Algoritmos y Estructuras de Datos II / Programación II

Trabajo Práctico Obligatorio

1er Cuatrimestre 2024

Grupo 3: Manuel Núñez, Santiago A. Ferreyra, Bautista Farabello Fontán, Santiago Ribecca y Juan I. Casareski.

Desarrollo del código

Interfaz de Ciudades TDA

```
public interface CiudadesTDA {

    public void Inicializar();

    public void CargarProvincia(String nombreProvincia);
    public DiccionarioSimpleStrTDA Provincias();
    public void CargarCiudad(String nombreCiudad, int idProvincia);
    public DiccionarioSimpleStrTDA CiudadesPorProvincia(int idProvincia);
    public void EliminarCiudades(int idCiudad);

    public ConjuntoTDA CiudadesVecinas(int idCiudad);
    public DiccionarioSimpleStrTDA CiudadesPuente(int idOrigen, int idDestino);
    public ConjuntoTDA CiudadesPredecesoras(int idCiudad);
    public ConjuntoTDA CiudadesExtremo();
    public DiccionarioSimpleTDA CiudadesFuertmenteConectadas();

    public void UnirCiudades(int idOrigen, int idDestino, int distancia);
    public int Distancia(int idOrigen, int idDestino);
}
```

En la interfaz del programa se puede observar las distintas implementaciones abstraídas del TPO. A continuación, se explicarán como fueron implementadas los métodos referidos a este TDA.

Todos los métodos del TDA devolverán Valores directamente, ya que el programa está desarrollado para que estos valores se utilicen en futuras operaciones.

Estrategia General

En este TDA decidimos implementarlo utilizando los distintos TDAs que vimos a lo largo de la cursada.

```
public class Ciudades implements CiudadesTDA {
    int ultimaProvincia;
    int ultimaCiudad;

    GrafoTDA ciudades;
    DiccionarioSimpleStrTDA nombresCiudades;
    DiccionarioSimpleStrTDA nombresProvincias;
    DiccionarioMultipleTDA ciudadesProvincias;

    @Override
    public void Inicializar() {
        ultimaCiudad = 1;
        ultimaProvincia = 1;

        ciudades = new GrafoDinamico();
        ciudades.InicializarGrafo();

        nombresCiudades = new DiccionarioSimpleStr();
        nombresCiudades.InicializarDiccionario();

        nombresProvincias = new DiccionarioSimpleStr();
        nombresProvincias.InicializarDiccionario();

        ciudadesProvincias = new DiccionarioMultipleDinamico();
        ciudadesProvincias.InicializarDiccionario();
    }
```

- GrafoTDA ciudades: como su tipo de dato indica, este grafo es la variable principal del trabajo practico. Al momento de cargar una ciudad en el sistema, llamaremos a las funciones propias del TDA grafo que cargará las ciudades como vértices que serán unidos por aristas con peso que serán las rutas con su respectiva distancia.
- DiccionarioSimpleStrTDA nombresCiudades: se almacenará el nombre de la ciudad junto con un ID (la clave) asignado por el contador “ultimaCiudad” que ira incrementando su valor en 1 por cada ciudad añadida al grafo.
- DiccionarioSimpleStr nombresProvincias: se utiliza la misma lógica que en “nombresCiudades” solo que, aplicado con las provincias, con su propio contador que cambia la id de cada provincia que se agrega.

- DiccionarioMultipleTDA ciudadesProvincias: este diccionario múltiple tendrá como claves las IDs de cada provincia y los valores serán las IDs de las ciudades que pertenecen a su respectiva provincia

En Inicializar() podemos observar cómo los contadores inician en 1 y cada TDA utilizado es creado de forma dinámica.

Cargar Provincia

```
public void CargarProvincia(String nombreProvincia) {
    nombresProvincias.Agregar(ultimaProvincia, nombreProvincia);

    ultimaProvincia++;
}
```

Cargar provincia recibe como parámetro el nombre de la provincia que se quiere agregar, proveniente del Script main, esta será cargada al diccionario simple “nombres provincias” junto con la id correspondiente del contador, luego, este último será aumentado para preparar el próximo ingreso.

Provincias

```
public DiccionarioSimpleStrTDA Provincias() {
        return nombresProvincias;
    }
```

Retorna el diccionario nombresProvincias, que tiene las ID y los nombres de las provincias.

Cargar Ciudad

```
public void CargarCiudad(String nombreCiudad, int idProvincia) {
    ciudades.AgregarVertice(ultimaCiudad);
    nombresCiudades.Agregar(ultimaCiudad, nombreCiudad);
    ciudadesProvincias.Agregar(idProvincia, ultimaCiudad);

    ultimaCiudad++;
}
```

Cuando cargamos una ciudad al Grafo ciudades, primero es cargada en el aire, luego" Unir ciudades" se encargará de unir los vértices solicitados con su respectiva distancia.

Luego, como al cargar una provincia, la ciudad será agregada a un diccionario simple con su ID. Este se cargará al diccionario múltiple, como valor asociado al ID de la provincia al que pertenece la ciudad.

Por último, el contador se incrementará en uno para dar paso a la carga de una nueva ciudad.

Ciudades Por Provincia

```
public DiccionarioSimpleStrTDA CiudadesPorProvincia(int idProvincia) {

        DiccionarioSimpleStrTDA ciudadesPorProvincia = new DiccionarioSimpleStr();
        ciudadesPorProvincia.InicializarDiccionario();
        ConjuntoTDA idciudades = ciudadesProvincias.Recuperar(idProvincia);

        while(!idciudades.ConjuntoVacio()){
            int aux=idciudades.Elegir();
            ciudadesPorProvincia.Agregar(aux, nombresCiudades.Recuperar(aux));
            idciudades.Sacar(aux);
        }

        return ciudadesPorProvincia;
    }
```

Este método recibe como parámetro una ID de una provincia y devuelve un diccionario simple de las ID y el nombre de las ciudades pertenecientes a esa provincia.

Primero por medio del método "Recuperar" del diccionario múltiple, obtenemos un conjunto con las claves de las ciudades de esa provincia, luego lo utilizamos para recuperar los nombres de estas provincias del diccionario simple "nombres ciudades" y juntar las ID y los nombre en el nuevo diccionario simple que devolveremos en la función.

Eliminar Ciudades

```
public void EliminarCiudades(int idCiudad) {
        ConjuntoTDA aux = new ConjuntoDinamico();
        aux = ciudades.Vertices();

        ciudades.EliminarVertice(idCiudad);
        nombresCiudades.Eliminar(idCiudad);

        while (!aux.ConjuntoVacio()) {
            int provincia = aux.Elegir();
            aux.Sacar(provincia);

            if (ciudadesProvincias.Claves().Pertenece(provincia)) {
                ciudadesProvincias.EliminarValor(provincia, idCiudad);
            }
        }
    }
```

Como indica su nombre, en este metodo se borrara todo el rastro de la ciudad, y sus conecciones dentro del grafo.

Ciudades Vecinas

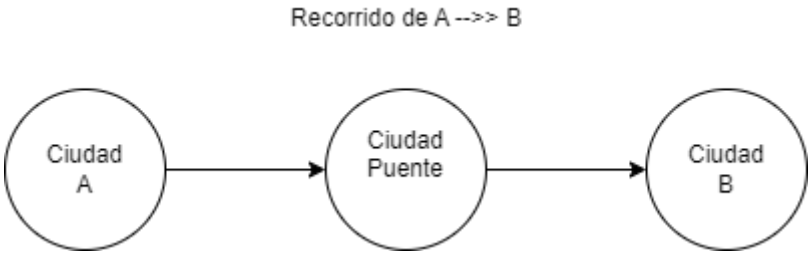
```
public ConjuntoTDA CiudadesVecinas(int idCiudad) {
        ConjuntoTDA ciudadesVecinas = new ConjuntoDinamico();
        ConjuntoTDA ciudadesComparar = new ConjuntoDinamico();

        ciudadesComparar = ciudades.Vertices();
        ciudadesComparar.Sacar(idCiudad);

        while (!ciudadesComparar.ConjuntoVacio()) {
            int ciudadVecina = ciudadesComparar.Elegir();

            if (ciudades.ExisteArista(idCiudad, ciudadVecina) != ciudades.ExisteArista(ciudadVecina, idCiudad)) {
                ciudadesVecinas.Agregar(ciudadVecina);
            }

            ciudadesComparar.Sacar(ciudadVecina);
        }

        return ciudadesVecinas;
    }
```

Recibiendo como parámetro la ID de la ciudad, el método ‘Ciudades Vecinas’ recorre un conjunto, hasta que este vacío, con todas las ciudades y las compara con la ciudad elegida hasta encontrar una ruta entre ambas. Si la comparación halla a la ciudad vecina entonces la agregara al conjunto ‘Ciudades vecinas’, que al final se retorna.

Ciudades Puente

```
public DiccionarioSimpleTDA CiudadesPuente(int idOrigen, int idDestino) {
        DiccionarioSimpleDinamico ciudadesPuente = new DiccionarioSimpleDinamico();
        ConjuntoTDA ciudadesComparar = ciudades.Vertices();

        ciudadesPuente.InicializarDiccionario();

        while (!ciudadesComparar.ConjuntoVacio()) {
            int ciudadPuente = ciudadesComparar.Elegir();
            ciudadesComparar.Sacar(ciudadPuente);

            if (ciudades.ExisteArista(idOrigen, ciudadPuente) && ciudades.ExisteArista(ciudadPuente,
idDestino)) {
                ciudadesPuente.Agregar(ciudadPuente, ciudades.PesoArista(idOrigen, ciudadPuente) +
ciudades.PesoArista(ciudadPuente, idDestino));
            }
        }

        return ciudadesPuente;
    } = new DiccionarioSimpleStr();
        nombresProvincias = new DiccionarioSimpleStr();
        ciudadesProvincias = new DiccionarioMultipleDinamico();
    }
```

El método CiudadesPuente está diseñado para encontrar todas las ciudades que actúan como "puentes" entre dos ciudades dadas (A y B). Una ciudad puente es aquella que tiene una ruta directa desde A a ella y desde ella a B. Además, el método calcula la distancia total que debe recorrer una persona para viajar desde A hasta B pasando por cada una de estas ciudades puente.



Por cada una de las ciudades del grafo, verifica si existe una ruta directa desde la ciudad de origen hasta la ciudad y desde esta ultima hasta la ciudad destino. Si encuentra una ciudad puente, calcula la distancia total pasando por esta ciudad y se almacena en un diccionario.

Al final, se retorna este diccionario con todas las ciudades puente y sus respectivas distancias. En caso de que se pase por parámetro, dos ciudades contiguas, se devolvera un diccionario vacio.

Ciudades Predecesoras

```
public ConjuntoTDA CiudadesPredecesoras(int idCiudad) {
        ConjuntoTDA ciudadesPredecesoras = new ConjuntoDinamico();
        ConjuntoTDA ciudadesComparar = ciudades.Vertices();

        ciudadesPredecesoras.InicializarConjunto();
        ciudadesComparar.Sacar(idCiudad);

        while (!ciudadesComparar.ConjuntoVacio()) {
            int ciudad = ciudadesComparar.Elegir();

            if (ciudades.ExisteArista(ciudad, idCiudad)) {
                ciudadesPredecesoras.Agregar(ciudad);
            }

            ciudadesComparar.Sacar(ciudad);
        }

        return ciudadesPredecesoras;
    }
```

En este método, se guarda en un conjunto todas las ciudades que tengan rutas que finalicen en la ciudad pasada como parámetro.

Mediante el uso de un ciclo repetitivo, que termina cuando el conjunto este vacío, buscamos en todas las ciudades de ‘ciudadesComparar’ si existe una arista entre la actual y la que recibe como parámetro al principio. Si se encuentra un resultado se agrega al conjunto inicializado ‘ciudadesPredecesoras’ que se retorna al finalizar el método.

Ciudades Extremo

```
public ConjuntoTDA CiudadesExtremo() {
    ConjuntoTDA conjuntoCiudades = new ConjuntoDinamico();
    ConjuntoTDA conjuntoCiudadesComparar = new ConjuntoDinamico();
    ConjuntoTDA ciudadesExtremo = new ConjuntoDinamico();


    conjuntoCiudades = ciudades.Vertices();

    while (!conjuntoCiudades.ConjuntoVacio()) {
        boolean esExtremo = true;
        int ciudadElegida = conjuntoCiudades.Elegir();
        conjuntoCiudades.Sacar(ciudadElegida);

        conjuntoCiudadesComparar = ciudades.Vertices();
        conjuntoCiudadesComparar.Sacar(ciudadElegida);

        while (!conjuntoCiudadesComparar.ConjuntoVacio() && esExtremo) {
            int ciudadComparar = conjuntoCiudadesComparar.Elegir();
            conjuntoCiudadesComparar.Sacar(ciudadComparar);

            if (ciudades.ExisteArista(ciudadElegida, ciudadComparar)) {
                esExtremo = false;
            }
        }

        if (esExtremo) {
            ciudadesExtremo.Agregar(ciudadElegida);
        }
    }

    return ciudadesExtremo;

}
```

Este método será la encargada de identificar y almacena en un conjunto las ciudades que únicamente tienen un camino que llegan a ellas.

La estrategia utilizada consiste en seleccionar una ciudad y luego comparar las relaciones con todas las demás ciudades, si existe un camino que salga de la primera ciudad hacia otra de las ciudades elegidas, si este camino existe, bajará una bandera, y elegirá otra ciudad para comparar con las demás.

En caso de que no exista ningún camino que salga de la ciudad, y solo caminos que llegan a la misma, la agregará al conjunto de ciudades extremo

Ciudades Fuertemente conectadas

```java
public DiccionarioMultipleTDA CiudadesFuertmenteConectadas() {
        ConjuntoTDA conjuntoCiudades = new ConjuntoDinamico();
        ConjuntoTDA conjuntoCiudadesComparar = new ConjuntoDinamico();
        DiccionarioMultipleTDA ciudadesFuertementeConectadas = new DiccionarioMultipleDinamico();

        conjuntoCiudades = ciudades.Vertices();

        while (!conjuntoCiudades.ConjuntoVacio()) {
            int ciudadElegida = conjuntoCiudades.Elegir();
            conjuntoCiudades.Sacar(ciudadElegida);

            conjuntoCiudadesComparar = ciudades.Vertices();
            conjuntoCiudadesComparar.Sacar(ciudadElegida);

            while (!conjuntoCiudadesComparar.ConjuntoVacio()) {
                int ciudadComparar = conjuntoCiudadesComparar.Elegir();
                conjuntoCiudadesComparar.Sacar(ciudadComparar);

                if (ciudades.ExisteArista(ciudadElegida, ciudadComparar) && ciudades.ExisteArista(ciudadComparar, ciudadElegida)) {
                    ciudadesFuertementeConectadas.Agregar(ciudadElegida, ciudadComparar);
                }
            }
        }

        return ciudadesFuertementeConectadas;
    }
```

Para este método iteramos sobre dos conjuntos dinámicos que almacenan a las ciudades ingresadas. En el primer loop elegimos una ciudad del primer conjunto y buscamos la existencia de rutas de ida y de vuelta con algunas de las ciudades guardadas en el otro set.

En caso de haber encontrado ‘ciudades fuertemente conectadas’, estas se agregarán a un diccionario múltiple dinámico (‘ciudadesFuertementeConectadas’) que se retornara al final.

Unir Ciudades

```java
public void UnirCiudades(int idOrigen, int idDestino, int distancia) {
        ciudades.AgregarArista(idOrigen, idDestino, distancia);
    }
```

En ese método simplemente se llama a la implementación de Grafo para agregar la arista entre las ciudades

Distancia

Programa Principal (TPO.JAVA)

En el Programa principal se hace uso de los distintos métodos de la interfaz, y se imprimen por pantalla los resultados de cada uno.

Problemáticas

Al comienzo, tuvimos dudas sobre qué parámetros debía recibir el método Cargar. No sabíamos si al cargar una ciudad debíamos incluir también las aristas con sus datos. Decidimos que primero era conveniente crear los nodos sin aristas, y luego, a través de otro método, unirlos, recibiendo como parámetros el peso y los nodos a conectar.

Otro problema fue la decisión de si los métodos del TDA debían imprimir los datos solicitados o simplemente devolverlos. Concluimos que es mejor que los métodos devuelvan valores para poder utilizar estos resultados en otros métodos. Además, no es adecuado que una función imprima resultados, ya que esto implicaría que la función cumple dos roles: imprimir y procesar datos. Esto contradice el principio de responsabilidad única.

Decidimos utilizar una tabla para asociar un número a un nombre de ciudad, como si fuera una identificación. Sin embargo, surgió un problema con el método CiudadesFuertementeConectadas, que necesitaba devolver pares de ciudades. Para solucionar esto, creamos un nuevo diccionario simple donde los valores son los nombres de las ciudades en formato String y las claves son sus IDs en enteros.

El trabajo práctico solicitaba muy pocas funciones relacionadas con las provincias; no se pedía cargarlas ni eliminarlas. Decidimos que las provincias se cargarían pero no se eliminarían, porque el programa necesita las provincias para cargar las ciudades con las cuales vamos a operar, aunque no se necesiten otras operaciones sobre ellas.

Al implementar la función de distancia, debatimos sobre qué camino debíamos retornar: el más corto, todos los caminos, o simplemente la distancia. Optamos por retornar el camino más corto implementando el algoritmo de Dijkstra, ya que esto proporciona una solución eficiente y clara para el problema planteado.

Conclusión

El desarrollo del código debatió sobre la funcionalidad de la interfaz, de los tipos de datos abstractos y su correcta aplicación en un código.

Los TDAs son un conjunto de datos y los métodos u operaciones que se pueden realizar sobre ellos definidas en una interfaz, sin que se entre en detalle de como lo hacen. Los TDAs proporcionan una manera de definir la estructura y comportamiento de los datos a un nivel abstracto, es decir, definen qué operaciones se pueden realizar y qué comportamientos se esperan, sin detallar cómo se llevan a cabo.

En la implementación es donde se detalla cómo funcionan las operaciones referentes al TDA definido en la Interfaz previamente.

Al comienzo del desarrollo, el grupo se dio cuenta de que lo solicitado en la consigna no estaba del todo desarrollado. Había funciones necesarias que faltaban; por ejemplo, todo lo que respecta a las provincias era casi inexistente. Únicamente se solicitaba que se listaran, pero no que el usuario pudiera agregarlas ni quitarlas.

Nuestro grupo optó por no incluir estas funcionalidades y se limitó a hacer lo que estaba especificado en el trabajo práctico. En un proyecto, el cliente tiene que especificar claramente lo que quiere que el programador haga. De no ser así, el programador no tiene por qué ponerse a adivinar lo que busca su empleador. Si implementa algo que cree necesario y esto está mal, la culpa será de él. Las malas especificaciones de funcionamiento son errores del cliente.