Algoritmos y complejidad

Actividad 15

Integrantes:

* Kloster, Matías Nicolas. LU 110873.
* Baschiera, Luciano Manuel. LU 111847.

Profesor:

* Fillottrani, Pablo R.

# Análisis teórico Espacio-Temporal

## Determinar si un grafo no-dirigido y pesado es conexo:

### Implementando un recorrido por niveles (BFS):

Para un grafo . , .

La implementación del algoritmo se encuentra en la clase Conexo, en la carpeta src, bajo el nombre de “conexoBFS”.

Nuestra implementación consiste en realizar un recorrido por niveles en el grafo, y luego recorrer la lista de nodos “padre” para determinar si hay más de un nodo que tenga -1 como padre, lo que indica que este grafo no es conexo.

El tiempo de ejecución de conexoBFS será:

* Θ (n) para setear todos los nodos a color blanco, y poner -1 como padre de cada nodo.
* Θ (1) para crear una nueva cola enlazada.
* Θ (n\*
* (1 para setear el color del nodo actual a gris.
* + 1 para encolar el nodo en Q.
* + n) para VisitarBFS: Para cada nodo en la cola, pregunta por los arcos adyacentes, para con los cuales obtiene el nodo en la otra punta, si es un nodo blanco, le cambia el color y el padre, y lo mete en la cola (Todo Θ(1)). Se puede probar por inducción que cada nodo es metido en la cola una sola vez, por tanto ejecuta n dequeues en total.

Entonces:

El espacio de ejecución de conexoBFS será:

* Θ(3\*n) para los arreglos de color, padre y la cola.
* Θ(n2) para los arreglos de adyacencia de los n nodos (Puede considerarse n si asumimos que reutilizamos la misma lista para cada nodo)

Entonces: o si tenemos en cuenta la consideración anterior.

### Mediante el uso de la estructura disjoint-set:

Para un grafo . , .

La implementación del algoritmo se encuentra en la clase Conexo, en la carpeta src, bajo el nombre de “conexoDS”.

Nuestra implementación consiste en crear n conjuntos, uno para cada nodo, y luego para cada arco del grafo, unir los conjuntos que contienen a los nodos del arco, hasta que haya un solo conjunto disjunto.

El tiempo de ejecución de conexoDS será analizado en función de la totalidad de las operaciones de la estructura de datos Disjoint Set:

Dado que estamos usando la estructura de datos implementada con las 2 heurísticas que se proponen en el libro Cormen (union by rank & path compression), podemos decir el tiempo total de las m operaciones será del

Dado que es una función de crecimiento suave, ya que representa el logaritmo iterado, podemos decir a entradas de gran tamaño esta función está acotada por 4,

ie  **->**

El análisis de la totalidad de las operaciones se debe a que estamos implementado esta estructura de datos con 2 Heuristicas, las cuales proveen un mecanismo que incrementa la performance en las sucesivas llamadas de algunas operaciones como por ejemplo find(x).

Para este método, el cual devuelve el elemento identificador del conjunto en el cual esta el elemento pasado por paramentro(x), se uso la heurística propuesta en el libron Cormen path compression, la cual funciona de la siguiente manera: Utilizando recursión busca al elemento identificador y va seteando como padre al elemento identificador del conjunto en los nodos que fueron visitados hasta llegar a las raíz(padre), esto implica que la primera llamada a find será en el peor caso del orden lineal ya que en el peor caso tendríamos a todos los nodos en una única foresta y llamamos al método con el nodo hoja. Pero luego de esta 1er llamada, si llamamos devuelta al método, sera del orden constante, ya que quedaron todos los nodos enlazados correctamente con su respectivo elemento identificador.

Para la unión, sin embargo, utilizamos otra heurística tambien presentada en Cormen, llamada Union by rank, la cual consiste en unir dos conjuntos, es decir unión(x,y) si y solo si el conjunto de la izquierda(y) tiene la misma(o menor) cantidad de elementos que el conjunto de la derecha(x), esto garantiza una cota superior para la cantidad de veces que se actualiza el puntero de un elemento en el conjunto que pertenece x , el cual es

Donde n representa la cardinalidad del conjunto resultado de la union

Es por esto que el análisis de tiempo de ejecución se esta estructura se hace sobre la totalidad de las operaciones y no se concentra en analizar una operación aislada

## Construir el arbol de cubrimiento minimal:

### Implementando un conjunto de arcos ordenados por su peso:

Nuestra implementación consiste en ordenar la lista de arcos, crear una lista para los arcos del árbol de cubrimiento minimal t, crear n conjuntos disjuntos, e ir eliminando el primer elemento de la lista de arcos para luego comprobar si los nodos que están a cada punta del arco pertenecen al mismo conjunto, caso contrario, los unimos, y metemos al arco en t. Este ciclo corta cuando , que es cuando obtuvimos nuestro resultado.

#### (Con heurísticas)

La implementación del algoritmo se encuentra en la clase Kruskal, en la carpeta src, bajo el nombre de “arcosOrdenadosCH”.

El tiempo de ejecución de arcosOrdenadosCH será:

* para crear el heap y ordenar la lista, ya que utilizamos heapSort para esto, que consiste en crear un heap con los elementos y extraerlos.
* donde n-1 corresponde a las n-1 uniones de conjuntos, 2\*a corresponde a los 2 finds que hacemos por cada arco y corresponde al tiempo de ejecución de los métodos union y find con heurísticas, como fue expuesto anteriormente.

Entonces: .

El espacio de ejecución de arcosOrdenadosCH será:

* para la lista de arcos ordenados.
* para la lista de arcos del árbol de cubrimiento minimal.
* para la estructura de conjuntos disjuntos

Entonces: .

#### (Sin heurísticas)

La implementación del algoritmo se encuentra en la clase Kruskal, en la carpeta src, bajo el nombre de “arcosOrdenadosSH”.

El tiempo de ejecución de arcosOrdenadosSH será:

* para inicializar la estructura de conjuntos disjuntos, crear el heap y ordenar la lista, con heapSort.
* donde n-1 corresponde a las n-1 uniones de conjuntos, 2\*a corresponde a los 2 finds que hacemos por cada arco y n corresponde al tiempo de ejecución de los métodos unión y find que fueron implementados sin heurísticas, como fue expuesto anteriormente.

Entonces: .

El espacio de ejecución de arcosOrdenadosSH será:

* para la lista de arcos ordenados y el heap usado en el ordenamiento.
* para la lista de arcos del árbol de cubrimiento minimal.
* para la estructura de conjuntos disjuntos

Entonces: .

### Implementado con un heap invertido (min-heap):

Nuestra implementación consiste en insertar todos los arcos del grafo en un min-heap, crear una lista para los arcos del árbol de cubrimiento minimal t, crear n conjuntos disjuntos, e ir eliminando el primer elemento del min-heap para luego comprobar si los nodos que están a cada punta del arco pertenecen al mismo conjunto, caso contrario, los unimos, y metemos al arco en t. Este ciclo corta cuando , que es cuando obtuvimos nuestro resultado.

#### (Con heurísticas)

La implementación del algoritmo se encuentra en la clase Kruskal, en la carpeta src, bajo el nombre de “conHeapCH”.

El tiempo de ejecución de conHeapCH será:

* para inicializar la estructura de conjuntos disjuntos y crear el heap.
* donde corresponde a remover todos los arcos del heap, a las uniones, y a los 2 finds que se realizan por cada arco, mientras que corresponde al tiempo de ejecución de los métodos union y find con heurísticas.

Entonces: .

El espacio de ejecución de conHeapCH será:

* para el heap de arcos.
* para la lista de arcos del árbol de cubrimiento minimal.
* para la estructura de conjuntos disjuntos

Entonces: .

#### (Sin heurísticas)

La implementación del algoritmo se encuentra en la clase Kruskal, en la carpeta src, bajo el nombre de “conHeapSH”.

El tiempo de ejecución de conHeapSH será:

* para inicializar la estructura de conjuntos disjuntos y crear el heap.
* donde corresponde a remover todos los arcos del heap, a las uniones, y a los 2 finds que se realizan por cada arco, mientras que corresponde al tiempo de ejecución de los métodos union y find sin heurísticas.

Entonces: .

El espacio de ejecución de conHeapSH será:

* para el heap de arcos.
* para la lista de arcos del árbol de cubrimiento minimal.
* para la estructura de conjuntos disjuntos

Entonces: .

# Tablas de resultados empíricos:

## Prueba de conectitud de grafos no dirigidos y pesados:

|  |  |  |  |
| --- | --- | --- | --- |
| Grafo | | BFS | Disjoint Set |
| Nodos | Arcos | ------------------------- | --------------------------- |
| 5 | 4 | 660034.0ns | 1.0802928E7ns |
| 50 | 49 | 3101214.0ns | 6138476.0ns |
| 150 | 150 | 1671008.0ns | 1.0998728E7ns |
| 333 | 54326 | 594110.0ns | 515553.0ns |
| 456 | 100000 | 400679.0ns | 232907.0ns |
| 500 | 124750 | 7106027.0ns | 2533553.0ns |

## Construcción de árboles de cubrimiento minimales en grafos ND, pesados y conexos:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Grafo | | Ordenado | | Con Heap | |
| Nodos | Arcos | C/ heurísticas | S/ heurísticas | C/ heurísticas | S/ heurísticas |
| 10 | 9 | 65900ns | 84900ns | 922000ns | 4111400ns |
| 100 | 99 | 524100ns | 472900ns | 420500ns | 503500ns |
| 200 | 200 | 1305300ns | 1461900ns | 3987200ns | 4105300ns |
| 366 | 54236 | 7907900ns | 11267400ns | 4585100ns | 11150100ns |
| 444 | 10660 | 1949800ns | 3971200ns | 1504200ns | 3134100ns |
| 500 | 65000 | 6895400ns | 9776500ns | 6490100ns | 7383300ns |

Nota: Estos datos fueron tomados desde la consola de la JVM, fueron obtenidos utilizando el método System.nanoTime() el cual devuelve el valor más preciso del timer del sistema en nanosegundos. La metodología fue utilizarlo antes de la ejecución de las verificaciones de conectitud, almacenar ese valor y luego volver a llamar al método una vez que la verificación haya terminado. Así, restando los valores almacenados pudimos obtener una aproximación del tiempo que demoró la operación en cuestión.

# Listado de los códigos implementados:

#### (En Java)

**package TDACola**;

public interface Queue<E> {

//Insertar un elemento en la cola

public void enqueue(E e);

//Elimina el 1er elemento de la cola y lo retorna

public E dequeue()throws EmptyQueueException;

//retorna el 1er elemento de la cola

public E front() throws EmptyQueueException;

//TRUE si la cola esta vacia, FALSE en caso contrario

public boolean isEmpty();

//Retorna la cantidad de elementos de la cola

public int size();

}

public class ColaEnlazada<E> implements Queue<E> {

protected Nodo<E> head,tail; // cabeza y talon

protected int tamaño;

public ColaEnlazada(){

head=null;

tail=null;

tamaño=0;

}

public void enqueue(E e){

Nodo<E> aux=new Nodo<E>();

aux.setElemento(e); // inicializo el nodo

aux.setSiguiente(null);// lo apunto a null

if (this.isEmpty()){ // Si la cola esta vacia, al elemento lo hago 'cabeza'

head=aux;

}

else{ // y si no, al ultimo elemento que esta en la cola lo hago apuntar al nuevo que acaba de entrar (los enlazo)

tail.setSiguiente(aux);

}

tail=aux; // siempre que inserto un elemento va a ser el 'talon' de la cola

tamaño++;

}

public E dequeue()throws EmptyQueueException{

if(this.isEmpty()){

throw new EmptyQueueException("Cola vacia");

}

else{

E aux= head.getElemento();

head=head.getSiguiente();

tamaño--;

if(this.isEmpty()){

tail=null; //////// me tengo que asegurar que si saco un elemento y justo ese era el ultimo a tail le tengo que asignar null, sino me queda ligado

}

return aux;

}

}

public E front()throws EmptyQueueException{

if(this.isEmpty()){

throw new EmptyQueueException("Cola vacia");

}

else{

return head.getElemento();

}

}

public boolean isEmpty(){

return tamaño==0;

}

public int size(){

return tamaño;

}

}

public class Nodo<E> {

private E elemento;

private Nodo<E> siguiente;

public Nodo() {

this(null, null);

}

public Nodo(E e) {

this(e,null);

}

public Nodo( E e, Nodo<E> sig ) {

elemento=e;

siguiente=sig;

}

public E getElemento(){

return elemento;

}

public void setElemento(E elemento){

this.elemento=elemento;

}

public Nodo<E> getSiguiente(){

return siguiente;

}

public void setSiguiente(Nodo<E>siguiente){

this.siguiente=siguiente;

}

}

public class EmptyQueueException extends Exception{

public EmptyQueueException(String msg){

super(msg);

}

}

**package TDAColaPrioridad;**

import business.Pesado;

public class Heap{

private Pesado[] ar;

private int size;

public Heap(int cant){

this.ar=new Pesado[cant];

this.size=0;

}

public int size() {

return size;

}

public boolean isEmpty() {

return this.size==0;

}

public Pesado min() throws EmptyPriorityQueueException {

if(isEmpty())

throw new EmptyPriorityQueueException("El heap está vacío");

else

return ar[1];

}

public void insert(Pesado p) {

ar[++size] = p;

int i = size;

boolean sigo = true;

while (i > 1 && sigo) {

Pesado elemActual = ar[i];

Pesado elemPadre = ar[(i / 2)];

if (elemActual.getPeso() < elemPadre.getPeso()) {

ar[i] = ar[i / 2];

ar[i / 2] = elemActual;

i /= 2;

} else {

sigo = false;

}

}

}

public Pesado removeMin() throws Exception {

Pesado arco=min();

if(size==1){

ar[size--]=null;

return arco;

}

else{

ar[1]=ar[size]; ar[size--]=null;

int i=1;

heapify(i);

}

return arco;

}

public void heapify(int i){

int hIzq=2\*i, hDer=2\*i+1;

int menor;

if ((hIzq <= size) && (ar[hIzq].getPeso() < ar[i].getPeso()))

menor=hIzq;

else menor=i;

if ((hDer <= size) && (ar[hDer].getPeso() < menor))

menor = hDer;

if (menor!=i){

Pesado aux= ar[menor];

ar[menor]=ar[i];

ar[i]=aux;

heapify(menor);

}

}

}

public class EmptyPriorityQueueException extends Exception {

/\*\*

\* Crea una instacia de la excepcion, la cual ocurre cuando se quiere operar sobre una Cola con prioridades vacia.

\* @param msg Mensaje con informacion del error

\*/

public EmptyPriorityQueueException(String msg){

super(msg);

}

}

public class InvalidKeyException extends Exception {

/\*\*

\* Crea una instacia de la excepcion, la cual ocurre cuando quiero operar sobre un mapeo con una llave invalida.

\* @param msg Mensaje con informacion del error

\*/

public InvalidKeyException(String msg){

super(msg);

}

}

**package TDADisjointSet;**

public class NodoDisjointSH {

private int elemento;

private NodoDisjointSH padre;

public NodoDisjointSH(int elemento, NodoDisjointSH padre, int rank){

this.elemento = elemento;

this.padre = padre;

}

public void setPadre(NodoDisjointSH padre){

this.padre = padre;

}

public NodoDisjointSH getPadre(){

return padre;

}

public int getElemento(){

return elemento;

}

}

public class NodoDisjoint {

private int elemento;

private NodoDisjoint padre;

private int rank;

public NodoDisjoint(int elemento, NodoDisjoint padre, int rank){

this.elemento = elemento;

this.padre = padre;

this.rank = rank;

}

public void setPadre(NodoDisjoint padre){

this.padre = padre;

}

public NodoDisjoint getPadre(){

return padre;

}

public int getElemento(){

return elemento;

}

public int getRank() {

return rank;

}

public void setRank(){

rank++;

}

}

import Exceptions.InvalidPositionException;

import TDALista.DoubleLinkedList;

import TDALista.NodoD;

public class DisjointSetSH {

DoubleLinkedList<NodoDisjointSH> parentList;

NodoD<NodoDisjointSH>[] nodos;

public DisjointSetSH(int n) {

//Crea una lista de nodos padre (cantidad de subconjuntos)

parentList = new DoubleLinkedList<>();

//Crea un arreglo de nodos del tamaño pasado por parámetro

nodos = new NodoD[n];

}

public void initiate(int[] nodos){

//Crea un conjunto para cada nodo en el arreglo de nodos

for(int i= 0; i < nodos.length; i++){

makeSet(nodos[i]);

}

}

//Crea conjuntos con un nodo.

public void makeSet(int n) {

//Crea un nodo con elemento n

NodoDisjointSH nodo = new NodoDisjointSH(n,null,0);

//Se asigna a sí mismo como padre

nodo.setPadre(nodo);

//agrega al nodo a la cabeza de la lista de padres

nodos[n] = parentList.addFirst(nodo);

}

//Retorna el nodo que representa al conjunto en el que se encuentra el nodo pasado por parámetro

public NodoDisjointSH find(int x) {

//Si el nodo no es su propio padre, significa que no es el representante del conjunto.

if (nodos[x].element().getPadre() != nodos[x].element()) {

//Entonces llamamos recursivamente a find con el padre del nodo x

return find(nodos[x].element().getPadre().getElemento());

}

//Caso contrario, estamos con el nodo representante del conjunto.

return nodos[x].element().getPadre();

}

//Une a los conjuntos que contienen a 'x' y a 'y'

public void union(int x, int y) throws InvalidPositionException {

//Busca a los conjuntos de x e y

NodoDisjointSH xRoot = find(x), yRoot = find(y);

//Si find no retornó el mismo nodo, significa que no pertenecen al mismo conjunto,

if (xRoot.getElemento() != yRoot.getElemento()) {

//Seteo a la raíz del conjunto de x como padre de la raíz del conjunto de y.

yRoot.setPadre(xRoot);

//Por lo tanto y ya no es un nodo padre de un conjunto, lo remuevo.

parentList.remove(nodos[y]);

}

}

//Retorna la cantidad de conjuntos en la estructura

public int parentSize(){

return parentList.size();

}

}

import Exceptions.InvalidPositionException;

import TDALista.DoubleLinkedList;

import TDALista.NodoD;

public class DisjointSetCH {

DoubleLinkedList<NodoDisjoint> parentList;

NodoD<NodoDisjoint>[] nodos;

public DisjointSetCH(int n) {

//Crea una lista de nodos padre (cantidad de subconjuntos)

parentList= new DoubleLinkedList<>();

//Crea una lista de nodos del tamaño del entero pasado por parámetro

nodos = new NodoD[n];

}

public void initiate(int[] nodos){

//Crea un conjunto para cada nodo en el arreglo de nodos

for(int i= 0; i < nodos.length; i++){

makeSet(nodos[i]);

}

}

//Crea conjuntos con un nodo

public void makeSet(int n) {

//Crea un nodo con elemento n

NodoDisjoint nodo = new NodoDisjoint(n,null,0);

//Se asigna como padre de sí mismo

nodo.setPadre(nodo);

//Agrega al nodo a la cabeza del arreglo de padres

nodos[n] = parentList.addFirst(nodo);

}

//Retorna el nodo que representa al conjunto en el que se encuentra el nodo pasado por parámetro

public NodoDisjoint find(int x) {

//Si el nodo no es su propio padre, significa que no es el representante del conjunto.

if (nodos[x].element().getPadre() != nodos[x].element()) {

//Entonces llamamos recursivamente a find con el padre del nodo x

nodos[x].element().setPadre(find(nodos[x].element().getPadre().getElemento()));

}

//Caso contrario, estamos con el nodo representante del conjunto.

return nodos[x].element().getPadre();

}

//Une a los conjuntos que contienen a 'x' y a 'y'

public void union(int x, int y) {

//Busca a los conjuntos de x e y

NodoDisjoint xRoot = find(x), yRoot = find(y);

//Si find no retornó el mismo nodo, significa que no pertenecen al mismo conjunto,

if (xRoot.getElemento() != yRoot.getElemento()) {

//Si el rango del conjunto que contiene a x es menor que el del conjunto que contiene a y

if (xRoot.getRank() < yRoot.getRank()) {

//Entonces ponemos al conjunto de x como hijo de la raíz del conjunto de y

xRoot.setPadre(yRoot);

try {

//Por lo tanto x ya no es un nodo padre de un conjunto, lo remuevo.

parentList.remove(nodos[x]);

} catch (InvalidPositionException e) {

e.printStackTrace();

}

}

//Caso contrario, el rango del conjunto de x puede ser mayor que el que contiene a y

else if (yRoot.getRank() < xRoot.getRank()) {

//Entonces ponemos al conjunto de y como hijo de la raíz del conjunto de x

yRoot.setPadre(xRoot);

try {

//Por lo tanto y ya no es un nodo padre de un conjunto, lo remuevo.

parentList.remove(nodos[y]);

} catch (InvalidPositionException e) {

e.printStackTrace();

}

}

//Si llegamos hasta aquí, el rango de ambos conjuntos es igual

else

{

//Ponemos al conjunto de x como padre del conjunto de y (es lo mismo hacer la inversa)

yRoot.setPadre(xRoot);

try {

parentList.remove(nodos[y]);

} catch (InvalidPositionException e) {

e.printStackTrace();

}

//Y recalculamos el rango del conjunto de x

xRoot.setRank();

}

}

}

//Retorna la cantidad de conjuntos en la estructura

public int parentSize(){

return parentList.size();

}

}

**package TDALista;**

public interface Position<E> {

public E element();

}

public class NodoD<E> implements Position<E> {

protected E elemento;

protected NodoD<E> siguiente,anterior;

public NodoD(E e,NodoD<E> sig,NodoD<E> ant){

elemento=e;

this.siguiente=sig;

this.anterior=ant;

}

public NodoD(NodoD<E> sig,NodoD<E> ant){

siguiente=sig;

anterior=ant;

}

public void setElemento(E e){

this.elemento=e;

}

public void setSiguiente(NodoD<E> sig){

siguiente=sig;

}

public void setAnterior(NodoD<E> ant){

anterior=ant;

}

public E element(){

return elemento;

}

public NodoD<E> getSiguiente(){

return siguiente;

}

public NodoD<E> getAnterior(){

return anterior;

}

}

public class Nodo<E> implements Position<E>{

private E elemento;

private Nodo<E> siguiente;

public Nodo() {

this(null, null);

}

public Nodo(E e) {

this(e,null);

}

public Nodo( E e, Nodo<E> sig ) {

this.elemento=e;

this.siguiente=sig;

}

public E element(){

return this.elemento;

}

public void setElemento(E elemento){

this.elemento=elemento;

}

public Nodo<E> getSiguiente(){

return this.siguiente;

}

public void setSiguiente(Nodo<E>siguiente){

this.siguiente=siguiente;

}

}

import java.util.Iterator;

import Exceptions.BoundaryViolationException;

import Exceptions.EmptyListException;

import Exceptions.InvalidPositionException;

public interface PositionList<E> extends Iterable<E>{

public int size();

public boolean isEmpty();

public Position<E> first() throws EmptyListException;

public Position<E> last() throws EmptyListException;

public Position<E> prev(Position<E> p) throws InvalidPositionException,BoundaryViolationException;

public Position<E> next(Position<E> p) throws InvalidPositionException,BoundaryViolationException;

public NodoD<E> addFirst(E e);

public void addLast(E e);

public void addAfter(Position<E> p,E e)throws InvalidPositionException;

public void addBefore(Position<E> p,E e)throws InvalidPositionException;

public E remove(Position<E>p)throws InvalidPositionException;

public Iterator<E> iterator();

public Iterable<Position<E>> positions();

}

import java.util.Iterator;

import Exceptions.BoundaryViolationException;

import Exceptions.EmptyListException;

import Exceptions.InvalidPositionException;

public class DoubleLinkedList<E> implements PositionList<E> {

private NodoD<E> head,tail;

private int longitud;

public DoubleLinkedList(){

longitud=0;

head= new NodoD<>(null, null, null);

tail= new NodoD<>(null, null, head);

head.setSiguiente(tail);

}

public int size(){ //O(1)

return longitud;

}

public boolean isEmpty(){ //O(1)

return longitud==0;

}

public Position<E> first() throws EmptyListException{ //O(1)

if(isEmpty()){

throw new EmptyListException("Lista vacia");

}

else{

return head.getSiguiente();

}

}

public Position<E> last()throws EmptyListException{ //O(1)

if(isEmpty()){

throw new EmptyListException("Lista vacia");

}

else{

return tail.getAnterior();

}

}

public Position<E> prev(Position<E> p)throws InvalidPositionException,BoundaryViolationException{//O(1)

NodoD<E> n=checkPosition(p);

if(n.getAnterior()==head){

throw new BoundaryViolationException("Lista:: prev() quiso buscar el anterior a la primera posicion");

}

else{

return n.getAnterior();

}

}

public Position<E> next(Position<E> p)throws InvalidPositionException,BoundaryViolationException{ //O(1)

NodoD<E> n=checkPosition(p);

if(n.getSiguiente()==tail){

throw new BoundaryViolationException("Lista :: next() quiso encontrar la siguiente posicion a la ultima posicion");

}

else{

return n.getSiguiente();

}

}

private NodoD<E> checkPosition(Position<E>p)throws InvalidPositionException{ //O(1)

try{

if (p==null || isEmpty()){

throw new InvalidPositionException("Posicion nula o lista vacia");

}

if(p==head){

throw new InvalidPositionException("La cabeza no es una posicion valida!");

}

if(p==tail){

throw new InvalidPositionException("La cola no es una osicion valida");

}

NodoD<E> aux=(NodoD<E>) p;

if((aux.getSiguiente()==null)&& (aux.getAnterior()==null)){

throw new InvalidPositionException("No es una posicion valida");

}

return aux;

}

catch(ClassCastException e){

throw new InvalidPositionException("La posicion corresponde a otra lista");

}

}

public NodoD<E> addFirst(E e){//O(1)

NodoD<E> n= new NodoD<>(e, head.getSiguiente(), head);

head.getSiguiente().setAnterior(n);

head.setSiguiente(n);

longitud++;

return n;

}

public void addLast(E e){ //O(1)

NodoD<E> n= new NodoD<>(e, tail, tail.getAnterior());

tail.getAnterior().setSiguiente(n);

tail.setAnterior(n);

longitud++;

}

public void addAfter(Position<E> p,E e)throws InvalidPositionException{ //O(1)

NodoD<E> n=checkPosition(p);

if(n!=tail){

NodoD<E> a= new NodoD<>(e, n.getSiguiente(), n);

n.getSiguiente().setAnterior(a);

n.setSiguiente(a);

longitud++;

}

}

public void addBefore(Position<E> p,E e)throws InvalidPositionException{//O(1)

NodoD<E> n=checkPosition(p);

NodoD<E> a= new NodoD<>(e, null, null);

if(n!=head){

a.setSiguiente(n);

a.setAnterior(n.getAnterior());

n.getAnterior().setSiguiente(a);

n.setAnterior(a);

longitud++;

}

}

public E remove(Position<E> p)throws InvalidPositionException{//O(1)

if(isEmpty())

throw new InvalidPositionException("Lista vacia");

else{

NodoD<E> n=checkPosition(p);

E aux = null;

if(n!=head && n!=tail){

aux=n.element();

n.getAnterior().setSiguiente(n.getSiguiente());

n.getSiguiente().setAnterior(n.getAnterior());

longitud--;

}

return aux;

}

}

public E set(Position<E> p,E e)throws InvalidPositionException{ //O(1)

NodoD<E> n=checkPosition(p);

E aux=null;

if(n!=head && n!=tail){

aux= n.element();

n.setElemento(e);

}

return aux;

}

public Iterator<E> iterator(){

return new ElementIterator<>(this);

}

public Iterable<Position<E>> positions(){

PositionList<Position<E>> P= new DoubleLinkedList<>();

try {

if(!isEmpty()){

Position<E> p=first();

while(true){

P.addLast(p);

if(p==last())

break;

p=next(p);

}

}

}

catch (InvalidPositionException | EmptyListException | BoundaryViolationException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return P;

}

public E removeFirst() throws InvalidPositionException {

return remove(head.siguiente);

}

}

**package business;**

public class Arco {

private int nodo1;

private int nodo2;

public Arco(int i, int j) {

this.nodo1 = i;

this.nodo2 = j;

}

public int getNodo1() {

return nodo1;

}

public int getNodo2() {

return nodo2;

}

}

public class Pesado {

private Arco arco;

private int peso;

public Pesado(int i, int j, int peso) {

this.arco = new Arco(i, j);

this.peso = peso;

}

public int getPeso(){

return peso;

}

public Arco getArco(){

return arco;

}

}

import TDAColaPrioridad.Heap;

import TDADisjointSet.DisjointSetCH;

import TDADisjointSet.DisjointSetSH;

import TDADisjointSet.NodoDisjoint;

import TDADisjointSet.NodoDisjointSH;

import TDALista.DoubleLinkedList;

import business.Pesado;

public class **Kruskal** {

private Grafo graph;

public Kruskal(Grafo graph) {

this.graph = graph;

}

public DoubleLinkedList<Pesado> arcosOrdenadosCH() throws Exception {

//Ordenamos los arcos en una lista

DoubleLinkedList<Pesado> ar = ordernarLista();

//Creamos una lista para los arcos del árbol de cubrimiento minimal

DoubleLinkedList<Pesado> t = new DoubleLinkedList<>();

int n = graph.getNodosCount();

DisjointSetCH d = new DisjointSetCH(graph.getNodosCount());

//Inicializamos la estructura de disjoint-set

d.initiate(graph.getNodos());

do {

//Obtenemos el primer arco de la lista ordenada y lo eliminamos de la misma

Pesado e = ar.removeFirst();

//Buscamos los nodos que representan al conjunto en donde se encuentran los nodos del arco removido

NodoDisjoint compu = d.find(e.getArco().getNodo1());

NodoDisjoint compv = d.find(e.getArco().getNodo2());

if (compu != compv) {

//Si son distintos conjuntos, los unimos

d.union(compu.getElemento(), compv.getElemento());

//Y agregamos el arco a la lista del árbol de cubrimiento minimal

t.addLast(e);

}

//Cortamos cuando el tamaño de la lista de arcos del árbol es igual a la cantidad de nodos - 1

} while (t.size() < n - 1);

//Retornamos la lista con los arcos que representan el árbol de cubrimiento minimal.

return t;

}

public DoubleLinkedList<Pesado> arcosOrdenadosSH() throws Exception {

//Ordenamos los arcos en una lista

DoubleLinkedList<Pesado> ar = ordernarLista();

//Creamos una lista para los arcos del árbol de cubrimiento minimal

DoubleLinkedList<Pesado> t = new DoubleLinkedList<>();

int n = graph.getNodosCount();

DisjointSetSH d = new DisjointSetSH(graph.getNodosCount());

//Inicializamos la estructura de disjoint-set

d.initiate(graph.getNodos());

do {

//Obtenemos el primer arco de la lista ordenada y lo eliminamos de la misma

Pesado e = ar.removeFirst();

//Buscamos los nodos que representan al conjunto en donde se encuentran los nodos del arco removido

NodoDisjointSH compu = d.find(e.getArco().getNodo1());

NodoDisjointSH compv = d.find(e.getArco().getNodo2());

if (compu != compv) {

//Si son distintos conjuntos, los unimos

d.union(compu.getElemento(), compv.getElemento());

//Y agregamos el arco a la lista del árbol de cubrimiento minimal

t.addLast(e);

}

//Cortamos cuando el tamaño de la lista de arcos del árbol es igual a la cantidad de nodos - 1

} while (t.size() < n - 1);

//Retornamos la lista con los arcos que representan el árbol de cubrimiento minimal.

return t;

}

private DoubleLinkedList<Pesado> ordernarLista() throws Exception {

//Creamos una lista para retornar los arcos ordenados

DoubleLinkedList<Pesado> ar = new DoubleLinkedList<>();

//Creamos un heap para ordenar los arcos, con tamaño a+1 (Porque la celda 0 del heap queda inutilizada).

Heap heap = new Heap(graph.getArcosCount() + 1);

for (Pesado pesado : graph.getArcos()) {

//Insertamos cada arco del grafo en el heap

heap.insert(pesado);

}

while (!heap.isEmpty()) {

//Insertamos cada arco removido del min-heap en la lista

ar.addLast(heap.removeMin());

}

//Retornamos la lista con los arcos ordenados por peso

return ar;

}

public DoubleLinkedList<Pesado> conHeapSH() throws Exception {

//Creamos una lista para los arcos del árbol de cubrimiento minimal

DoubleLinkedList<Pesado> t = new DoubleLinkedList<>();

//Creamos un heap que contenga los arcos del grafo

Heap heap = ordenarAenHeap();

int n = graph.getNodosCount();

DisjointSetSH d = new DisjointSetSH(n);

//Inicializamos la estructura disjoint-set

d.initiate(graph.getNodos());

do {

//Obtenemos el primer arco del heap y lo eliminamos del mismo.

Pesado e = heap.removeMin();

//Buscamos los nodos que representan al conjunto en donde se encuentran los nodos del arco removido

NodoDisjointSH compu = d.find(e.getArco().getNodo1());

NodoDisjointSH compv = d.find(e.getArco().getNodo2());

if (compu != compv) {

//Si son distintos conjuntos, los unimos

d.union(compu.getElemento(), compv.getElemento());

//Y agregamos el arco a la lista del árbol de cubrimiento minimal

t.addLast(e);

}

//Cortamos cuando el tamaño de la lista de arcos del árbol es igual a la cantidad de nodos - 1

} while (t.size() < n - 1);

//Retornamos la lista con los arcos que representan el árbol de cubrimiento minimal.

return t;

}

private Heap ordenarAenHeap() {

//Creamos un heap para ordenar los arcos, con tamaño a+1 (Porque la celda 0 del heap queda inutilizada).

Heap heap = new Heap(graph.getArcosCount() + 1);

for (Pesado pesado : graph.getArcos()) {

//Insertamos todos los arcos del grafo en el heap

heap.insert(pesado);

}

return heap;

}

public DoubleLinkedList<Pesado> conHeapCH() throws Exception {

//Creamos una lista para los arcos del árbol de cubrimiento minimal

DoubleLinkedList<Pesado> t = new DoubleLinkedList<>();

//Creamos un heap que contenga los arcos del grafo

Heap heap = ordenarAenHeap();

int n = graph.getNodosCount();

DisjointSetCH d = new DisjointSetCH(n);

//Inicializamos la estructura disjoint-set

d.initiate(graph.getNodos());

do {

//Obtenemos el primer arco del heap y lo eliminamos del mismo.

Pesado e = heap.removeMin();

//Buscamos los nodos que representan al conjunto en donde se encuentran los nodos del arco removido

NodoDisjoint compu = d.find(e.getArco().getNodo1());

NodoDisjoint compv = d.find(e.getArco().getNodo2());

if (compu != compv) {

//Si son distintos conjuntos, los unimos

d.union(compu.getElemento(), compv.getElemento());

//Y agregamos el arco a la lista del árbol de cubrimiento minimal

t.addLast(e);

}

//Cortamos cuando el tamaño de la lista de arcos del árbol es igual a la cantidad de nodos - 1

} while (t.size() < n - 1);

//Retornamos la lista con los arcos que representan el árbol de cubrimiento minimal.

return t;

}

}

import Exceptions.InvalidPositionException;

import TDACola.ColaEnlazada;

import TDACola.EmptyQueueException;

import TDACola.Queue;

import TDADisjointSet.DisjointSetCH;

import TDADisjointSet.NodoDisjoint;

import business.Pesado;

public class **Conexo** {

private int[] padre;

private Color[] color; // usamos arreglos globales asi no tenemos que pasarlos como parametro

/\*

\* Estrategia basica: Cargar los arcos en la estructura Disjoint set y luego si la cantidad de conjuntos en la estructura es 1

\* entonces el grafo es conexo

\* \*/

public boolean conexoDS(Grafo graph) throws InvalidPositionException {

DisjointSetCH d= new DisjointSetCH(graph.getNodosCount());

d.initiate(graph.getNodos());// inicializamos la ed

for (Pesado pesado: graph.getArcos()){

NodoDisjoint compu = d.find(pesado.getArco().getNodo1());//

NodoDisjoint compv = d.find(pesado.getArco().getNodo2());//buscamos el elemento identificador de cada nodo

if (compu!=compv){// si son distintos, significa que estan en conjuntos separados

d.union(compu.getElemento(), compv.getElemento());

}

if (d.parentSize()==1) // si en algun momento la cantidad de conjuntos en la ed es 1, terminamos

break;

}

return d.parentSize()==1;

}

public void BFS(Grafo graph) throws EmptyQueueException {

color = new Color[graph.getNodosCount()];// arreglo necesario para no pasar por un mismo nodo 2 veces

padre = new int[graph.getNodosCount()];// arreglo necesario para verificar la conectitud

for (int n : graph.getNodos()){

color[n] = Color.blanco;

padre[n]= -1; //inicializamos con -1, luego si hay 2 nodos con padre -1 entonces el grafo no es conexo

}

Queue<Integer> Q=new ColaEnlazada<>();

for(int n:graph.getNodos()){ // recorremos todos los nodos

if(color[n]==Color.blanco){ // si no recorri este nodo

color[n]= Color.gris;

Q.enqueue(n);

visitarBFS(graph,Q);//lo visito

}

}

}

private void visitarBFS(Grafo graph,Queue<Integer> Q) throws EmptyQueueException {

while(!Q.isEmpty()){

int n=Q.front(); // pregunto por el tope de la cola

Pesado [] adyacentes = graph.getAdyacentes(n);

for(Pesado pesado: adyacentes){ // visito todos los adyacentes al nodo tope de la cola

if(pesado!=null){

int v = pesado.getArco().getNodo2();

if(v==n){ // este if es para obtener el nodo diferente al ya obtenido por el tope de la cola

v=pesado.getArco().getNodo1();

}

if(color[v] == Color.blanco){ //si el adyacente no lo recorri

Q.enqueue(v);// lo encolo y por lo tanto lo recorro

color[v] = Color.gris;

padre[v]= n;// al adyacente le seteo que pude llegar a el, atraves del padre obtenido por el tope de la cola

}

}

}

Q.dequeue();// una vez que recorri todos los adyacentes , desencolo el nodo y lo pongo en negro

color[n] = Color.negro;

}

}

public boolean conexoBFS(Grafo graph) throws EmptyQueueException {

BFS(graph); // calculo bfs y asi obtengo al arreglo padre

int count=0;int i=0;

while ((count<2)&&(i<padre.length)) { // si el contador es mayor a 1 significa que ya encontre 2 nodos con padre -1, lo cual implica

// que no es conexo

if (padre[i++]==-1)

count++;

}

return count<2;

}

}

import TDALista.DoubleLinkedList;

import business.Pesado;

import java.util.ArrayList;

public class **Grafo** {

private int[] nodos;

private DoubleLinkedList<Pesado> arcos;

private Pesado[][] matrizAdyacencia;

public int getNodosCount(){

return this.nodos.length;

}

public int getArcosCount(){

return this.arcos.size();

}

@SuppressWarnings("rawtypes")

public Grafo(GrafoObj grafoJson){

this.nodos = grafoJson.nodos;

this.arcos = new DoubleLinkedList();

int cantNodos=nodos.length;

matrizAdyacencia=new Pesado[cantNodos][cantNodos];

for(int i=0;i<cantNodos;i++){

for(int j=0;j<cantNodos;j++){

matrizAdyacencia[i][j]=null;

}

}

Object[][] arcosJson = grafoJson.arcos;

for (int i = 0; i<arcosJson.length; i++){

ArrayList<Integer> arcoLista = new ArrayList<>();

int nodoIzquierdo=((Double) ((ArrayList) arcosJson[i][0]).get(0)).intValue();

int nodoDerecho=((Double) ((ArrayList) arcosJson[i][0]).get(1)).intValue();

arcoLista.add(nodoIzquierdo);

arcoLista.add(nodoDerecho);

Pesado pesado = new Pesado(arcoLista.get(0), arcoLista.get(1), ((Double) arcosJson[i][1]).intValue());

this.arcos.addFirst(pesado);

matrizAdyacencia[nodoIzquierdo][nodoDerecho]=pesado;

matrizAdyacencia[nodoDerecho][nodoIzquierdo]=pesado;

}

}

public static class GrafoObj {

int[] nodos;

Object[][] arcos;

}

public void showGraph(){

for(int i=0;i<matrizAdyacencia.length;i++){

for(int j=0;j<matrizAdyacencia.length;j++){

if(matrizAdyacencia[i][j]==null)System.out.print("0 ");

else System.out.print(matrizAdyacencia[i][j].getPeso()+" ");

}

System.out.println();

}

}

public DoubleLinkedList<Pesado> getArcos(){

return arcos;

}

public int[] getNodos(){

return nodos;

}

public Pesado[] getAdyacentes(int n){

return matrizAdyacencia[n];

}

}

import Exceptions.InvalidPositionException;

import TDACola.EmptyQueueException;

import TDALista.DoubleLinkedList;

import business.Pesado;

import com.google.gson.Gson;

import com.google.gson.GsonBuilder;

import java.io.InputStream;

import java.util.Scanner;

import java.util.concurrent.TimeUnit;

public class **AnalisisEmpirico**{

public static void main(String[] args) throws Exception {

Grafo grafo;

DoubleLinkedList<Resultado> lista=new DoubleLinkedList<>();

System.out.println("Obteniendo 6 grafos, cargando ....");

grafo = getGrafo(5,4,0);

lista.addFirst(analisisConectitudDS(grafo));

lista.addFirst(analisisConectitudBFS(grafo));

TimeUnit.MILLISECONDS.sleep(500);

grafo = getGrafo(50,49,0);

lista.addFirst(analisisConectitudDS(grafo));

lista.addFirst(analisisConectitudBFS(grafo));

TimeUnit.MILLISECONDS.sleep(500);

grafo = getGrafo(150,150,0);

lista.addFirst(analisisConectitudDS(grafo));

lista.addFirst(analisisConectitudBFS(grafo));

TimeUnit.MILLISECONDS.sleep(500);

grafo = getGrafo(333, 54236,0);

lista.addFirst(analisisConectitudDS(grafo));

lista.addFirst(analisisConectitudBFS(grafo));

TimeUnit.MILLISECONDS.sleep(500);

grafo = getGrafo(456, 100000,0);

lista.addFirst(analisisConectitudDS(grafo));

lista.addFirst(analisisConectitudBFS(grafo));

TimeUnit.MILLISECONDS.sleep(500);

grafo = getGrafo(500, 124750,0);

lista.addFirst(analisisConectitudDS(grafo));

lista.addFirst(analisisConectitudBFS(grafo));

TimeUnit.MILLISECONDS.sleep(500);

System.out.println("6 grafos verificando conectitud:");

int i=1;

for(Resultado rs: lista){

System.out.println("Grafo numero:"+i+++", Conexo: "+rs.isResultado()+" time:"+rs.getTime()+"ns");

}

DoubleLinkedList<ResultadoKruskal> listaKruskal;

System.out.println("Obteniendo 6 grafos conexos, cargando ....");

grafo=getGrafo(10,9,1);

mostrarKruskal(analisisArbolMinimoCubrimiento(grafo));

grafo=getGrafo(100,99,1);

mostrarKruskal(analisisArbolMinimoCubrimiento(grafo));

grafo=getGrafo(200,200,1);

mostrarKruskal(analisisArbolMinimoCubrimiento(grafo));

grafo=getGrafo(366,54236,1);

mostrarKruskal(analisisArbolMinimoCubrimiento(grafo));

grafo=getGrafo(444,10660,1);

mostrarKruskal(analisisArbolMinimoCubrimiento(grafo));

grafo=getGrafo(500,65000,1);

mostrarKruskal(analisisArbolMinimoCubrimiento(grafo));

System.out.println("6 grafos conexos, obteniendo el arbol de cubrimiento minimal:");

}

private static void mostrarKruskal(DoubleLinkedList<ResultadoKruskal> resultadosKruskal) {

int i=1;

for(ResultadoKruskal res:resultadosKruskal){

System.out.println("Grafo numero:"+i+++" time:"+res.getTime()+"ns"+" tipo:"+res.getExamen());

}

}

private static Resultado analisisConectitudDS(Grafo grafo) throws InvalidPositionException {

Conexo conexoHelper = new Conexo();

boolean resultado;

long startTime = System.nanoTime();

resultado = conexoHelper.conexoDS(grafo);

long endtime = System.nanoTime();

return new Resultado(resultado, endtime - startTime );

}

private static Resultado analisisConectitudBFS(Grafo grafo) throws EmptyQueueException {

Conexo conexoHelper =new Conexo();

boolean resultado;

long startTime = System.nanoTime();

resultado=conexoHelper.conexoBFS(grafo);

long endtime=System.nanoTime();

return new Resultado(resultado,endtime-startTime);

}

private static DoubleLinkedList<ResultadoKruskal> analisisArbolMinimoCubrimiento(Grafo grafo) throws Exception{

Kruskal kruskal =new Kruskal(grafo);

DoubleLinkedList<ResultadoKruskal> toRet=new DoubleLinkedList<>();

DoubleLinkedList<Pesado> resultado;

long startTime;

long endtime;

startTime = System.nanoTime();

resultado=kruskal.conHeapSH();

endtime=System.nanoTime();

toRet.addLast(new ResultadoKruskal(resultado,endtime-startTime, "heap sh"));

startTime = System.nanoTime();

resultado=kruskal.conHeapCH();

endtime=System.nanoTime();

toRet.addLast(new ResultadoKruskal(resultado,endtime-startTime, "heap ch"));

startTime = System.nanoTime();

resultado=kruskal.arcosOrdenadosSH();

endtime=System.nanoTime();

toRet.addLast(new ResultadoKruskal(resultado,endtime-startTime, "arcos sh"));

startTime = System.nanoTime();

resultado=kruskal.arcosOrdenadosCH();

endtime=System.nanoTime();

toRet.addLast(new ResultadoKruskal(resultado,endtime-startTime, "arcos ch"));

return toRet;

}

private static Grafo getGrafo(int nodos, int arcos,int conexo) throws Exception {

// TODO Auto-generated method stub

String consulta = "curl http://cs.uns.edu.ar/~mom/AyC2019/grafo.php?nodos="+nodos+"&arcos="+arcos+"&conexo="+conexo;

Process process = Runtime.getRuntime().exec(consulta);

InputStream inputSt = process.getInputStream();

@SuppressWarnings("resource")

Scanner s = new Scanner(inputSt).useDelimiter("\\A");

String jsonString = s.hasNext() ? s.next() : "";

Gson gson = new GsonBuilder().create();

try{

Grafo.GrafoObj gr = gson.fromJson(jsonString, Grafo.GrafoObj.class);

return new Grafo(gr);

} catch (Exception e) {

throw new Exception(jsonString);

}

}

}

public class Resultado {

private boolean resultado;

private double time;

public Resultado(boolean resultado, double time) {

this.resultado = resultado;

this.time=time;

}

public boolean isResultado() {

return resultado;

}

public double getTime() {

return time;

}

}

import TDALista.DoubleLinkedList;

import business.Pesado;

public class ResultadoKruskal {

private DoubleLinkedList<Pesado> arbolCub;

private long time;

private String examen;

public ResultadoKruskal(DoubleLinkedList<Pesado> arbolCub, long time, String examen) {

this.arbolCub = arbolCub;

this.time = time;

this.examen=examen;

}

public DoubleLinkedList<Pesado> getArbolCub() {

return arbolCub;

}

public long getTime() {

return time;

}

public String getExamen() {

return examen;

}

}

public enum Color {

blanco, negro, gris

}