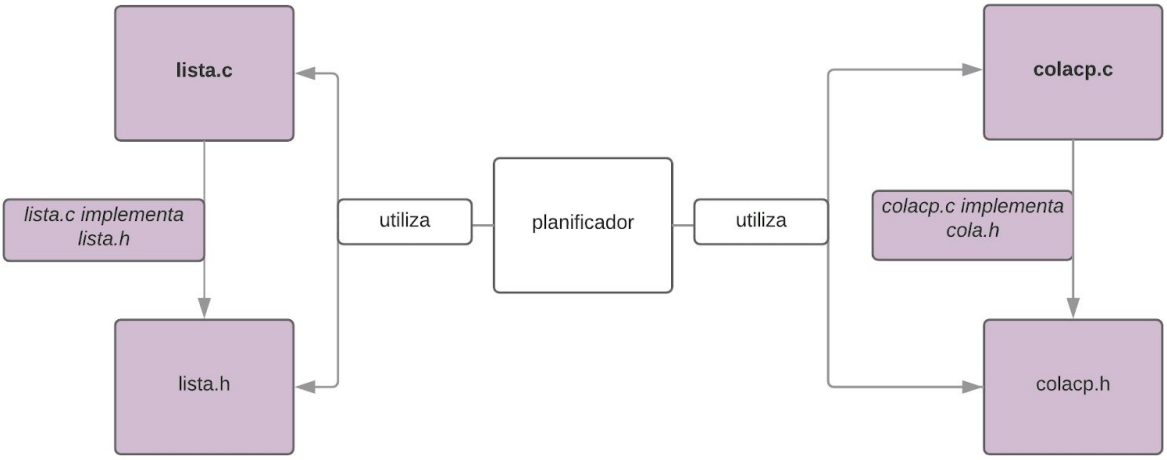
El programa funciona bajo especificado en *Herramientas utilizadas* por lo que no se asegura que bajo otros sistemas operativos el programa se ejecute correctamente.

5. Arquitectura del sistema

5.1. Descripción jerárquica

El programa fue desarrollado con una estructura monolítica, donde se tienen dos módulos pertenecientes a estructuras de almacenamiento y un tercer módulo núcleo (planificador) donde se lleva a cabo la función principal del programa mediante la utilización de recursos de librerías y de servicios provistos por los dos módulos de estructuras de datos.

5.2. Diagrama de módulos



5.3. Descripción individual de los módulos

5.3.1. Modulo lista

5.3.1.1. Descripción general y propósito

Es una estructura de datos que utiliza el modelo matemático de secuencia de celdas, y al ser doblemente enlazada, cada celda conoce su anterior y su posterior si es que existe el elemento. Permite almacenar información secuencial de elementos genéricos.

5.3.1.2. Responsabilidades y restricciones

Este módulo tiene la función de crear nuevas listas y ejecutar operaciones sobre las mismas, tales como insertar un elemento en una determinada posición, eliminar una determinada posición, retornar una posición solicitada, mostrar su tamaño, recuperar el elemento almacenado en una celda, y

destruir la lista en caso de que sea necesario.

5.3.1.3. Dependencias

Para que las operaciones de este módulo funcione (con excepción de la creación de la lista) es necesario que la lista argumentada esté inicializada, caso contrario se producirá una finalización anormal del sistema.
La lista se implementó con el uso de memoria dinámica, utilizando las funciones de librería *malloc* y *free*. También accede al archivo “lista.h” donde se encuentra la definición del tipo de dato lista.

5.3.1.4. Implementación

La implementación de esta estructura se encuentra en el archivo “lista.c” y la definición del encabezado dentro del archivo “lista.h”.

5.3.2. Modulo cola con prioridad

5.3.2.1. Descripción general y propósito

Es una estructura de datos que almacena una colección de elementos, llamados valores, los cuales tienen asociados una clave que es provista en el momento en el que el elemento es insertado. Cumple con la propiedad de orden total ya que utiliza una función de comparación que establece el orden entre prioridades.

5.3.2.2. Responsabilidades y restricciones

Tiene la función de crear nuevas colas con una función de prioridad dada, insertar elementos de forma arbitraria, eliminar el elemento de mayor prioridad, consultar la cantidad de elementos y destruirla si es necesario.

5.3.2.3. Dependencias

Para que las operaciones de este módulo funcione (con excepción de la creación de la cola) es necesario que la cola con prioridad argumentada esté inicializada, caso contrario se producirá una finalización anormal del sistema.
La cola se implementó con el uso de memoria dinámica, utilizando las funciones de librería *malloc* y *free*. También accede al archivo “colacp.h” donde se encuentra la definición del tipo de dato cola con prioridad.

5.3.2.4. Implementacion

La implementación de esta estructura se encuentra en el archivo “colacp.c” y la definición del encabezado dentro del archivo “colacp.h”

5.3.3. Modulo planificador

5.3.3.1. Descripción general y propósito

Implementa una aplicación de consola que recibiendo como argumento por línea de comando el nombre de un archivo de texto, compuesto por ciudad y sus respectivas coordenadas, muestra de forma ascendente y descendente las mismas dependiendo de su distancia y calcula la reducción de horas de manejo, esto es, el recorrido que se debe tomar para recorrer todas las ciudades de forma eficiente.

5.3.3.2. Responsabilidad y restricciones

Tiene la responsabilidad de brindar cuatro funciones: Mostrar Ascendente, Mostrar Descendente, Reducir Horas Manejo y Salir. Estas operaciones han sido detalladas en la sección *Descripción de procesos y servicios ofrecidos por el sistema.*

5.3.3.3. Dependencias

Utiliza los archivos **colacp.h, lista.h.**

5.3.3.4. Implementacion

La implementación del mismo se encuentra en el archivo planificador.c

5.4. Dependencias externas

5.4.1. Librerías utilizadas

Fueron utilizadas las librerias **stdio.h, stdlib.h, string.h**

<string.h>

- Strcmp: compara dos cadenas de caracteres.

<stdlib.h>

- exit(..): termina la ejecución del sistema con un valor entero argumentado.

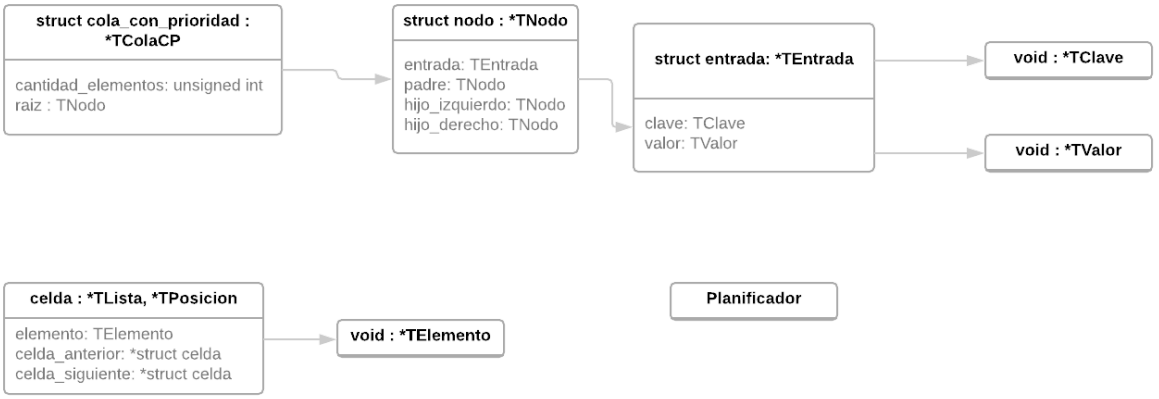
<stdio.h>

- printf(...): Imprime por pantalla un mensaje con un tipo de formato establecido y una cantidad acorde de variables para cada formato dentro del mensaje
- scanf(...): Lee la entrada de flujo de entrada estándar.
- fopen(const char *path, const char *mode): Abre el fichero cuyo nombre es la

cadena apuntada por path y asocia un flujo de datos a el; mode hace referencia a los permisos que se tendrá sobre el fichero.

- fclose(FILE *flujo): Disocia el flujo especificado de su fichero asociado.
- feof(FILE *flujo): Inspecciona el indicador de fin-de-fichero para el flujo indicado en *flujo*.
- fgetc(FILE *flujo): Lee el siguiente caracter de flujo y lo devuelve como un unsigned char modelado a un int, o EOF al llegar al final del flujo o en caso de error.

6. Diseño del modelo de datos



6.1. Datos internos

- Celda: almacena un elemento de tipo genérico, contiene un puntero a su celda anterior y su siguiente.
- Nodo: tiene un puntero a TEntrada, y tres punteros a su nodo padre, hijo izquierdo, e hijo derecho.
- Entrada: consiste de dos punteros, uno al tipo TClave y otro al tipo TValor, ambos genéricos.
- Cola_con_prioridad: define la estructura de una cola con prioridad, tiene una cantidad de elementos y un puntero a la raíz del heap.

6.2. Datos de entrada y salida

- TClave: puntero a un tipo genérico.
- TValor: puntero a un tipo genérico.
- TEntrada: puntero a entrada.
- TNodo: puntero a nodo.
- TElemento: puntero a un tipo genérico.
- TLista: puntero a celda. Apunta a la primera posicion de la lista.
- TPosicion: puntero a celda. Es utilizada para un mejor entendimiento de las posiciones directas de la lista.

7. Descripción de procesos y servicios ofrecidos por el sistema

7.1. Procesos ofrecidos

Una vez ejecutado el programa se dispondrá en la pantalla un menú listando las operaciones que se pueden ejecutar.

7.1.1. **Mostrar ascendente**: permite visualizar el listado de todas las ciudades a visitar, ordenadas de forma ascendente en función de la distancia que existe entre la ubicación de estas ciudades y la ubicación actual del usuario.
Crea una cola con prioridad y reserva la memoria a utilizar, con la función ascendente.
Imprime por consola las entradas removidas de mayor prioridad de la cola.

7.1.2. **Mostrar descendente**: muestra de forma descendente las ciudades del archivo.
Crea una cola con prioridad y reserva la memoria a utilizar, con la función descendente.
Imprime por consola las entradas removidas de mayor prioridad de la cola.

7.1.3. **Reducir horas manejo**: permite visualizar un listado con el orden en que todas las ciudades a visitar deben ser visitadas, de forma tal que el usuario ubicado en una ciudad de origen conduzca siemprea la próxima ciudad más cercana al originen, reduciendo las horas de manejo entre las ciudades visitadas. Finalmente, se debe listar la distancia total recorrida con esta planificación.

Utiliza simultáneamente distintas colas y distintas listas para poder garantizar la mayor prioridad posible y utilizando al máximo la propiedad de orden de la cola con prioridad.

7.1.4. **Salir**: permite finalizar el programa liberando toda la memoria utilizada para su funcionamiento.

Libera toda la memoria utilizada por el programa a través de las funciones destruir de ambas estructuras y free de todos los registros para los cuales se reservaron memoria.

7.2. Procesos implementados

7.2.1. LISTA

7.2.1.1. **void crear_lista(TLista* lista):** inicia la lista argumentada.

Lo apuntado por lista es igual a nulo.

7.2.1.2. **Int l_insertar(TLista* lista, TPosicion pos, TElemento elem):** inserta el

elemento **elem** en la posición **pos**, si la posición es nula, lo inserta en la primera posición de la lista, si la lista no está inicializada, se aborta el programa. Retorna verdadero si procede con éxito, falso en caso contrario.

Si la lista es nula
    Aborta el programa con lista no inicializada.
Reserva memoria para la nueva celda y le asigna su entrada
Si la posición es nula entonces
    Inserta la celda en la primera posición de la lista
Sino
    Si no es la última posición en la lista
        Asigna los punteros de celda anterior y celda siguiente a la nueva celda
    Sino
        La inserta como última celda de la lista

7.2.1.3. **Int l_eliminar(TLista* lista, TPosicion pos):** elimina el elemento de la posición y luego restablece el orden de la lista. Retorna verdadero si procede con éxito, falso en caso contrario.

Si la lista es nula
    Aborta el programa con lista no inicializada
Si la posición es la primera posición en la lista
    Lo apuntado por lista es la celda siguiente a la primera
Sino
    Si la posición es la última
        Elimina la referencia de el mismo en su celda anterior
    Sino
        Elimina la referencia de el mismo en su celda anterior y en su celda siguiente
Libera la memoria del elemento y de la posición

7.2.1.4. **TPosicion l_primera(TLista lista):** retorna la primera posición de la lista.

Retorna la posición apuntada por lista.

7.2.1.5. **TPosicion l_ultima(TLista lista):** retorna la última posición de la lista.

Retorna la última posición de la lista.

7.2.1.6. **TPosicion l_anterior(TLista lista, TPosicion pos):** retorna la posición anterior a la posición argumentada, si la posición es la primera posición en la lista, retorna POS_NULA.

Si la posición es la primera en la lista
    Retorna POS_NULA
Sino retorna el anterior a la posición

7.2.1.7. **TPosicion l_siguiente(TLista lista, TPosicion pos):** retorna la posición siguiente a la posición argumentada, si la posición es la última posición en la lista, retorna POS_NULA.

Si la posición es la última en la lista
Retorna POS_NULA
Sino retorna el siguiente a la posición.

7.2.1.8. **TElemento l_recuperar(TLista lista, TPosicion pos):** recupera el elemento de la posición argumentada, si la posición es nula retorna elemento nulo.
Si la posición es nula retorna elemento nulo
Sino retorna el elemento de la posición

7.2.1.9. **Int l_size(TLista lista):** retorna la cantidad de elementos de la lista.
Recorre la lista hasta el último elemento llevando el contador de la cantidad de elementos y lo retorna.

7.2.1.10. **l_destruir(TLista* lista):** elimina todas las celdas y libera la memoria utilizada, retorna verdadero si procede con éxito, falso en caso contrario. Si la lista no está inicializada aborta con lista no inicializada.
Si la lista es nula
Aborta el programa con lista no inicializada
Mientras haya posiciones en la lista
Elimina y libera el espacio de la primera posición y entrada de la lista y reemplaza lo apuntado por lista por la siguiente posición

7.2.2. COLA CON PRIORIDAD

7.2.2.1. **int (*comparador)(TEntrada p, TEntrada q):** Define el prototipo de función del comparador.

7.2.2.2. **TNodo buscarPadre(TColaCP cola, int cant):** busca recursivamente el último nodo
Si la cantidad es 1, el nodo es la raíz de la cola.
Sino llama recursivamente con la cantidad dividido 2
Si es impar, el nodo siguiente es el nodo derecho
Sino, el nodo siguiente es el nodo izquierdo
Si el siguiente no es nulo, el nodo a retornar es el siguiente
Retorna el nodo.

7.2.2.3. **void swap(TNodo p, TNodo q):** intercambia las entradas de los nodos argumentados.

7.2.2.4. **TNodo hijo_menor(TNodo padre):** retorna el hijo menor de un nodo
Asume que el hijo izquierdo es el menor
Si el derecho no es nulo y es menor que su hermano izquierdo, el derecho es el

menor
Retorna el hermano menor.

7.2.2.5. **void downHeap(TColaCP cola,TNodo n):** Intercambia de forma recursiva un nodo con el mayor de sus dos hijos si el nodo es menor que alguno de ellos.
Toma el hijo menor de n
Si el hijo menor no es nulo y es menor que su padre
Los intercambia y llama a la recursión con su padre.

7.2.2.6. **void upHeap(TColaCP cola, TNodo n):** intercambia de forma recursiva si un nodo es mayor que su padre.
Si n no es la raíz y n es mayor que su padre
Los intercambia y llama a la recursión con sí mismo.

7.2.2.7. **void crear_cola_cp(TColaCP* cola, int (*f)(TEntrada p, TEntrada q)):** reserva memoria e inicializa la cola con prioridad argumentada con cantidad de elementos 0 y raíz nula.

7.2.2.8. **int cp_insertar(TColaCP cola, TEntrada entr):** inserta la entrada en la cola, si la cola no está inicializada, aborta con cola con prioridad no inicializada.
Si la cola es nula
Aborta el programa con CCP_NO_INI
Reserva memoria para el nuevo nodo y lo arma con la entrada argumentada, padre, hijo izquierdo e hijo derecho nulos.
Si la cantidad de elementos es 0 entonces lo agrega como raíz
Sino busca el padre de la posición a insertar, pregunta cual es el hijo que no tiene y se inserta como su hijo.
Burbujea para arriba y restablece el orden de la prioridad.
Aumenta la cantidad de elementos en uno.

7.2.2.9. **TNodo ultimo_hijo(TNodo p):** retorna el último hijo de un nodo argumentado.
Asume que el último hijo es el izquierdo
Si el derecho no es nulo, el último hijo es el derecho
Retorna el último hijo.

7.2.2.10. **void eliminar_ultimo_hijo(TNodo n):** elimina el último hijo del nodo n.
Si el hijo derecho no es nulo lo elimina y libera el espacio en memoria
Sino, elimina y libera el espacio en memoria del hijo izquierdo.

7.2.2.11. **TEntrada cp_eliminar(TColaCP cola):** elimina la entrada de mayor prioridad de la cola y la retorna. Si la cola es nula aborta con cola con prioridad no inicializada.
Si la cola es nula

Aborta con CCP_NO_INI
Si la cantidad de elementos de la cola es 0 retorna ELE_NULO
Si la cantidad de elementos es 1
Remueve y libera el espacio en memoria de la raiz
Sino
Busca el último nodo ingresado en el heap e intercambia la entrada con la raíz
Elimina el último nodo y libera el espacio en memoria.
Burbujea para abajo y restablece el orden de la prioridad.

7.2.2.12. **int cp_size(TColaCP cola):** retorna la cantidad de entradas de la cola con prioridad argumentada.

7.2.2.13. **void destruir_rec(TColaCP cola, TNodo nodo):** elimina y libera la memoria de cada nodo de forma recursiva.
Si el nodo no es nulo
Si el hijo derecho no es nulo, llama recursivamente con el hijo derecho
Sino
Si el hijo izquierdo no es nulo, llama recursivamente con el hijo izquierdo
Libera la memoria de la entrada y del nodo.

7.2.2.14. **int cp_destruir(TColaCP cola):** destruye la cola y libera la memoria ocupada por ella. Si la cola es nula aborta con cola con prioridad no inicializada.
Si la cola es nula
Aborta con CCP_NO_INI
Si la cantidad de elementos es distinta de 0
Llama a destruir_rec con la cola y la raíz de la misma.
Libera la cola.

8. Documentación técnica - Especificación API

El proceso fue realizado por medio de la herramienta especificada por la cátedra **Doxygen**. Se puede visualizar el archivo **index.html** accediendo a la carpeta html ubicada en la carpeta raíz del proyecto.

9. Conclusiones

Tuvimos inconvenientes a la hora de crear la cola con prioridad ya que es necesario reservar la memoria a utilizar no en el inicio de la cola, sino en el planificador, ya que si se quería localizar en la creación de la cola el malloc, al no ser un doble puntero, el registro quedaba eliminado luego de la

finalización de la función.

En el proceso de desarrollo, aprendimos a organizar los tiempos de las tareas a realizar, el trabajo colaborativo, y la importancia de la bibliografía por sobre todas las cosas. La dicotomía que tuvimos para el desarrollo por querer seguir las consignas fue un problema, por las ambigüedades generadas, pero sobre el final, logramos que el programa funcione sin anormalidades.

También nos ha sido de mucha ayuda lo aprendido durante la carrera, por lo que tratamos de garantizar la modularización de las funciones y la claridad ante todo.