Universidad Nacional de General Sarmiento

Tecnicatura Universitaria en Informática
Licenciatura en Sistemas

Carrera(s)

Programación 3

Comisión 1 S1 2022

Materia

Trabajo Práctico 3

¡El TP del Censo 2022!

Patricia Bagnes y Javier Marenco

Docente(s)

APELLIDO Y NOMBRE	LEGAJO	EMAIL
Tula, Ignacio Mariano	35.226.620/2014	itula@logos.net.ar
Valente, Franco Gabriel	42.645.783/2019	valente.franco@hotmail.com

https://github.com/logos914/TUI_UNGS_-_13_-_Progra_3_-_TP_3_Censo__VALENTE_TULA_-_Com1_S1_2022

Link a repositorio

Trabajo Práctico 3

Índice

Introducción

En el presente informe presentaremos el código que se implementó para la creación del programa "Censo 2022", junto a sus debidas explicaciones en cuanto a las decisiones del mismo.

El mismo fue creado utilizando un patrón de diseño "MVC" (Model, View, Controller) para el uso de un "controlador" que mediara entre las clases del "Frontend" y las del "Backend", para que el primero no utilizase código de negocio en lo absoluto.

El programa consiste en asignar manzanas de un radio censal a la menor cantidad de censistas. Para lo cual en principio debe importarse un radio censal, el que luego podrá verse en el mapa. Y finalmente, se asignan las manzanas a tantos censistas sean necesarios para cubrir dicha área o radio censal. Cada censista es representado por un número, y las manzanas poseen cada uno un número indicando a que censista fue asignada.

Del Criterio Goloso

Para obtener una respuesta exacta y que asegure ser la mínima cantidad de censistas requerida, y que no exista una menor cantidad. Sería necesario obtener un recorrido óptimo en el sentido de que la asignación, permita siempre darle 3 manzanas a todos los censistas. Lo cual, a veces por cuestiones geográficas no es posible, porque no todos los radios censales son uniformes, no todas las manzanas tienen la misma cantidad de manzanas contiguas y en misma disposición. Lo que incrementa la dificultad o la complejidad computacional de conocer si es posible asignar exactamente un número de censistas divisor de la cantidad de manzanas.

En nuestra solución el criterio goloso aplicado consiste en centrarnos en cada censista y asignarle la cantidad máxima de manzanas posible, intentando que dicho número sea tres siempre que se cumpla la condición (al menos una manzana debe ser contigua a las otras dos). Es decir, en formar una solución local (el actual censista que recibe asignación de manzanas) con el máximo posible, a través de elegir siempre una manzana contigua a la anterior.

Este criterio relega la decisión de cuáles serán las próxima manzanas a elegir a la combinación de: El órden en que se importaron al radio censal, las manzanas y sus adyacencias, es decir como se construyó el grafo; Sumado también a las condiciones de no repetir o utilizar manzanas asignadas.

La miopía de este criterio goloso se encuentra en que no ve a largo plazo si el camino formado por la asignación de las manzanas conduce a una distribución óptima. En algunos casos, esta miopía conduce a que el camino que se va formando por la asignación se encierre, deje censistas con una o dos manzanas asignadas, sin posibilidad de asignar una tercera por falta de adyacencia, a la vez que siguen existiendo otras manzanas disponibles para asignar en el radio censal.

Clases

Dentro del paquete "Censo" encontraremos la siguiente clase:

- "Censo" que actúa como clase "Main" del programa.

Dentro del paquete "Controlador" encontraremos la siguiente clase:

- "Controlador" que actúa mediando entre el frontend y el modelo.

Dentro del paquete "Frontend" encontraremos las siguientes clases:

- "BackgroundPane" asiste en colocar imágenes en el fondo de un Jframe
- "Frontend" maneja la creación de elementos de la interfaz gráfica y el llamado a funciones del controlador a través de los eventos.

Dentro del paquete "Grafo" encontraremos las siguientes clases:

- "ArbolGeneradorMinimo"
- "ArbolGeneradorMinimoTest"
- "Arista"
- "AristaTest"
- "BFS"
- "BFSTest"
- "Distancia"
- "DistanciaTest"
- "Grafo"
- "GrafoTest"
- "Nodo"
- "NodoTest"

Dentro del paquete "Modelo" encontraremos la siguientes clases:

- Dentro de dicho paquete hallamos las clases:
    - "Censista".
    - "Manzana".
    - "RumboCardinal".
- Otro Paquete dentro de Modelo, llamado "georreferenciable", dentro del cual hallamos además dos carpetas "externo" y "geométrico".
    - Modelo georreferenciable externo:
        - "ElementoGeometricoGeorreferenciable"
        - "GeoJSON"
        - "ListadoAbstractoDeCoordenadas"
        - "ListadoDeCoordenadasParaLinea"
        - "ListadoDeCoordenadasParaPoligono"
        - "ListadoDeCoordenadasParaPunto"
        - "TipoDeElemento"

    - DModelo georreferenciable geométrico:
        - "PoligonoDeCuatroLados".
        - "PoligonoDeCuatroLadosTest".

Dentro del paquete "Visual" no encontraremos ninguna clase, sólo veremos imágenes para el fondo de la aplicación y para los censistas, y un archivo ".jar" utilizado para el mapa.

Métodos

Por motivos de facilitar la lectura de este informe, obviamos las clases de tipo "test" ya que asumimos que su descripción no es necesaria.

Dentro del paquete "censo" mencionamos previamente que existía una clase que funcionaba como clase "main" del programa, por lo que asumimos su explicación es innecesaria para el desarrollo del informe.

A continuación veremos la clase "controlador" y sus métodos:

- importarDatos(): Obtiene y guarda internamente un objeto GeoJSON a partir de un archivo de extensión .geojson
- convertirGeoJsonenGrafo(): Convierte los listados de listas de coordenadas para polígonos, líneas y puntos en las estructuras que necesitamos para el grafo:
    - convertirPoligonosDeJsonEnVertices():
      Listas con listas de coordenadas para polígonos, en manzanas que son nodos del grafo, y cada manzana contiene un polígono construido con la lista de coordenadas y un centro obtenido de la lista de coordenadas para puntos.
    - convertirLineasDeJsonEnAristas():
      Listas con listas de coordenadas para lineas, en aristas del grafo que indican la adyacencia entre manzanas.
    - convertirPuntosDeJsonEnCentrosDeNodo():
      Listas con coordenadas para puntos, en centros para las manzanas
- getManzanas(): devuelve todos los vértices del radio censal.
- getManzanasDeRecorrido(): devuelve todos los vértices del recorrido que se realiza para asignar las manzanas a censistas
- elegirNombre(): se encarga de elegir un nombre para el censista de acuerdo a su género.
- hayRecorridoYaCreado(): se encarga de devolver true en caso de que ya exista un recorrido creado entre nodos o false en caso contrario.
- encontrarManzanaALaQuePertenece(): se encarga de recorrer todas las manzanas del radio censal y de verificar a cuál pertenece determinada coordenada.
- obtenerArbolCensal(): se encarga de generar un árbol censal, dónde cada tres manzanas asignamos un censista distinto.
- reinicializar(): se encarga de eliminar el radio censal, el árbol censal y el recorrido realizado por el programa en caso de que se quiera volver al menú una vez cargados dichos datos para poder cargar otro archivo con coordenadas nuevas, o simplemente porque se busca una nueva asignación con el mismo radio censal cargado.

Dentro del paquete "Frontend" hallaremos las siguientes clases:

La clase "BackgroundPane" existe únicamente para dibujar sobre el fondo de los JFrames utilizados una imágen, es decir, para dibujar un fondo de pantalla sobre las pantallas de la aplicación.

Dentro de la clase "frontend" hallamos los siguientes métodos:

- initialize(): se encarga de iniciar la clase llamando al frame inicial de la misma.
- frameInicial(): se encarga de mostrar por pantalla el menú principal de la aplicación.
- frameMapa(): se encarga de mostrar por pantalla el mapa ya con la información del archivo elegido por el usuario cargada y lista para leer.
- frameCensistas(): se encarga de mostrar por pantalla una lista con los perfiles de los censistas utilizados en el "censo", mostrando así su nombre, número de ID y una fotografía..
- fondoDePantalla(): se encarga de dibujar un fondo de pantalla sobre los JFrames de la aplicación.
- volverAlMenu(): se encarga de generar un nuevo controlador, de eliminar el frame del mapa y de generar un nuevo frame del menú principal.
- zoomAMarcadores(): se encarga de llevar al usuario hasta la ubicación de los marcadores del mapa en caso de haberse alejado.
- switcherPane(): se encarga de generar un nuevo JPanel de acuerdo al censista que estemos eligiendo en el menú de "Perfil de los censistas".
- importarRadioCensal(): se encarga de llamar al método "buscarArchivoMapa".
- mostrarRadioCensalImportado(): se encarga de enviarle al controlador el path del archivo con la información, de crear el frame del mapa y de llamar al método "ubicarEnMapa".
- buscarArchivoMapa(): se encarga de mostrar por pantalla al usuario una pantalla donde podrá elegir el archivo a leer para luego ubicar en el mapa distintos marcadores.
- ubicarEnMapa(): se encarga de ubicar en el mapa las coordenadas leidas desde el archivo ".JSON".
- distribuirCensistas(): se encarga de llamar al controlador y de generar los censistas necesarios de acuerdo a las manzanas ingresadas.

En el paquete grafo se encuentran clases ya conocidas, que fueron presentadas en el TP anterior. Se actualizó solamente la clase BFS, que antes se utilizaba para saber si un grafo es conexo. Ahora también ayuda a realizar un recorrido sobre el mismo, guardando el órden en que recorre los nodos:

La primer clase es "ArbolGeneradorMinimo" y sus métodos son:

- ArbolGeneradorMinimo(): Es el constructor de la clase, y debe recibir el grafo original.
- Los siguientes métodos, son internos de la clase , sirven para realizar los pasos del algoritmo de Prim y algunas tareas internas de verificación.
    - confirmarElNuevoVerticeAgregado()
    - quitarAristaDeMenorPesoDelListadoParaProcesar()
    - agregarAristaDeMenorPesoAlArbol()
    - revalidarVerticeActualDelGrafo()
    - agregarElSiguienteVerticeParaElArbolSiEsNecesario()
    - tomarElSiguienteVerticeParaElArbol()
    - tomarElSiguienteVerticeDelGrafo()
    - agregarElPrimerVerticeAlArbol()
    - agregarVerticeActualAlArbol()
    - tomarElPrimerVerticeDelGrafo()
    - estaAristaYaExiste(Arista<T1>)
    - agregarAristasParaVerificar()
    - encontrarLaAristaDeMenorPeso()
    - quitarAristasQueNoRequierenProcesar()

La segunda clase es "Arista" y sus métodos son:

- Arista():
- losExtremosSonElMismoVertice(): Sirve para validar que no existan loops, es decir no puede haber un nodo que tenga una arista consigo mismo.
- La clase presenta una serie de "getters" y "setters" para las variables de la arista en sí.

La tercer clase es "BFS" y sus métodos son:

- BFS(): es el método constructor de la clase.
- esConexo(): se encarga de verificar que un árbol sea conexo o no usando BFS y asegurándose de obtener todos los nodos del árbol original..
- Los siguientes métodos son internos de la clase, sirven para realizar los pasos del recorrido de un grafo y algunas tareas internas de verificación.
    - marcarVecinos(Nodo<T1>)
    - marcarTambienConfirmarActual(int)
    - seRecorrieronTodosLosVertices()
    - hayMasParaProcesar()
    - proximoParaProcesar()
    - completarRecorridoSiEsNecesario()
    - agregarVerticeEnElRecorrido(int)

La cuarta clase es "Distancia" y sus métodos son:

- La clase presenta un constructor y distintos "getters" y "setters" para sus variables.

La quinta clase es "Grafo" y sus métodos son:

- Agregan vértices al grafo:

    - agregarVertice(T1)
    - agregarVertice(Nodo<T1>)
    - Agregan Aristas al grafo:
    - agregarArista(Arista<T1>) | Método privado, se usan por las siguientes dos funciones públicas.
    - agregarArista(Nodo<T1>, Nodo<T1>, float)
    - agregarArista(T1, T1, float)
- Verificaciones que lanzan excepciones cuando se intenta modificar nuestro invariante de representación:
    - verificarQueLosNodosNoSeanElMismo(Arista<T1>)
    - verificarVerticesDeArista(Arista<T1>)
    - verificarExisteArista(Arista<T1>)
    - verificarVertice(Nodo<T1>)
- Eliminar Arista
    - eliminarArista(Arista<T1>)
- Otras consultas y Getters
    - existeArista(Arista<T1>)
    - tamano(): Devuelve la cantidad de Nodos del grafo.
    - vecinos(Nodo<T1>): Devuelve un listado de Distancias que son los vecinos de este Nodo.
    - obtenerTodosLosVertices(): Devuelve el listado de nodos.
- Estos Getters devuelven instancias del Nodo solicitado
    - obtenerVerticeConVecinos(int)
    - obtenerVerticeConVecinos(Nodo<T1>)
    - obtenerVerticeConVecinos(T1)

- Estos Getters devuelven un nuevo nodo con mismo valor que el del grafo, pero sin sus vecinos. Útil para cuando uno necesita construir un nuevo grafo o un árbol a partir de un grafo original.
    - obtenerVerticeSinVecinos(T1): Devuelve un nodo del grafo si tiene el mismo valor T1
    - obtenerVerticeSinVecinos(Nodo<T1>): Devuelve un valor T1 si existe un nodo<T1>
    - obtenerNodoSinVecinos(Nodo<T1>): Devuelve una instancia de Nodo<T1> diferente si existe en el grafo la instancia del Nodo<T1> pasado por parámetro.

La sexta clase es "Nodo" y sus métodos son:

- Getters y Setters sobre los datos del Nodo
    - getInformacion()
    - toString()
    - setInformacion(T1)
- Operaciones sobre la lista de vecinos:
    - agregarVecino(Nodo<T1>, Float)
    - obtenerTodosLosVecinos()
    - obtenerCantidadDeVecinos()
    - obtenerDistancia(int):

    - obtenerDistancia(Nodo<T1>):
    - esVecino(Nodo<T1>)
    - eliminarVecino(int)
    - eliminarVecino(Nodo<T1>)
    - indiceDeVecino(Nodo<T1>)
- Verificaciones para poder agregar o eliminar vecino vecinos:
    - verificarRequisitoSerVecino(Nodo<T1>): Lanza excepción si se quiere eliminar un vecino que no es vecino.
    - verificarRequisitoParaSerVecino(Nodo<T1>): Lanza excepción si se quiere agregar un vecino que ya es vecino.
- Comparación entre nodos. Los nodos son iguales si su información es igual. La comparación no incluye ni nos interesa saber si tiene la misma lista de vecinos.
    - equals(Nodo<T1>):

Dentro del paquete "modelo" hallamos las siguientes clases:

La primer clase es "Censista" y contiene los siguientes métodos:

- Censita(nombre,numero): que funciona como constructor de la clase:
- getNombre(): devuelve el nombre del censista.
- setNombre(): le define un nombre al censista.
- getNumero(): devuelve el número del censista.
- setNumero(): le define un número al censista.

La segunda clase es "Manzana" y presenta los siguientes métodos y propiedades:

- Censista asignado, para indicar a qué censista le asignaron esta manzana.
- Un polígono de cuatro lado, el cual sirve para realizar cálculos (por ej. si una coordenada está dentro o fuera del mismo (es decir, de la manzana).
- Un listado de sus coordenadas
- Un punto llamado centro, con coordenadas que está inscrito en el polígono
- Manzana(aristas): funciona como clase constructora y toma como parámetro un arraylist de coordenadas que funcionan como aristas.
- manzanasDesdeCoordenadas(): se encarga de crear y devolver una nueva manzana con las aristas pasadas por parámetro y de crear un polígono de cuatro lados.
- conversionTipoDeCoordenadas(): convierte un tipo de información (una matriz de tipo float de aristas) y de utilizarlas como coordenadas.
- asignarCensista(): se encarga de asignarle un censista a la manzana.
- crearPoligonoDeCuatroLados(): crea un polígono geográfico de cuatro lados con las coordenadas ya obtenidas.
- getCoordenadasDeAristas(): devuelve las coordenadas de aristas.
- laManzanaContiene(): revisa que el polígono geográfico contenga un elemento de tipo "coordenada".
- getCentro(): devuelve el centro de la manzana.
- setCentro(): setea un centro a la manzana según ciertas coordenadas.
- getCensista(): devuelve el censista asignado a la manzanza.

La clase "RumboCardinal" funciona como un objeto de tipo "ENUM" con los posibles puntos cardinales existentes.

Dentro del paquete "Georreferenciable" hallamos dos carpetas, la primera se llama "externo" y presenta las siguientes clases:

- GeoJSON: representa la estructura de un archivo con extensión homónima. Formado por un Type y una lista de Features. Los features son elementos geométricos que se pueden ubicar en un mapa (polígonos, líneas y puntos). Modelizamos estos tres tipos de elementos como la clase siguiente.

- ElementoGeometricoGeorreferenciable: representa un Feature de GeoJSON. Posee características que son Strings (properties, type, etc). Pero nos interesa en realidad el Map geometry que contiene las coordenadas. Pero dependiendo si el "type" es "polygon", "stringline" o "point", el map puede contener un "listado de listas de lista de número con decimal" (si, no es un error de tipeo, así es un polígono geojson), o un listado de lista de número con decimal (líneas) o un listado de decimales para punto.

- Adaptamos entonces el elemento geométrico georeferenciable en tres clases según qué conforma cada listado, y utilizamos una clase abstracta para que hereden de ella. Cada clase castea el map geometry del feature, y lo convierte en el listado necesario y luego poseen un getter para obtener ese listado, ya no de números decimales, sino de un tipo Coordinate.
    - ListadoAbstractoDeCoordenadas
    - ListadoDeCoordenadasParaLinea
    - ListadoDeCoordenadasParaPoligono
    - ListadoDeCoordenadasParaPunto

Por otro lado, en la carpeta "Geometrico" hallamos la clase "PoligonoDeCuatroLados" que sirve como una clase intermedia entre la clase externa Polygon del paquete JTS. Los polígonos de JTS pueden ser de muchos lados, a nosotros solamente nos interesa formar polígonos de cuatro lados. Contiene métodos para asegurarse de ellos, getters de las coordenadas y una función contiene que asiste en saber si dada una coordenada la misma se encuentra dentro del polígono.

- PoligonoDeCuatroLados()
- convertirArrayListEnArreglo(): transforma un arraylist de aristas en un array de coordenadas.
- ParametroDeCreacionCorrecto(): revisa que la cantidad de aristas enviadas sean 5 y de que se pueda formar un circuito cerrado con las mismas.
- esUnCircuitoCerradoDeCincoAristas(): revisa que el polígono sea un circuito cerrado de cinco aristas comparando la primera contra la última.
- obtenerCoordenadas(): devuelve las coordenadas del polígono.
- contiene(): se encarga de devolver "true" en caso de que un determinado punto exista en un determinado polígono.

Problemas y Soluciones

Problema de la clase(s) "Coordenada"

El primer problema a resolver era la estructura de datos que permitiera representar las manzanas de forma tal que fuera simple representarlas visualmente, y que se pudiera operar con las coordenadas que la conforman. La solución encontrada no fue del todo elegante.

Se tuvieron que utilizar dos tipos de "coordenadas", una que permite desplegar visualmente un polígono en el mapa de "JMapViewer", y otra coordenada de un paquete "JTS" (Java Topology Suite) que nos permitió realizar cálculos sobre si un punto o segmento se encontraba contenido dentro de un polígono (manzana).

Problema de la creación de un radio censal

Otro inconveniente, pero en materia de la administración del tiempo para completar el desafío de este trabajo práctico, fue la creación de una interfaz que permitiera definir las manzanas de los radios censales.

Dicha interfaz debería ser intuitiva, pero encontramos que no solamente alcanzaba con que un usuario elija las manzanas, sino que también se debía conservar la información relativa a la adyacencia entre las mismas.

Además no encontramos documentación de "JMapViewer" ni de "OpenStreetMap" concernientes a obtener la información requerida (donde empieza y termina una manzana, con cuáles es adyacente).

Decidimos entonces que el programa no se encargaría de extraer los datos necesarios para crear un radio censal, sino más bien que pudiera interpretar los radios censales desde un archivo.

Hemos dispuesto tres radios censales en archivos de formato "GeoJSON" que se pueden crear desde "OpenStreetmap". Pueden encontrar la web dónde fueron creados los radios censales aquí: http://u.osmfr.org/m/769269/

Un radio censal entonces requiere de tres colecciones de elementos geométricos que además son ubicables en un mapa (son a su vez, georeferenciales):

- Una colección de polígonos, dónde cada polígono es representado como otra colección de coordenadas. De esta manera se puede extraer la información de una manzana.
- Una colección de líneas, cuyo extremo comienza en el interior de una manzana y finaliza en otra. Así es posible representar la adyacencia entre manzanas, y además las aristas del grafo.
- Una colección de puntos. Dentro de cada manzana, que sirve para ubicar un marcador dentro de la misma, cuando se pueda indicar que esta tiene un censista asignado.

El desafío fue resuelto creando una clase que contenga la misma estructura utilizada por el formato abierto "GeoJSON", que además es de tipado dinámico, lo que se confrontaba con lo fuertemente tipado que es Java. Para lo cual tuvimos que castear en el mismo listado de elementos (features en el código fuente) estas tres colecciones diferentes. Que en sí, no son "listados de polígonos", sino una "lista de listas de coordenadas", por ejemplo.

Finalmente, una vez separadas las listas de coordenadas según el tipo, ya pudimos incorporar en nuestra clase "Manzana" la estructura "polígono de cuatro lados", el censista asignado y un centro, que es más bien punto geográfico de referencia dentro de la manzana. Incorporar cada bloque o cuadra como vértice del grafo, e interpretar cada línea que unía a dos de estas, como una arista entre dos nodos del grafo.

Cuestión del órden de importación del radio censal

Descubrimos que el órden de elección, de cuál sería la primera manzana en asignarle el primer censista y cuál sería la segunda manzana, y así sucesivamente, dependía del órden en que eran importados los datos del archivo GeoJSON. Es decir, que la asignación y el camino a tomar no cambiarían, sino que venía determinado por el órden de aparición de las coordenadas de un archivo estático.

Para eliminar esa jerarquía no deseada, incorporamos un checkbox (habilitado por defecto) que indica "ignorar jerarquía de importación". Que genera un shuffle, una mezcla pseudo random, del listado de coordenadas para formar polígonos (luego manzanas) y también del listado que forma aristas. De manera que en una asignación de censistas, el órden de los nodos en el grafo, y el órden de la lista de vecinos de cada nodo, no sea siempre la misma.