Trabajo Práctico n° 3:
*“*Censo 2022*”*

Alumnos:

- Nissero, Matías
- Avalos, Nazareno
- Gonzalez Cabral, Francisco

En el presente informe hemos de notificar la implementación acerca de la lógica de negocio desarrollada para este TP.

Podemos afirmar sin lugar a dudas, que pudimos terminar el tp sin mayores problemas. La experiencia adquirida en cuanto a grafos y algoritmos aplicados a estos mismos, nos ha ayudado no sólo a desarrollar e implementar un algoritmo goloso (como se indicaba en el enunciado) sino también un algoritmo de fuerza bruta.

Sin embargo, uno de los pequeños problemas que no supimos resolver es el hecho de recibir un archivo y leerlo. Por lo tanto, no pudimos implementar en la interfaz de usuario dicha característica para la aplicación.

Esto es una pena, puesto que teníamos la intención de agregar esta característica porque en el TP2, tampoco pudimos resolver el inconveniente.

Habiendo hecho mención a esa observación, continuamos con una breve descripción de la implementación propuesta.

Implementación de la lógica de negocio:

- *Class “RadioCensal”:*

Esta clase se dedica a la lógica de la aplicación. Para realizar esto, necesitamos un objeto tipo “Ciudad” y una lista de objetos tipo “Censista”.

Firma del método:	**Funcion:**
agregarConexionEntreManzanas(int manzana1, int manzana2)	Se agrega una “conexión” entre los dos índices de las manzanas como parámetro si es que son índices válidos.
agregarCensista(String nombre, int dni)	Se añade al censista a censista, siempre y cuando no haya un censista previamente añadido con el mismo dni y su nombre no esté vacío o sea null.
censar()	Censa a la ciudad y asigna a los censistas necesarios hasta tres manzanas o, en caso de no ser necesario, ninguna.
int cantDeCensistasAsignados()	Devuelve la cantidad de censistas que fueron asignados a una o más manzanas.
int cantDeManzanasCensadas()	Devuelve la cantidad de manzanas que serán censadas.
String censistasNombreYDNI()	Devuelve un String el cual contiene los nombres y dni de todos los censistas agregados previamente.
String censistasToString()	Devuelve un String con la información de todos los censistas agregados previamente (nombre, dni, manzanas a censar).
int getCantidadManzanas()	Devuelve la cantidad de manzanas que tiene la ciudad.
int cantCensistas()	Devuelve la cantidad de censistas agregados.

boolean estanConectadas()	Devuelve true si desde cualquier manzana se puede llegar a todas viajando entre las manzanas vecinas, de lo contrario devuelve false.
boolean hayCensistas()	Devuelve true si hay censistas registrados, devuelve false en caso contrario

Ciudad ciudad: Representa una ciudad modelada por un grafo, cada vértice representa una manzana de la misma.

ArrayList<Censista> censistas: ArrayList de censistas el cual se encarga de guardar la información de los mismos, este luego será utilizado para censar las manzanas y guardar información de censado. Cada índice de este ArrayList representa un censista de nuestro censo.

Sus operaciones son las siguientes:

Implementación de un grafo en contexto de una ciudad:

Comentarios de los alumnos:

Esta parte de la implementación nos llevó mucho tiempo debido a las diferentes ideas propuestas entre nosotros mismos. Al final se eligió la siguiente implementación porque, en la mayoría de los métodos, la complejidad es menor que en las propuestas descartadas. Además nos resultó más fácil de leer e interpretar.

- *Class "Ciudad":*

Esta clase tiene como objetivo modelar a la ciudad sobre la cual el usuario trabajará, para ello, utilizamos un grafo basado en una lista de adyacencia pero con algunos cambios en la implementación Los cambios están enfocados en intentar que la lectura del código sea más clara. Contiene los siguientes atributos:

int vertices: Entero el cual se encarga de medir la cantidad de manzanas que tiene la ciudad.

Manzana[] manzanas: ArrayList del objeto Manzana que se encarga de modelar el grafo, cada índice de este array se encarga de guardar la información sobre cada manzana.

Sus operaciones son las siguientes:

Firma del método:	**Función:**
agregarArista(int, int)	Agrega una arista entre los dos ID's de manzanas pasadas como parámetro, siempre y cuando sean ID's válidos, de no ser así, se devolverá una excepción.
boolean existeArista(int, int)	Devuelve true si las manzanas correspondientes a los ID's pasados como parámetros tienen una conexión entre ellas, en caso contrario devuelve false. Si los ID's son inválidos devuelve una excepción.
boolean fueCensada(int)	Devuelve true si la manzana con el ID pasado como parámetro fue censada, en caso contrario devuelve false. Si el ID es inválido devuelve una excepción.
censar(int i)	Censa la manzana con el ID pasado como parámetro, Si el ID es inválido devolverá una excepción.
desCensar(int i)	"Descensa" la manzana con el ID pasado como parámetro, Si el ID es inválido devolverá una excepción.
Set<Integer> vecinos(int i)	Dado un int 'i' se devuelve un Set de los vecinos de ese mismo 'i'. Si 'i' es inválido devolverá una excepción.

ArrayList<Integer> vecinosList(int i)	Realiza la misma acción que el método descrito anteriormente, con la diferencia que en vez de retornar un Set, retorna un ArrayList.
int getTamanio()	Devuelve la cantidad de manzanas que tiene la ciudad.
List<Integer> getVertices()	Devuelve los vértices en una lista.

Implementación de la clase Manzana:

- ***class "Manzana"***

La clase Manzana se ocupa de ser un vértice del grafo, en este caso, una manzana de la ciudad. Es la clase que complementa a la clase Ciudad teniendo dentro de ella la lista de sus vecinos, el array manzanas de la clase Ciudad termina siendo una especie de array de listas.

Esta clase modela las manzanas (o cuadras) de una ciudad: contiene un ID, una lista de sus vecinos y un boolean el cual indica si la misma fue censada o no.

Entonces, sus atributos son los siguientes:

int ID: ID de la manzana.
ArrayList<Manzana>: Lista de las conexiones con otras manzanas (aristas).
boolean fueCensada: Booleano el cual se ocupa de indicar si la manzana fue censada o no.

Sus operaciones son las siguientes:

Firma del método:	**Función:**
añadirArista(Manzana m)	Agrega al objeto manzana pasado como parámetro a la lista vecinos, en caso de que la manzana pasada como parámetro no sea válida se arrojará una excepción.
boolean existeArista(int m)	Devuelve true si manzanas contiene una manzana con el mismo ID que el int pasado como parámetro, en caso contrario, devuelve false. Si el ID pasado como parámetro es el mismo que this.ID devuelve una excepción.
censar()	Pone a fueCensada en true siempre y cuando no sea true previamente, en ese caso arrojará una excepción.
desCensar()	Pone a fueCensada en false.
boolean fueCensada()	devuelve fueCensada.

int getID()	devuelve ID.
Set<Integer> vecinos()	Devuelve un Set de Integer el cual contiene los ID de los vecinos de la manzana.
ArrayList<Integer> vecinosList()	Realiza la misma acción que el método descrito anteriormente, con la diferencia que devuelve un ArrayList<Integer> y los ID son ordenados de menor a mayor dependiendo si la manzana fue censada o no.

Implementación de la clase Censista:

- **class "Censista"**

La clase Censista se ocupa de modelar a una persona que censa. Por lo tanto, necesitamos tener en cuenta su dni y su nombre. Además, se le brinda al censista su propia lista de sus futuras manzanas a ser asignadas (esta lista está vacía pero será rellenada si es posible).

Sus atributos son los siguientes:

int dni: int el cual toma la función de identificar al censista. **String nombre:** String el cual se ocupa de darle nombre al censista. **ArrayList<Integer>:** ArrayList el cual se ocupa de registrar los ID de las manzanas que el censista se ocupará de visitar a la hora de censar.

Sus operaciones son las siguientes:

Firma del método:	**Funcion:**
asignarManzana(Integer)	Agrega al integer pasado como parámetro a 'manzanasACensar', en caso de que 'manzanasACensar' ya contenga el integer o si el tamaño de 'manzanasACensar' es 3 devolverá una excepción.
ArrayList<Integer> getManzanasACensar()	Devuelve un ArrayList de 'integers' el cual contiene los ID de las manzanas que el censista visitará.
String nombre()	Devuelve el nombre.
int dni()	Devuelve dni.
int cantManzanas()	Devuelve la cantidad de manzanas que el censista se ocupará de censar.
boolean censaManzanas()	Devuelve true si el censista tiene asignado aunque sea una manzana, en caso contrario devuelve false.

resetManzanas()	Vuelve a iniciar manzanasACensar, por ende, eliminando previo registro de manzanas.
String nombreYDNI()	Devuelve un String el cual solo contiene el nombre y dni del censista.

Implementación de un algoritmo goloso:

- ***class "SolverGoloso"***

<u>Comentarios de los alumnos:</u>

Esta fue la parte más complicada de todo el trabajo práctico. Nos llevó varios días pensar el algoritmo; desde su implementación hasta las clases que facilitan el mismo.

Al igual que la clase "Ciudad", para este algoritmo tuvimos varias ideas entre nosotros (los alumnos). Luego de haber analizado las ideas propuestas, nos decantamos por esta implementación del algoritmo, ya que, su complejidad era menor que las demás.

Tratándose de una heurística, nuestro objetivo siempre fue obtener una solución lo más aceptable posible sin olvidarnos de la complejidad. Por ello mismo, nos aprovechamos de ordenar todas las listas para que siempre al hacer "lista.get(0)" nos de el objeto más conveniente.

Por ejemplo, ordenar a los censistas dependiendo de la cantidad de manzanas que tiene que censar, o en el caso de las manzanas, ordenarlas dependiendo de si fueron censadas o no.

Esta clase se ocupa de, teniendo un objeto "Ciudad" y una lista de censistas, asignar manzanas de la ciudad a los censistas y censarlas.

Para realizar esto, la clase cuenta con un objeto Ciudad, un ArrayList de censistas y un 'int' para guardar el ID de la última manzana durante los ciclos del algoritmo. Sus atributos son los siguientes:

<u>Ciudad g:</u> Ciudad la cual sus manzanas podrán ser censadas, además, también podrán ser asignadas a algún censista de censistas.

<u>ArrayList<Censista> censistas:</u> Lista de censistas los cuales podrán ser asignados con hasta tres manzanas de g. **<u>int ultimaManzana:</u>** Int el cual guardará durante la ejecución el ID de la última
manzana en ser censada.

Operaciones:

Firma del método:	**Función:**

SolucionGoloso resolver()	Este método se encarga de asignar a los censistas de censistas a como máximo tres manzanas y de censar las manzanas de g. Retorna un objeto SolucionGoloso.

Implementación solucionGoloso:

- class “SolucionGoloso”

Esta es una clase complementaria del solver Goloso, la cual guarda a un objeto “Ciudad” y un ArrayList de censistas. Se utiliza para poder retornar el objeto Ciudad y la lista de censistas al terminar con la ejecución de la operación “resolver()” del solver. Sus atributos son:

Ciudad g: Objeto ciudad el cual en la clase SolverGoloso sería el resultado de utilizar el método resolver().

ArrayList<Censista> censistas: ArrayList de censistas, el cual en la clase SolverGoloso sería el resultado de utilizar el método resolver().

Sus operaciones son las siguientes:

Firma del método:	**Funcion:**
Ciudad getCiudad()	Devuelve g
ArrayList<Censista> getCensistas	Devuelve censistas

Implementación Solver Fuerza Bruta:

- ***class SolverFuerzaBruta:***

Este solver en particular, no nos llevó poco tiempo en implementar, porque usamos la misma lógica de encontrar TODAS las combinaciones posibles entre vértices usando recursión. Adaptamos el método recursivo del problema de la mochila (mostrado en clase y videos) a nuestro problema.

Sin embargo, una solución factible o válida solamente son aquellas combinaciones de vértices que son contiguas entre sí o tienen un vértice contiguo en común. Y por sobre todas las cosas, esas combinaciones tienen que ser menores o iguales a 3 elementos. Es decir, no pueden haber más de 3 elementos por combinación.

Luego, dichas combinaciones de vértices lo hemos llamado “GrupoDeVertices”, la solución factible de posibles combinaciones de GrupoDeVertices la hemos llamado “SolucionParticular” y la solución óptima (o LA mejor solución) la llamamos “SolucionGeneral”.

Entonces…

- Un grupo de vértices no tiene vértices repetidos, es decir, no existe un vértice v1 tal que un vértice v2 y un vértice v3 sean iguales:
- v1 != v2,
- v1 != v3, - v2 != v3.
- Ej: (0,1,2) o (2,4,5) o (6,1,3)

- Una solución particular tiene varios grupos de vértices que no son iguales, es decir, los grupos de vértices propuestos como posible solución no tienen vértices repetidos (de lo contrario no es una posible solución).
- [(0,1,2) (3,4,5) (6,7,8) (9)]

- Una solución general (o solución óptima) es aquella solución particular de una combinación de grupos de vértices específica tal que la cantidad de grupos de vértices posibles es la menor de todas las combinaciones de grupos. Por ejemplo:
- [(0,1,2) (3,6,5) (4,7,9) (8,10, 11)] < [(0,1,2) (3,4,5) (6,7) (8,9) (10,11)] size() = 4 < size() = 5

Finalmente…

- El solver necesita la información del grafo que representa a la ciudad y una lista de censistas.Tiene como una operación principal “*resolver()*” que devuelve un objeto tipo “*Solucion*”.

- El método recursivo tiene la siguiente firma: “*generarDesde*(**int** *indice*)”.

- La lógica para generar las posibles soluciones particulares está dentro del método "*buscarMejorSolucion()*".
- **Nota del alumno que lo implementó:** "*No se me ocurrió como abstraer esta operación sin causar más ilegibilidad al leer el código*".

- La lógica para decidir o quedarse con la mejor de las soluciones particulares se encuentra en "*buscarMejorSolucion()*".
- *Sabiendo que el criterio de '**mejor solución**' es preguntarse "**¿cuál es la solución particular que tiene la menor cantidad de grupos de vértices?**".*

- Implementamos una clase auxiliar (así como también se hizo en el problema de la mochila) para verificar que el grupo de vértices generado recursivamente es válido.

- Implementamos en la clase auxiliar, un método para clonar esos grupos de vértices válidos y guardar el clon en la clase "*GrupoDeVertices*".

- La clase "*GrupoDeVertices*" es un TAD que contiene una lista. Dicha lista, solamente puede recibir un "Integer" que haga referencia a un vértice válido del grafo.

- La clase "*SolucionParticular*" es un TAD que contiene un **Set**<*GrupoDeVertices*>. Dicha clase recibe grupos de vértices para ser añadidos al Set.

- La clase "*SolucionGeneral*" es un TAD que contiene un **Set**<*GrupoDeVertices*>.
- No confundirse con la clase de la solución particular, puesto que la solución general no recibe grupos de vértices para añadirse, sino un objeto tipo "*SoluciónParticular"* para cambiar '*this.set*' por el de '*SolucionParticular.set*'.