**ACTIVIDAD 15: ANÁLISIS EMPÍRICO**

SECCIÓN 1: ANALISIS TEORICO ESPACIO-TEMPORAL

Sección 1.1: Porqué Lista de Adyacencias y no Matriz de Adyacencias?

Una de las primeras decisiones que fue necesario tomar para afrontar esta actividad fue la selección de la implementación de la ED Grafo más eficiente para el problema, luego de deliberar sobre la cuestión llegamos a la conclusión que la mejor implementación en este caso sería el Grafo con listas de Adyacencias, veamos el porqué

Sabemos por lo visto en teoría que el espacio de ejecución de la implementación por matriz de Adyacencia es O(n^2), mientras que el espacio para la lista de adyacencias es O(n + a), por las restricciones del problema, en el peor caso (donde n = 500 y a = (500\*499)/2, el espacio para la matriz de adyacencias seria O(500^2) = 250000 mientras que el espacio en la lista de adyacencias seria O(500 + 124750) = 125250 , la diferencia de espacio es aproximadamente la mitad y entonces se justifica el uso de la Lista de Adyacencias

Y que ocurre con el tiempo de ejecución? Sabemos que, en este aspecto, la Matriz de Adyacencias es mejor ya que al ser una matriz, los accesos a la misma se pueden realizar en tiempo constante mientras que los accesos en la lista de adyacencias son O(n+a), pero en este aspecto, es posible reducir el tiempo de los accesos a las listas por medio del uso de punteros por ende se elimina la ventaja que poseía la matriz en cuanto al orden de acceso a la estructura. Por ende, nos fue más valioso ahorrar en espacio de ejecución del grafo

Sección 1.2: Análisis de Tiempo y Espacio de las EDs:

Analicemos los tiempos y espacios de Ejecución ED por ED:

Cola: Esta ED se comporta de la misma forma que la ED Cola vista en clase, por ende, los tiempos de las operaciones son:

* Enqueue: El tiempo de ejecución de esta operación es Θ(1) porque las operaciones de setElement, setNext, asignaciones e incrementar el tamaño son de tiempo constante.
* Dequeue: El tiempo de ejecución de esta operación es Θ(1) porque todas las operaciones que se realizan son de tiempo constante.
* Size: El tiempo de ejecución de esta operación es Θ(1) porque es devolver un atributo.
* Front: El tiempo de ejecución de esta operación es Θ(1) porque es devolver un elemento de un Nodo.
* isEmpty: El tiempo de ejecución de esta operación es Θ(1) porque es una comparación con igualdad.

El espacio de la cola es O(n) debido a que almacena los n elementos que tiene la cola de entrada.

Grafo: Esta ED implementa un Grafo según la implementación de Grafo con Listas de Adyacencias según la teoría, por ende, los tiempos de ejecución son:

Disjoint-Set: El Disjoint-Set esta presente de dos formas distintas, aunque ambas implementaciones respetan que la estructura interna de los cjtos es un árbol, las operaciones son implementadas de forma distinta, ya que una de las implementaciones del disjoint Set NO hace uso de las Heurísticas vistas en teoría, entonces, denominaremos al DIsjoint-Set SH a la implementación de disjoint set que NO hace uso de las heurísticas y a Disjoint-Set CH a la implementación de Disjoint-Set que HACE uso de las heurísticas vistas en teoría, entonces, los tiempos de ejecución son:

* Disjoint-Set SH:
* Union
* Disjoint-Set CH
* Union

Heap: La ED Heap se comporta de la misma manera que la ED vista en teoría, entonces los timepos de ejecución son:

* Size: El tiempo de ejecución de esta operación es Θ(1) porque es devolver un atributo.
* isEmpty: El tiempo de ejecución de esta operación es Θ(1) porque es una comparación con igualdad.
* Min: El tiempo de ejecución de esta operación es Θ(1) porque es una comparación y devuelve el primer elemento del arreglo.
* Insert: El tiempo de ejecución es O (log n) porque se realizan operaciones de tiempo constante y se realiza lo que hace el método min\_heapify para mantener la propiedad del heap que tiene O (log n). Las comparaciones son de tiempo constante. Donde n es la cantidad de elementos del heap.
* removeMin: El tiempo de ejecución es O (log n) porque se realizan operaciones de tiempo constante y se realiza lo que hace el método min\_heapify para mantener la propiedad del heap que tiene O (log n). Las comparaciones son de tiempo constante. Donde n es la cantidad de elementos del heap.
* ReplaceKey: El tiempo de ejecución es Θ(n) – donde n son la cantidad de elementos del heap – porque se hace un foreach que en el peor de los casos recorre todos los elementos del heap.
* checkKey: El tiempo de ejecución de esta operación es Θ(1) porque es una comparación con igualdad.

El espacio del heap es O(n) donde n es la cantidad de elementos del heap almacenados en un arreglo.

Sección 1.3: Análisis de Tiempo y Espacio de los Problemas:

Analicemos los tiempos y espacios de ejecución problema por problema:

Problema 1: Grafo Conexo:

Problema A: BFS

El problema fue resuelto siguiendo la siguiente estrategia: al realizar el recorrido BFS, ir marcando cada nodo visitado y una vez terminado el recorrido, verificar si con un solo BFS pude recorrer todo el grafo

Problema B: Conjunto Disjunto

El problema fue resuelto siguiendo la siguiente estrategia, dados los arcos del grafo, se procede a recorrer los arcos del grafo, sabiendo que el Disjoint Set creado previamente ya fue inicializado con los nodos del grafo ingresado y ejecutar la operación de Union entre los nodos de cada arco si se da el caso que los cjtos que representa cada nodo son disjuntos.

Entonces, el grafo ingresado será conexo si una vez recorrido el grafo, existe UN UNICO CONJUNTO en la estructura Disjoint Set.

El recorrido de los arcos del grafo se hace por medio de un forEach que recorre TODOS los arcos del grafo y dentro de este for Each se hace unión entre los nodos si estos representan a cjtos que son disjuntos, sabemos que como el Disjoint Set usado USA las heurísticas vistas en teoría, este tiene un tiempo de ejecución de O(a log\*a)

El recuento de la cantidad de cjtos en el Disjoint Set al final se realiza con una búsqueda secuencial en la estructura disjoint set, lo cual tiene un tiempo de ejecución O(n) donde n es la cantidad de nodos en el grafo ingresado

Entonces, TConexoDisjointSet (n,a) = O(n + a log\*a)

Con respecto al Espacio, se mantiene por un lado una estructura Disjoint Set que es creada en otra clase y mantiene los arcos del grafo mientras que se mantiene en el código del problema un arreglo de arcos por ende, el espacio de ejecución del algoritmo es E = O(max(a, n log\*n)) = O(n log\* n)

Problema 2: Árbol Minimal de Cubrimiento:

Problema 1A: Kruskal Lista Ordenada con Disjoint Set Sin Heurística

Se comienza obteniendo todos los arcos existentes en el grafo y se almacenan en un ArrayList llamada arcos. Esta ArrayList <ArcoED> contiene “a” elementos (siendo “a” la cantidad de arcos en el grafo). Esta operación es de tiempo constante.

Luego se llama al metodo mergeSort con arcos. Este metodo se encarga de ordenar los “a” arcos tal que:

Si la cantidad de arcos es de 1, entonces el ArrayList está ordenado en tiempo constante, pero si es mayor, entonces se llama otra vez a mergeSort con las dos mitades del ArrayList de arcos. Luego de esto, ambas listas de arreglos van a estar ordenadas y se llama a un método merge para unirlas en una única lista de arreglos. Se retorna este resultado. El tiempo de mergeSort es un tiempo recursivo, tal que se realizan 2 llamadas recursivas con a/2 elementos (al partirse el ArrayList a la mitad).

Merge utiliza dos punteros que recorren ambos ArrayList para ir agregando el menor elemento entre los dos, al ArrayList resultante A. De esta forma se recorren todos los elementos de los dos ArrayList, es decir, merge tiene un tiempo de θ (2a).

Al tener conocer la cantidad de llamadas recursivas y el tiempo de merge (nuestra función arbitraria), podemos obtener el orden de la siguiente función.

T(a) = { 1 , si a = 1

{ 2T(a/2) + 2a , si a > 1

Por el teorema maestro comparamos la funciòn auxiliar 2a con cada uno de los tres casos posibles (utilizando log2 (2) = 1):

Comparo 2a con a. Tengo que el primer caso no se cumple ya que 2a no es del orden de O(a)

Se cumple con el segundo caso tal que 2a es del orden de θ(a) ya que

a x C1 <= a <= a x C2

Entonces tengo que mergeSort tiene un tiempo del orden θ(a lg a).

Para el recorrido utilizo un ArrayList T de ArcoED, en donde almaceno los arcos resultantes. Entra en un ciclo while que, en el peor caso, recorre todos los arcos del ArrayList arcos utilizando un ListIterator.

Los findset son operaciones constantes, pero se tiene que, al no tener heuristicas, los union, en su peor caso, involucran el recorrido de n-1 nodos, al tener que cambiar las referencias de cada uno. Entonces, en su peor caso, el proceso de kruskal tardará n^2, siendo n la cantidad de nodos en el grafo.

∈ O(n^2). Entonces se tiene que:

a log a para el mergesort.

n^2 para las operaciones de disjoint set.

Esto nos da un tiempo final de θ(n^2 a log a) para Disjoint-Set SH (siendo n la cantidad de nodos y a la cantidad de arcos).

En cuanto al espacio, se utilizan dos ArrayList (uno para el resultado final y otro para el merge) que pueden contener los a arcos del grafo, y un iterator que recorre dichos arcos. Entonces el espacio total es de 3a, es decir, O(a).

Problema 1B: Kruskal Lista Ordenada con Disjoint Set Con Heurística

Al utilizar Heuristicas, se tiene el mismo mergeSort que sin heuristicas, por lo tanto se que tendrá un tiempo θ (a log a), como fue explicado anteriormente.

En cuanto al algoritmo Kruskal con heuristica, se tiene el mismo iterator que antes, recorriendo, en el peor caso, el ArrayList de arcos por completo. Esta vez, al aplicar la función union, se tiene la heuristica de “union by rank” donde se une siempre al elemento de mayor rango. Y a la vez se aplica la “compresión de caminos” donde, al realizar el findset, se asigna recursivamente la raiz como padre de los nodos hijos para disminuir el tiempo que se toma en realizar el recorrido. Esto genera un tiempo de ejecución de θ(n log n).

Este tiempo se debe a que, por cada unión tardo un tiempo de log n, y tal acción se realiza para cada nodo en el grafo. Por lo tanto, la operación toma un tiempo de n log n.

Entonces tengo que el tiempo total es de θ(a α n).

En cuanto al espacio, se tiene al igual que antes, un ArrayList para el merge que contiene los “a” arcos del grafo para ordenarlos, y un iterator que recorre esos “a”. Tambien se tiene un ArrayList donde se almacenan los arcos resultantes de aplicar kruskal, que puede contener “a” arcos. Por lo tanto, el espacio total de esta estructura es de 3a al igual que sin heuristica, en otras palabras, O(a). Tanto con como sin heuristica, se tiene que se crean elementos enteros o unos nodos, pero estos no afectan al espacio final.

Problema 1C: Kruskal Heap con Disjoint Set Sin Heurística

Al comenzar el kruskal con heap sin Heuristica, se obtienen la cantidad de nodos totales del grafo, luego se crea un min-heap que contendrá a los nodos del grafo, ordenados tal que el menos queda ubicado en la raiz, llamado arcos.

Luego se recorre, por cada ArcoED ubicado en el grafo (es decir, una cantidad “a” de arcos), se inserta el arco actual en el heap. Al terminar, se construye un DisjointSet con los nodos del grafo.

Problema 1D: Kruskal Heap con Disjoint Set Con Heurística

SECCIÓN 2: TABLAS DE RESULTADOS EMPÍRICOS

A continuación, se muestran las tablas de resultados empíricos producto de ejecutar los algoritmos implementados para una variedad de grafos, se resaltará en amarillo el tiempo menor entre las variantes de implementación para los problemas resueltos.

Las Tablas se encuentran Ordenadas primero de menor a mayor número de nodos y en segunda instancia por número de arcos de menor a mayor, tomados en Milisegundos, cada ejercicio fue ejecutado 100 veces sobre cada grafo aquí presentado y los tiempos de las 100 ejecuciones fueron promediados para disminuir el ruido generado por el Hardware y el Sistema Operativo y la JVM lo máximo posible

Tabla de resultados empíricos para el ejercicio 1:

|  |  |  |  |
| --- | --- | --- | --- |
| Grafo | | BFS | Disjoint-Set |
| N | A |  |  |
| 2 | 1 | 1.4749998525000148E-4 | 2.6519997348000264E-4 |
| 5 | 10 | 4.34999565000435E-4 | 8.70999129000871E-4 |
| 50 | 49 | 9.309937623417923E-5 | 0.003915073769005748 |
| 71 | 900 | 8.999280057595392E-5 | 0.05120590352771778 |
| 179 | 179 | 1.252975817566721E-4 | 0.015751695992267348 |
| 190 | 300 | 5.999880002399952E-5 | 0.024269514609707808 |
| 200 | 15000 | 0,852355639458215 | 0.6332335329341318 |
| 420 | 69870 | 1,264492962945073 | 5.611111111111111 |
| 500 | 40000 | 3,819462387054663 | 4.115537848605578 |
| 500 | 124750 | 2,033061649264620 | 8.790123456790123 |

Tabla de resultados empíricos para el ejercicio 2:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Grafo | | Ordenado (NanoSeg) | | Heap | |
| N | A | C/heurística | S/heurística | C/heurística | S/heurística |
| 2 | 1 | 0,024457 | 0,026721 | 0,024003 | 0,017663 |
| 10 | 10 | 0,038496 | 0,048007 | 0,025815 | 0,025816 |
| 20 | 30 | 0,100544 | 0,129528 | 0,136322 | 0,161231 |
| 80 | 100 | 4,818838 | 7,499997 | 0.0361696 | 0.039099609 |
| 124 | 6999 | 8.888111 | 12.876223 | 3,76902 | 5,442479 |
| 173 | 10000 | 15.803196 | 18.379620 | 1.773226 | 2.0199800 |
| 211 | 210 | 0,83288 | 0,908061 | 0.078916 | 0.079798 |
| 300 | 41258 | 63.99176 | 90.40329 | 7.88477 | 8.75720 |
| 361 | 500 | 0.49387 | 0.52577 | 0.21538 | 0.27633 |
| 500 | 124750 | 225.72839 | 495.93827 | 32.27160 | 40.69135 |

SECCIÓN 3: CONCLUSIONES

Observando las tablas de resultados empíricos, es posible observar que:

* Para el Problema 1:
  + Si A >> N, el BFS tendrá un tiempo de ejecución menor al Disjoint-Set ya que el tiempo del disjoint set es dependiente de la cantidad de arcos presentes en el grafo mientras que el BFS NO necesita recorrer todos los arcos del grafo para determinar si un grafo es conexo o no
  + Si el grafo es ralo (esto es, A se acerca a N-1), el tiempo del disjoint set será mucho menor que el tiempo del BFS
  + Si A = N, el Disjoint Set parece ser más eficiente que el BFS, esto se debe a que el grafo en si será ralo y por la observación anterior, el disjoint set será más eficiente en tiempo
* Para el Problema 2:
  + Para casos pequeños, los tiempos entre las implementaciones con heurística y sin heurística varían muy poco, pero a medida que se avanza en el tamaño de la instancia, se empieza a notar que las implementaciones donde el Disjoint Set usa las heurísticas son muchísimo más eficientes que las implementaciones donde el disjoint set no usa dichas heurísticas
  + El usar un Heap como estructura auxiliar para ordenar los grafos por peso de menor a mayor reduce considerablemente el tiempo de ejecución de los algoritmos, comparado a ordenarlos por un método de ordenamiento

Esto tiene una vinculación directa con los tiempos de ejecución teóricos brindados en la sección 1, al principio, los tiempos de ejecución empíricos para los casos donde el grafo es chico NO tienen una diferencia muy notable en cuanto al tiempo, pero a medida que se avanza en los casos y el grafo a recorrer crece en tamaño, se empieza a notar una mayor brecha entre los tiempos de ejecución empíricos ya que el algoritmo de peor tiempo de ejecución empieza a crecer en tiempo mientras que el algoritmo más eficiente crece menos rápido que esta función poco eficiente

Con respecto a los algoritmos de Kruskal, se nota desde el principio que el uso del disjoint set con las heurísticas afecta directamente a la eficiencia del algoritmo frente a otro algoritmo que usa un Disjoint Set que no hace uso de las heurísticas vistas en teoría, esto viene a colación de cómo las heurísticas “ahorran” Tiempo de Ejecución en las operaciones baratas para luego amortizar el costo de una operación costosa cuando esta ocurra (esta operación siendo la unión entre conjuntos), como en los ejercicios 2B y 2D NO se cuentan con las heurísticas, la estructura Disjoint Set al querer unir conjuntos sufre todo el impacto de tener que ejecutar la unión y hace que esta operación tome mucho más tiempo (es una diferencia entre O(n^2) para un DIsjoint Set sin heurísticas contra un tiempo Supralineal para un Disjoint Set con heurísticas )

SECCIÓN 4: CODIGOS FUENTE

El lenguaje elegido para implementar las soluciones y EDs fue Java, usando el IDE Eclipse

Clase AnalisisEmpirico:

**public** **class** AnalisisEmpirico{

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

Grafo[] grafosComunes = **new** Grafo[10];

Grafo[] grafosConexos = **new** Grafo[10];

**try**{

//-----------------------------------------------------GRAFOS COMUNES-------------------------------------------------

//Caso especial, grafo más chico permitido

Grafo grafo1 = *getGrafo*(2,1);

System.***out***.println("Grafo con "+ grafo1.getNodosCount() + " nodos y "+ grafo1.getArcosCount() + " arcos construido");

//Grafo pesado pequeño, tipo de peor caso

Grafo grafo2 = *getGrafo*(5,10);

System.***out***.println("Grafo con "+ grafo2.getNodosCount() + " nodos y "+ grafo2.getArcosCount() + " arcos construido");

//Caso especial: A == N-1

Grafo grafo3 = *getGrafo*(50,49);

System.***out***.println("Grafo con "+ grafo3.getNodosCount() + " nodos y "+ grafo3.getArcosCount() + " arcos construido");

//TIPO DE MEJOR CASO: GRAFO RALO

Grafo grafo4 = *getGrafo*(71,900);

System.***out***.println("Grafo con "+ grafo4.getNodosCount() + " nodos y "+ grafo4.getArcosCount() + " arcos construido");

//Caso especial: N == A

Grafo grafo5 = *getGrafo*(179,179);

System.***out***.println("Grafo con "+ grafo5.getNodosCount() + " nodos y "+ grafo5.getArcosCount() + " arcos construido");

Grafo grafo6 = *getGrafo*(190,300);

System.***out***.println("Grafo con "+ grafo6.getNodosCount() + " nodos y "+ grafo6.getArcosCount() + " arcos construido");

Grafo grafo7 = *getGrafo*(200,15000);

System.***out***.println("Grafo con "+ grafo7.getNodosCount() + " nodos y "+ grafo7.getArcosCount() + " arcos construido");

Grafo grafo8 = *getGrafo*(420,69870);

System.***out***.println("Grafo con "+ grafo8.getNodosCount() + " nodos y "+ grafo8.getArcosCount() + " arcos construido");

Grafo grafo9 = *getGrafo*(500,40000);

System.***out***.println("Grafo con "+ grafo9.getNodosCount() + " nodos y "+ grafo9.getArcosCount() + " arcos construido");

//PEOR CASO: MAXIMO NUMERO DE NODOS Y ARCOS

Grafo grafo10 = *getGrafo*(500,124750);

System.***out***.println("Grafo con "+ grafo10.getNodosCount() + " nodos y "+ grafo10.getArcosCount() + " arcos construido");

//-----------------------------------------------------GRAFOS CONEXOS------------------------------------------------------------

//Caso Especial: grafo más chico permitido

Grafo grafo1C = *getGrafoConexo*(2,1);

System.***out***.println("Grafo Conexo con "+ grafo1C.getNodosCount() + " nodos y "+ grafo1C.getArcosCount() + " arcos construido");

//Caso especial: N == A

Grafo grafo2C = *getGrafoConexo*(10,10);

System.***out***.println("Grafo Conexo con "+ grafo2C.getNodosCount() + " nodos y "+ grafo2C.getArcosCount() + " arcos construido");

Grafo grafo3C = *getGrafoConexo*(20,30);

System.***out***.println("Grafo Conexo con "+ grafo3C.getNodosCount() + " nodos y "+ grafo3C.getArcosCount() + " arcos construido");

//TIPO DE MEJOR CASO: GRAFO RALO

Grafo grafo4C = *getGrafoConexo*(80,100);

System.***out***.println("Grafo con "+ grafo4C.getNodosCount() + " nodos y "+ grafo4C.getArcosCount() + " arcos construido");

Grafo grafo5C = *getGrafoConexo*(124,6999);

System.***out***.println("Grafo Conexo con "+ grafo5C.getNodosCount() + " nodos y "+ grafo5C.getArcosCount() + " arcos construido");

Grafo grafo6C = *getGrafoConexo*(173,10000);

System.***out***.println("Grafo Conexo con "+ grafo6C.getNodosCount() + " nodos y "+ grafo6C.getArcosCount() + " arcos construido");

//Caso especial: A == N-1

Grafo grafo7C = *getGrafoConexo*(211,210);

System.***out***.println("Grafo Conexo con "+ grafo7C.getNodosCount() + " nodos y "+ grafo7C.getArcosCount() + " arcos construido");

Grafo grafo8C = *getGrafoConexo*(300,41258);

System.***out***.println("Grafo Conexo con "+ grafo8C.getNodosCount() + " nodos y "+ grafo8C.getArcosCount() + " arcos construido");

Grafo grafo9C = *getGrafoConexo*(361,500);

System.***out***.println("Grafo Conexo con "+ grafo9C.getNodosCount() + " nodos y "+ grafo9C.getArcosCount() + " arcos construido");

//PEOR CASO: MAXIMO NUMERO DE NODOS Y ARCOS Y CONEXO

Grafo grafo10C = *getGrafoConexo*(500,124750);

System.***out***.println("Grafo Conexo con "+ grafo10C.getNodosCount() + " nodos y "+ grafo10C.getArcosCount() + " arcos construido");

grafosComunes[0] = grafo1;

grafosComunes[1] = grafo2;

grafosComunes[2] = grafo3;

grafosComunes[3] = grafo4;

grafosComunes[4] = grafo5;

grafosComunes[5] = grafo6;

grafosComunes[6] = grafo7;

grafosComunes[7] = grafo8;

grafosComunes[8] = grafo9;

grafosComunes[9] = grafo10;

grafosConexos[0] = grafo1C;

grafosConexos[1] = grafo2C;

grafosConexos[2] = grafo3C;

grafosConexos[3] = grafo4C;

grafosConexos[4] = grafo5C;

grafosConexos[5] = grafo6C;

grafosConexos[6] = grafo7C;

grafosConexos[7] = grafo8C;

grafosConexos[8] = grafo9C;

grafosConexos[9] = grafo10C;

} **catch** (Exception e) {

System.***out***.println(e.getMessage());

}

//para cada grafo Comun creado, ejecuto el problema 1 según las 2 variantes y tomo los tiempos

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

//creacion de estampillas de tiempo y tomado del tiempo para el problema 1 con BFS

EDGrafoListaAdyacencias ED = **new** EDGrafoListaAdyacencias(grafosComunes[i]);

**long** timesRun = Math.*round*(100\*( (**double**)( 100000)/grafosComunes[i].getArcosCount() )) + 1;

LocalTime t = LocalTime.*now*();

**for**(**int** v = 0; v < timesRun; v++) {

ConexoBFS p1A = **new** ConexoBFS(ED);

p1A.checkConexo();

}

**double** executionMilli = Duration.*between*(t, LocalTime.*now*()).toMillis();

**double** Time = executionMilli/timesRun;

System.***out***.println("el tiempo para el Problema 1 por BFS para el grafo" + (i+1) + " es: " + (Time) );

//creacion de estampillas de tiempo y tomado del tiempo para el problema 1 con Disjoint-Set

LocalTime t2 = LocalTime.*now*();

**for**(**int** v = 0; v < timesRun ; v++) {

ConexoDisjointSet p1A2 = **new** ConexoDisjointSet(ED);

p1A2.checkConexo();

}

**double** executionMilli2 = Duration.*between*(t2, LocalTime.*now*()).toMillis();

**double** Time2 = executionMilli2/timesRun;

System.***out***.println("el tiempo para el Problema 1 por Disjoint-Set para el grafo" + (i+1) + " es: " + (Time2) );

}

//para cada grafo Conexo creado, ejecuto el problema 2 según las 4 variantes y tomo los tiempos

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

EDGrafoListaAdyacencias ED = **new** EDGrafoListaAdyacencias(grafosConexos[i]);

//creacion de estampillas de tiempo y tomado del tiempo para el problema 2 con Arreglo Ordenado y Disjoint-Set SIN Heuristicas

**long** timesRun = Math.*round*(100\*( (**double**)( 100000)/grafosConexos[i].getArcosCount() )) + 1;

LocalTime t2 = LocalTime.*now*();

**for**(**int** v = 0; v < timesRun; v++) {

KruskalOrdenadoSH p2A2 = **new** KruskalOrdenadoSH(ED);

p2A2.Kruskal();

}

**double** executionMilli2 = Duration.*between*(t2, LocalTime.*now*()).toMillis();

**double** Time2 = executionMilli2/timesRun;

System.***out***.println("el tiempo para el Problema 2 por Lista Ordenada con Disjoint-Set SIN Heuristica para el grafo Conexo grafo" + (i+1) + "C " + " es: " + (Time2) );

//creacion de estampillas de tiempo y tomado del tiempo para el problema 2 con Arreglo Ordenado y Disjoint-Set CON Heuristicas

LocalTime t = LocalTime.*now*();

**for**(**int** v = 0; v < timesRun; v++) {

KruskalArcosOrdenados p2A = **new** KruskalArcosOrdenados(ED);

p2A.Kruskal();

}

**double** executionMilli = Duration.*between*(t, LocalTime.*now*()).toMillis();

**double** Time = executionMilli/timesRun;

System.***out***.println("el tiempo para el Problema 2 por Lista Ordenada con Disjoint-Set CON Heuristica para el grafo Conexo grafo" + (i+1) + "C " + " es: " + (Time) );

//creacion de estampillas de tiempo y tomado del tiempo para el problema 2 con Heap y Disjoint-Set SIN Heuristicas

LocalTime t4 = LocalTime.*now*();

**for**(**int** v = 0; v < timesRun; v++) {

KruskalHeapSH p2A4 = **new** KruskalHeapSH(ED);

p2A4.minimumSpanningTree();

}

**double** executionMilli4 = Duration.*between*(t4, LocalTime.*now*()).toMillis();

**double** Time4 = executionMilli4/timesRun;

System.***out***.println("el tiempo para el Problema 2 por Heap con Disjoint-Set SIN Heuristica para el grafo Conexo grafo" + (i+1) + "C " + " es: " + (Time4) );

//creacion de estampillas de tiempo y tomado del tiempo para el problema 2 con Heap y Disjoint-Set CON Heuristicas

LocalTime t3 = LocalTime.*now*();

**for**(**int** v = 0; v < timesRun; v++) {

KruskalHeapCH p2A3 = **new** KruskalHeapCH(ED);

p2A3.minimumSpanningTree();

}

**double** executionMilli3 = Duration.*between*(t3, LocalTime.*now*()).toMillis();

**double** Time3 = executionMilli3/timesRun;

System.***out***.println("el tiempo para el Problema 2 por Heap con Disjoint-Set CON Heuristica para el grafo Conexo grafo" + (i+1) + "C " + " es: " + (Time3) );

}

/\*

\* Generar varios grafos de diferente configuracion y buscar

\* arbol de cubrimiento minimal para cada uno.

\*

\* Medir el rendimiento usando timestamps. (es una clase en java.util.Date)

\*

\*/

}

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

// **TODO** Auto-generated method stub

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

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() : "";

System.***out***.println("Tengo el grafo en formato JSON. Lo convierto...");

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);

}

}

**private** **static** Grafo getGrafoConexo(**int** nodos, **int** arcos) **throws** Exception {

// **TODO** Auto-generated method stub

String consulta = "curl http://cs.uns.edu.ar/~mom/AyC2019/grafo.php?nodos="+nodos+"&arcos="+arcos+"&conexo=1";

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() : "";

System.***out***.println("Tengo el grafo en formato JSON. Lo convierto...");

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);

}

}

}

Clase Grafo (este es el modelo proveido por la catedra)

**public** **class** Grafo {

**private** **int**[] nodos;

**private** ArrayList<Pesado> arcos;

**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** ArrayList<Pesado>();

Object[][] arcosJson = grafoJson.arcos;

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

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

arcoLista.add(((Double) ((ArrayList) arcosJson[i][0]).get(0)).intValue());

arcoLista.add(((Double) ((ArrayList) arcosJson[i][0]).get(1)).intValue());

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

**this**.arcos.add(pesado);

}

}

**public** **int**[] getNodos(){

**return** nodos;

}

**public** ArrayList<Pesado> getArcos(){

**return** arcos;

}

**public** **static** **class** GrafoObj {

**int**[] nodos;

Object[][] arcos;

}

}

Clase Arco:

**public** **class** Arco {

**private** **int** nodo1;

**private** **int** nodo2;

**public** Arco(**int** i, **int** j) {

// **TODO** Auto-generated constructor stub

**this**.nodo1 = i;

**this**.nodo2 = j;

}

**public** **int** getNodoSource(){

**return** nodo1;

}

**public** **int** getNodoTarget(){

**return** nodo2;

}

}

Clase Pesado

**public** **class** Pesado {

**private** Arco arco;

**private** **int** peso;

Pesado(ArrayList<Integer> arcoLista, **int** peso) {

// **TODO** Auto-generated constructor stub

**this**.arco = **new** Arco(arcoLista.get(0), arcoLista.get(1));

**this**.peso = peso;

}

**public** Arco getArco(){

**return** arco;

}

**public** **int** getPeso(){

**return** peso;

}

}

Clase ColaConEnlaces:

**public** **class** ColaConEnlaces<T> **implements** Queue<T>{

**protected** Node<T> head;

**protected** Node<T> tail;

**protected** **int** size;

**public** ColaConEnlaces(){

head = **new** Node<T>();

tail = **new** Node<T>();

size = 0;

}

/\*\*

\* Dado un elemento, lo encola en la ED Cola

\*/

**public** **void** enqueue(T e){

Node<T> aux = **new** Node<T>();

aux.setElement(e);

aux.setNext(**null**);

**if**(size == 0){

head = aux;

}

**else**{

tail.setNext(aux);

}

tail = aux;

size++;

}

/\*\*

\* Saca un elemento de la cola, eliminandolo en el proceso

\*/

**public** T dequeue() **throws** EmptyQueueException{

T aux = **null**;

**if** (size == 0)

**throw** **new** EmptyQueueException("Cola Vacia");

aux = head.element();

head = head.getNext();

size--;

**if**(size == 0)

tail = **null**;

**return** aux;

}

/\*\*

\* devuelve el tamaño actual de la cola, en elementos

\*/

**public** **int** size(){

**return** size;

}

/\*\*

\* devuelve el elemento al frente de la cola, sin eliminarlo

\*/

**public** T front() **throws** EmptyQueueException{

T aux = **null**;

**if**(size == 0)

**throw** **new** EmptyQueueException("Cola Vacia");

**else**{

aux = head.element();

}

**return** aux;

}

/\*\*

\* Verifica si la cola esta vacia

\*/

**public** **boolean** isEmpty(){

**return** size == 0;

}

}

Clase EmptyQueueException

**public** **class** EmptyQueueException **extends** Exception {

**public** EmptyQueueException(String arr){

**super**(arr);

}

}

Clase FullQeueException

**public** **class** FullQueueException **extends** Exception {

**public** FullQueueException (String arr){

**super**(arr);

}

}

Clase Node (este Node es usado por la Cola de Enlaces )

**public** **class** Node<E> **implements** Position<E> {

**private** E element;

**private** Node<E> next;

**public** Node(){

element = **null**;

next = **null**;

}

**public** Node(E e, Node<E> n){

element = e;

next = n;

}

**public** E element(){

**return** element;

}

**public** Node<E> getNext(){

**return** next;

}

**public** **void** setElement(E e){

element = e;

}

**public** **void** setNext(Node<E> newNext){

next = newNext;

}

}

Interfaz Position (Denota una Posición en una lista que es usada por la Cola de enlaces):

**public** **interface** Position<E> {

**public** E element();

}

Interfaz Queue (representa una cola)

**public** **interface** Queue<T> {

**public** **void** enqueue(T e) **throws** FullQueueException;

**public** T dequeue() **throws** EmptyQueueException;

**public** T front() **throws** EmptyQueueException;

**public** **boolean** isEmpty();

**public** **int** size();

}

Clase DisjointSet CON HEURISTICA

**public** **class** EDDisjointSetCH{

**private** Nodo[] cjtos;

**private** **int** ultimaPos = 0;

**public** EDDisjointSetCH(ArrayList<Nodo> nodos) {

cjtos = **new** Nodo[nodos.size()];

**for**(Nodo n: nodos) {

makeSet(n);

}

}

/\*\*

\* Se encarga de crear un cjto e insertarlo en la estructura

\* **@param** n un entero que representara al elemento representante del cjto creado

\*/

**public** **void** makeSet(Nodo n){

cjtos[ultimaPos] = n;

n.setPosEnDS(ultimaPos);

n.setPadre(n);

ultimaPos++;

}

/\*\*

\* Se encarga de encontrar un cjto en la estructura

\* **@param** n un nodo pertenenciente a la estructura

\* **@return** el nodo que es representante del conjunto buscado

\*/

**public** Nodo findSet(Nodo n){

Nodo padreN = n.getPadre();

//Realizo la compresion de caminos al asignar recursivamente padres a los cjtos

**if**(!(padreN.getRotulo() == n.getRotulo())) {

n.setPadre(findSet(padreN));

}

**return** n.getPadre();

}

/\*\*

\* Se encarga de unir dos cjtos presentes en el arbol

\* **@param** x nodo perteneciente a algun cojto

\* **@param** y nodo perteneciente a algun otro cjto

\*/

**public** **void** union(Nodo x, Nodo y){

Link(findSet(x), findSet(y));

}

/\*\*

\* Se encarga de unir los conjuntos, aplicando la heuristica de Union por rango

\* **@param** x representante del primer cjto

\* **@param** y representante del segundo cjto

\*/

**private** **void** Link(Nodo x, Nodo y){

**int** rankX = x.getRango();

**int** rankY = y.getRango();

**if**(rankX > rankY) {

y.setPadre(x);

cjtos[y.getPosEnDS()] = **null**;

y.setPosEnDS(x.getPosEnDS());

ultimaPos--;

}

**else**{

x.setPadre(y);

cjtos[x.getPosEnDS()] = **null**;

x.setPosEnDS(y.getPosEnDS());

ultimaPos--;

**if**(rankX == rankY)

y.setRango(rankY + 1);

}

}

/\*\*

\* devuelve la cantidad de cjtos presentes en el Cjto-Disjunto

\*/

**public** **int** size(){

//devolver Length estaba mal porque length me devuelve la longitud del arreglo

//NO cuantos cjtos hay efectivamente en la ED

**int** i = 0;

//Busco cuantos elementos existen realmente en la ED

**for**(Nodo n : cjtos) {

**if**(n != **null**)

i++;

}

**return** i;

}

}

Clase DIsjoint Set SIN HEURISTICA:

**public** **class** EDDisjointSetSH{

**private** Nodo[] cjtos;

**private** **int** ultimaPos = 0;

**public** EDDisjointSetSH(ArrayList<Nodo> nodos) {

cjtos = **new** Nodo[nodos.size()];

**for**(Nodo n: nodos) {

makeSet(n);

}

}

/\*\*

\* Se encarga de crear un cjto e insertarlo en la estructura

\* **@param** n un entero que representarÃ¡ al elemento representante del cjto creado

\*/

**public** **void** makeSet(Nodo n){

//**TODO**: Verificar tema de como meter el nuevo nodo en la estructura en si, ver posicionamiento en el arreglo

cjtos[ultimaPos] = n;

n.setPosEnDS(ultimaPos);

n.setPadre(n);

ultimaPos++;

}

/\*\*

\* Se encarga de encontrar un cjto en la estructura

\* **@param** n un nodo pertenenciente a la estructura

\* **@return** el nodo que es representante del conjunto buscado o NULL en caso de no encontrarlo

\*/

**public** Nodo findSet(Nodo n){

Nodo padreN = n.getPadre();

//SIN HEURISTICA, solo devuelvo el representante del cjto, sin comprimir los caminos hasta la raiz

**if**(padreN.getRotulo() != n.getRotulo())

**return** findSet(padreN);

**else**

**return** n;

}

/\*\*

\* Se encarga de unir dos cjtos presentes en el arbol

\* **@param** x nodo perteneciente a algun cojto

\* **@param** y nodo perteneciente a algun otro cjto

\*/

**public** **void** union(Nodo x, Nodo y){

Nodo RepresentanteX = findSet(x);

Nodo RepresentanteY = findSet(y);

//SIN HEURISTICA! adoso el cjto x a y SIEMPRE

RepresentanteX.setPadre(RepresentanteY);

cjtos[RepresentanteY.getPosEnDS()] = **null**;

**int** dif = RepresentanteX.getRango() - RepresentanteY.getRango();

**if**(dif < 0)

RepresentanteX.setRango(RepresentanteX.getRango() + (-1\*dif));

**else**

RepresentanteX.setRango(RepresentanteX.getRango() + dif);

}

}

Clase ArcoED (Este es el tipo de arcos que maneja la estructura EDGrafoListaAdyacencias que se usa a través de la resolución de los problemas)

**public** **class** ArcoED **implements** Comparable<ArcoED>{

**private** Nodo source;

**private** Nodo target;

**private** **int** peso;

**private** **int** posArcos,posArcoSucesor,posArcoPredecesor;

**public** ArcoED(**int** p, Nodo s, Nodo t){

peso = p;

source = s;

target = t;

}

**public** **int** getPeso(){

**return** peso;

}

**public** Nodo getSource(){

**return** source;

}

**public** Nodo getTarget(){

**return** target;

}

**public** **int** getPosArcos(){

**return** posArcos;

}

**public** **int** getPosArcoSucesor(){

**return** posArcoSucesor;

}

**public** **int** getPosArcoPredecesor(){

**return** posArcoPredecesor;

}

**public** **void** setPosArcos(**int** pos){

posArcos = pos;

}

**public** **void** setPosArcoSucesor(**int** pos){

posArcoSucesor = pos;

}

**public** **void** setPosArcoPredecesor(**int** pos){

posArcoPredecesor = pos;

}

@Override

**public** **int** compareTo(ArcoED a) {

**if**(peso > a.getPeso())

**return** 1;

**if**(peso < a.getPeso())

**return** -1;

**return** 0;

}

}

Clase EDGrafoListaAdyacencias (Esta es la estructura grafo que se usa a través de la resolución de los problemas)

**public** **class** EDGrafoListaAdyacencias{

**private** ArrayList<ArcoED> arcos;

**private** ArrayList<Nodo> nodos;

**public** EDGrafoListaAdyacencias(Grafo g){

arcos = **new** ArrayList<ArcoED>();

nodos = **new** ArrayList<Nodo>();

**int**[] nodosEntrada = g.getNodos();

ArrayList<Pesado> arcosEntrada = g.getArcos();

//Para cada nodo en el grafo pasado como entrada, lo paso a mi arreglo de nodos, insertandolo en el grafo

**for**(**int** i = 0; i < g.getNodosCount(); i++){

**this**.insertarVertice(nodosEntrada[i]);

}

//para cada Arco del grafo pasado como entrada, lo paso a mi arraylist de arcos, insertandolo en el grafo, cuidando la correspondencia

//con los nodos inicio y fin

**for**(Pesado p : arcosEntrada){

//Problema acá con indexOf

Nodo n1 = buscarNodo(p.getArco().getNodoSource());

Nodo n2 = buscarNodo(p.getArco().getNodoTarget());

**int** peso = p.getPeso();

**this**.insertarArco(n1, n2, peso);

}

}

/\*\*

\* devuelve los arcos del grafo

\*/

**public** ArrayList<ArcoED> getArcos(){

**return** arcos;

}

/\*\*

\* devuelve los nodos del grafo

\*/

**public** ArrayList<Nodo> getNodos(){

**return** nodos;

}

/\*\*

\* Dado un rotulo, busco el nodo en al ED asociado a ese rotulo (SE ASUME QUE LOS ROTULOS SON TODOS DISTINTOS)

\* **@param** rot

\* **@return**

\*/

**private** Nodo buscarNodo(**int** rot) {

**for**(Nodo n : nodos) {

**if**(n.getRotulo() == rot) {

**return** n;

}

}

**return** **null**;

}

/\*\*

\* devuelve los arcos incidentes a un nodo

\* **@param** n nodo sobre el cual buscar los arcos adyacentes

\* **@return** arraylist de arcos adyacentes al nodo n

\*/

**public** ArrayList<ArcoED> incidentes(Nodo n){

**return** n.getAdyacentes();

}

/\*\*

\* dado un nodo y un arco, se obtiene el nodo opuesto

\* **@param** n nodo base

\* **@param** a arco base

\* **@return** nodo opuesto a n

\*/

**public** Nodo getOpuesto(Nodo n, ArcoED a){

**if**(a.getTarget().equals(n)){

**return** a.getSource();

}

**else**{

**return** a.getTarget();

}

}

/\*\*

\* Dado un arco, se obtienen los extremos del nodo

\* **@param** a arco

\* **@return** un arreglo de dos componentes, los cuales son los nodos extremos del arco a

\*/

**public** Nodo[] endVertices(ArcoED a){

Nodo[] nodos = **new** Nodo[2];

nodos[0] = a.getSource();

nodos[1] = a.getTarget();

**return** nodos;

}

/\*\*

\* verifica si dos nodos son adyacentes

\*/

**public** **boolean** sonAdyacentes(Nodo a, Nodo b){

**boolean** is = **false**;

ArrayList<ArcoED> arcos = a.getAdyacentes();

**for**(ArcoED e: arcos){

**if**(e.getSource() == b || e.getTarget() == b){

is = **true**;

**break**;

}

}

**return** is;

}

/\*\*

\* inserta un vertice en la ED Grafo

\* **@param** rot rotulo del vertice a crear

\*/

**public** **void** insertarVertice(**int** rot){

Nodo n = **new** Nodo(rot);

nodos.add(n);

n.setPosEnNodos(nodos.indexOf(n));

}

/\*\*

\* inserta un arco en la ED Grafo

\* **@param** s nodo fuente del arco a crear

\* **@param** t nodo destino del arco a crear

\* **@param** peso peso del arco a crear

\*/

**public** **void** insertarArco(Nodo s, Nodo t, **int** peso){

ArcoED a = **new** ArcoED(peso,s,t);

arcos.add(a);

s.getAdyacentes().add(a);

t.getAdyacentes().add(a);

a.setPosArcos(arcos.indexOf(a));

a.setPosArcoPredecesor(s.getAdyacentes().indexOf(a));

a.setPosArcoSucesor(t.getAdyacentes().indexOf(a));

}

}

Clase Nodo (Este es el Nodo que se usa tanto en el DIsjoint Set Y en el EDGrafoListaAdyacencias)

**public** **class** Nodo{

**private** **int** rotulo;

**private** ArrayList<ArcoED> adyacentes;

**private** **int** posEnListaNodos;

**private** **int** posEnDS;

**private** String color;

**private** Nodo PadreDS;

**private** **int** rango = 0;

**public** Nodo(**int** r){

rotulo = r;

adyacentes = **new** ArrayList<ArcoED>();

color = "blanco";

}

**public** **int** getRango() {

**return** rango;

}

**public** Nodo getPadre() {

**return** PadreDS;

}

**public** **int** getPosEnDS() {

**return** posEnDS;

}

**public** String getColor(){

**return** color;

}

**public** **int** getRotulo(){

**return** rotulo;

}

**public** **int** getPosEnNodos(){

**return** posEnListaNodos;

}

**public** ArrayList<ArcoED> getAdyacentes(){

**return** adyacentes;

}

**public** **void** setPosEnDS(**int** n) {

posEnDS = n;

}

**public** **void** setRango(**int** n) {

rango = n;

}

**public** **void** setPadre(Nodo p) {

PadreDS = p;

}

**public** **void** setColor(String color){

**this**.color = color;

}

**public** **void** setPosEnNodos(**int** pos){

posEnListaNodos = pos;

}

}

Clase Comparator (esto es un Comparador que se usa en el Heap):

**public** **class** Comparator<T> **implements** java.util.Comparator<T> {

@Override

**public** **int** compare(T e1, T e2) {

**return** ((Comparable<T>)e1).compareTo(e2);

}

}

Clase EmpryPriorityQueueException (es una excepción que puede arrojar la estructura Heap)

**public** **class** EmptyPriorityQueueException **extends** Exception {

**public** EmptyPriorityQueueException(String err){

**super**(err);

}

}

Clase Entrada (esto es un elemento dentro de en un Heap)

**public** **class** Entrada<K,V> **implements** Entry<K,V> {

**protected** K Key; //Usado por el comparador

**protected** V Value; //Cosa que guardo, "Etiqueta"

**public** Entrada(K k, V v){

Key = k;

Value = v;

}

//setters

/\*\*

\* modifica la componenete Key de la Entrada

\* **@param** k la nueva clave

\*/

**public** **void** setKey(K k){

Key = k;

}

/\*\*

\* modifica la componente Value de la Entrada

\* **@param** v el nuevo value

\*/

**public** **void** setValue(V v){

Value = v;

}

//getters

/\*\*

\* devuelve la clave de la Entrada

\* **@return** la clave de la Entrada

\*/

**public** K getKey(){

**return** Key;

}

/\*\*

\* devuelve el Value de la Entrada

\* **@return** el Value de la Entrada

\*/

**public** V getValue(){

**return** Value;

}

}

Interfaz Entry (representa un elemento en el Heap)

**public** **interface** Entry<K,V> {

//getters

**public** K getKey();

**public** V getValue();

**public** **void** setValue(V value);

**public** **void** setKey(K key);

}

Clase Heap (es la ED Heap que se usará en los ejercicios pedidos)

**public** **class** Heap<K,V> **implements** PriorityQueue<K, V> {

**protected** **int** size;

**protected** Entrada<K,V>[] ArbolHeap;

**protected** Comparator<K> comp;

/\* RECORDAR: sea i la posicion actual en la Heap...

\* Padre(i) = i/2

\* Hijo\_izquierdo(i) = 2i

\* Hijo\_derecho(i) = 2i+1

\*/

**public** Heap (**int** maxElems, Comparator<K> comp){

size = 0;

ArbolHeap = (Entrada<K,V>[]) **new** Entrada[maxElems];

**this**.comp = comp;

}

/\*\*

\* Consulta la cantidad de elementos de la cola.

\* **@return** Cantidad de elementos de la cola.

\*/

**public** **int** size(){

**return** size;

}

/\*\*

\* Consulta si la cola esta vacia.

\* **@return** Verdadero si la cola esta vacia, falso en caso contrario.

\*/

**public** **boolean** isEmpty(){

**return** size == 0;

}

/\*\*

\* Devuelve la entrada con menor prioridad de la cola.

\* **@return** Entrada con menor prioridad.

\* **@throws** EmptyPriorityQueueException si la cola esta vacia.

\*/

**public** Entry<K,V> min()**throws** EmptyPriorityQueueException{

**if**(ArbolHeap[1] == **null**){

**throw** **new** EmptyPriorityQueueException("Cola vacia");

}

**else**

**return** ArbolHeap[1];

}

/\*\*

\* Inserta un par clave-valor y devuelve la entrada creada.

\* **@param** key Clave de la entrada a insertar.

\* **@param** value Valor de la entrada a insertar.

\* **@return** Entrada creada.

\* **@throws** InvalidKeyException si la clave es invalida.

\*/

**public** Entry<K,V> insert(K key,V value)**throws** InvalidKeyException{

checkKey(key);

Entrada<K,V> ret = **new** Entrada<K,V>(key,value);

ArbolHeap[++size] = ret;

**int** i = size;

**boolean** seguir = **true**;

**while** (i > 1 && seguir){

Entry<K,V> EA = ArbolHeap[i];

Entry<K,V> padre = ArbolHeap[i/2];

**if**(comp.compare(EA.getKey(), padre.getKey()) < 0){ //el hijo es mas chico que el Padre, esto es una MinHeap, asi que NO Puede pasar

Entrada<K,V> aux = ArbolHeap[i]; //guardo el hijo

ArbolHeap[i] = ArbolHeap[i/2]; //mando al padre donde estaba el hijo

ArbolHeap[i/2] = aux; //mando el que inserte para arriba

i /= 2; //reduzco el contador, ya que ya hice el movimiento, me muevo hacia la raï¿½z del arbol

}

**else**{ //sino, el arbol ya esta ordenado, no es necesario seguir

seguir = **false**;

}

}

**return** ret;

}

/\*\*

\* Remueve y devuelve la entrada con menor prioridad de la cola.

\* **@return** Entrada con menor prioridad.

\* **@throws** EmptyPriorityQueueException si la cola esta vacia.

\*/

**public** Entry<K,V> removeMin()**throws** EmptyPriorityQueueException{

**if**(size == 0)

**throw** **new** EmptyPriorityQueueException("Cola Vacia");

Entry<K,V> ret = min();

**if**(size == 1){

ArbolHeap[1] = **null**;

size = 0;

}

**else**{

ArbolHeap[1] = ArbolHeap[size];

ArbolHeap[size] = **null**;

size--;

**int** i = 1;

**boolean** seguir = **true**;

**while**(seguir){

//calculo los hijos de la posicion en la que estoy

**int** HI = 2\*i;

**int** HD = (2\*i)+1;

**boolean** EHI = HI <= size(); //determino si realmente existen los hijos de la posicion

**boolean** EHD = HD <= size();

**if**(!EHI) //si no hay Hijo Izquierdo, llegue a una hoja y ya movi todo de forma ordenada, no hace falta seguir

seguir = **false**;

**else**{

**int** m; // Minimo de los hijos de i.

**if**( EHD ) { //si existe Hijo Derecho, es necesario determinar el minimo de los dos

// Calculo cual es el menor de los hijos.

**if**( comp.compare( ArbolHeap[HI].getKey(), ArbolHeap[HD].getKey()) < 0 ) m = HI;

**else** m = HD;

}

**else** m = HI; //sino, el minimo es el Hijo Izquierdo

**if**(comp.compare(ArbolHeap[i].getKey(), ArbolHeap[m].getKey()) > 0){ //si el minimo de los hijos es mas grande que el padre

Entrada<K,V> aux = ArbolHeap[i]; //Guardo el elemento que estoy moviendo

ArbolHeap[i] = ArbolHeap[m]; //intercambio de lugar el minimo de los hijos y el elemento que estoy moviendo

ArbolHeap[m] = aux;

i = m; //sigo con "m" siendo la nueva posicion

}

**else**

seguir = **false**; //el minimo de los hijos es mas chico, ya ordene la Heap

}

}

}

**return** ret;

}

/\*\*

\* metodo auxiliar para clonar Heaps

\* **@return** un clon de la Heap

\*/

**public** Heap<K,V> clone(){

Heap<K,V> aux = **new** Heap<K,V>(size,comp);

**try**{

**for**(**int** i = 1; i < size; i++){

aux.insert(ArbolHeap[i].getKey(), ArbolHeap[i].getValue());

}

}

**catch**(InvalidKeyException e){}

**return** aux;

}

**public** **void** replaceKey(Entry<K,V> e, K key){

**for**(Entry<K,V> r: ArbolHeap){

**if**(r == e){

r.setKey(key);

**break**;

}

}

}

**public** **void** remove(K key){

}

**private** **void** checkKey(K key)**throws** InvalidKeyException{

**if**(key == **null**)

**throw** **new** InvalidKeyException("Llave vacia");

}

}

Clase InvalidEntryException (es una excepción que puede arrojar el Heap)

**public** **class** InvalidEntryException **extends** Exception {

**public** InvalidEntryException(String err){

**super**(err);

}

}

Clase InvalidKeyException (es una excepción que puede arrojar el Heap)

**public** **class** InvalidKeyException **extends** Exception {

**public** InvalidKeyException(String err){

**super**(err);

}

}

Interfaz PriorityQueue (es la interfaz que terminará implementado en el Heap)

/\*\*

\* Interface PriorityQueue

\*/

**public** **interface** PriorityQueue< K, V > {

/\*\*

\* Consulta la cantidad de elementos de la cola.

\* **@return** Cantidad de elementos de la cola.

\*/

**public** **int** size();

/\*\*

\* Consulta si la cola está vacía.

\* **@return** Verdadero si la cola está vacía, falso en caso contrario.

\*/

**public** **boolean** isEmpty();

/\*\*

\* Devuelve la entrada con menor prioridad de la cola.

\* **@return** Entrada con menor prioridad.

\* **@throws** EmptyPriorityQueueException si la cola está vacía.

\*/

**public** Entry<K,V> min()**throws** EmptyPriorityQueueException;

/\*\*

\* Inserta un par clave-valor y devuelve la entrada creada.

\* **@param** key Clave de la entrada a insertar.

\* **@param** value Valor de la entrada a insertar.

\* **@return** Entrada creada.

\* **@throws** InvalidKeyException si la clave es inválida.

\*/

**public** Entry<K,V> insert(K key,V value)**throws** InvalidKeyException;

/\*\*

\* Remueve y devuelve la entrada con menor prioridad de la cola.

\* **@return** Entrada con menor prioridad.

\* **@throws** EmptyPriorityQueueException si la cola está vacía.

\*/

**public** Entry<K,V> removeMin()**throws** EmptyPriorityQueueException;

}

Interface BFS

**public** **interface** BFS<V,E> {

**public** **int** doBFS();

**public** **boolean** esConexo() **throws** FullQueueException;

}

Clase BreadthFisrtSearch (Implementa la interface BFS y hará el recorrido BFS según lo visto en teoría)

**public** **class** BreadthFirstSearch<V,E> **implements** BFS<V,E> {

**protected** EDGrafoListaAdyacencias graph;

**protected** Queue<Nodo> cola;

**public** BreadthFirstSearch(EDGrafoListaAdyacencias g){

graph = g;

cola= **new** ColaConEnlaces<Nodo>();

}

/\*\*

\* Metodo principal que se encarga de recorrer el grafo segun el algoritmo BFS

\*/

**public** **int** doBFS() {

ArrayList<Nodo> arregloNodos=graph.getNodos();

Nodo nodo\_origen=arregloNodos.get(0);

nodo\_origen.setColor("gris");

**try** {

cola.enqueue(nodo\_origen);

}

**catch**(FullQueueException e){

System.***out***.println("La cola esta llena y no puede incorporar nuevo elemento");

}

**int** visitados = visitarBFS();

**return** visitados;

}

/\*\*

\* Metodo auxiliar que realiza la visita a los nodos encontrados por el algoritmo BFS

\*/

**private** **int** visitarBFS() {

Nodo u = **null**;

**int** posicion=1;

**while**(!cola.isEmpty() && posicion < graph.getArcos().size()) {

**try** {

u=cola.front();

//Si el nodo es una hoja sin nodos incidentes, directamente lo desencolo

**if**(graph.incidentes(u).isEmpty()) {

u.setColor("negro");

cola.dequeue();

}

**else** {

//Para cada arco incidente al nodo obtenido de la cola

**for**(ArcoED e: graph.incidentes(u)){

//Obtengo el nodo opuesto según el arco que estoy considerando en este momento

Nodo z = graph.getOpuesto(u, e);

//Verifico si el nodo obtenido es Blanco, si lo es, lo agrego a la cola y sumo 1 al contador de nodos visitados

**if**( z.getColor().equals("blanco") && !(z.equals(u))) {

z.setColor("gris");

cola.enqueue(z);

posicion++;

}

**else**

{

u.setColor("negro");

cola.dequeue();

**break**;

}

}

}

}

**catch**(EmptyQueueException e) {

System.***out***.println("Empty Queue");

}

**catch**(FullQueueException e) {

System.***out***.println("Full Queue");

}

}

**return** posicion;

}

/\*\*

\* determina si un grafo es Conexo o No

\*/

**public** **boolean** esConexo() **throws** FullQueueException{

//Realizo el BFS y cuento cuantos nodos visité

**int** visitados = doBFS();

//si con un solo BFS recorrí todo el grafo y marqué todos los nodos, entonces, el grafo es conexo

**return** visitados == graph.getNodos().size();

}

}

Clase ConexoBFS (se encargará de resolver el ejercicio 1A)

**public** **class** ConexoBFS{

**private** EDGrafoListaAdyacencias grafo;

**public** ConexoBFS(EDGrafoListaAdyacencias g){

grafo = g;

}

/\*\*

\* metodo que se encarga de verificar si el grafo ingresado es conexo o no

\* por medio de un BFS

\* **@return** si el grafo es conexo o no

\*/

**public** **boolean** checkConexo(){

BFS<Nodo,ArcoED> ejercicio = **new** BreadthFirstSearch<Nodo,ArcoED>(grafo);

**boolean** esGrafoConexo = **false**;

**try** {

esGrafoConexo = ejercicio.esConexo();

} **catch** (FullQueueException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** esGrafoConexo;

}

}

Clase ConexoDisjointSet (se encarga de resolver el ejercicio 1B)

**public** **class** ConexoDisjointSet{

**private** EDDisjointSetCH DS;

**private** ArrayList<ArcoED> arcos;

**public** ConexoDisjointSet(EDGrafoListaAdyacencias g){

arcos = g.getArcos();

DS = **new** EDDisjointSetCH(g.getNodos());

}

/\*\*

\* Metodo que se encarga de verificar si el grafo ingresado es conexo o no

\* por medio de un Disjoint-Set

\* **@return** si el grafo es conexo o no

\*/

**public** **boolean** checkConexo(){

**for**(ArcoED e: arcos){

Nodo v1 = e.getSource();

Nodo v2 = e.getTarget();

**if**(! (DS.findSet(v1).equals(DS.findSet(v2)) )) {

DS.union(v1,v2);

}

}

//Si existe un solo cjto en la ED Disjoint-Set, entonces

//Encontré que todos los nodos del grafo estaban conectados entre si, por ende

//el grafo es conexo

**return** DS.size()==1;

}

}

Clase KruskalHeapCH (es el algoritmo de Kruskal que usa un Min-Heap para ordenar los arcos y un Disjoint Set que usa las heurísitcas de compresión de caminos y unión por rankeo)

**public** **class** KruskalHeapCH{

**private** PriorityQueue<ArcoED,Integer> arcos;

**private** EDDisjointSetCH DS;

**private** **int** cantNodos;

**public** KruskalHeapCH(EDGrafoListaAdyacencias g) {

arcos = **new** Heap<ArcoED,Integer>(g.getArcos().size()+1, **new** Comparator<ArcoED>() );

**for**(ArcoED a : g.getArcos()) {

**try** {

arcos.insert(a, a.getPeso());

} **catch** (InvalidKeyException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

DS = **new** EDDisjointSetCH(g.getNodos());

cantNodos = g.getNodos().size();

}

/\*\*

\* Metodo que se encarga de obtener el arbol de cubrimiento minimal segun Kruskal

\* **@return** arraylist de arcos que generan el arbol de cubrimiento minimal para el grafo ingresado

\*/

**public** ArrayList<ArcoED> minimumSpanningTree() {

ArrayList<ArcoED> T = **new** ArrayList<ArcoED>();

**while**(T.size() != (cantNodos-1)){

**try** {

ArcoED uv = arcos.removeMin().getKey();

Nodo compu = DS.findSet(uv.getSource());

Nodo compv = DS.findSet(uv.getTarget());

**if**(!(compu.equals(compv))){

DS.union(uv.getSource(), uv.getTarget());

T.add(uv);

}

} **catch** (EmptyPriorityQueueException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**return** T;

}

}

Clase KruskalHeapSH (es el algoritmo de Kruskal que usa un min-heap para ordenar los arcos y un disjoint set el cual NO usa las heurísticas vistas en teoría)

**public** **class** KruskalHeapSH{

**private** PriorityQueue<ArcoED,Integer> arcos;

**private** **int** cantNodos;

**private** EDDisjointSetSH DS;

**public** KruskalHeapSH(EDGrafoListaAdyacencias g) {

cantNodos = g.getNodos().size();

arcos = **new** Heap<ArcoED,Integer>(g.getArcos().size()+1, **new** Comparator<ArcoED>() );

**for**(ArcoED a : g.getArcos()) {

**try** {

arcos.insert(a, a.getPeso());

} **catch** (InvalidKeyException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

DS = **new** EDDisjointSetSH(g.getNodos());

}

**public** ArrayList<ArcoED> minimumSpanningTree() {

ArrayList<ArcoED> T = **new** ArrayList<ArcoED>();

**while**(T.size() != (cantNodos-1)){

**try** {

ArcoED uv = arcos.removeMin().getKey();

Nodo compu = DS.findSet(uv.getSource());

Nodo compv = DS.findSet(uv.getTarget());

**if**(!(compu.equals(compv))){

DS.union(uv.getSource(), uv.getTarget());

T.add(uv);

}

} **catch** (EmptyPriorityQueueException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**return** T;

}

}

Clase KruskalArcosOrdenados (es el algoritmo de Kruskal el cual usa un MergeSort para ordenar los arcos y usa un Disjoint Set el cual usa las heurísticas de compresión de caminos y unión por rankeo)

**public** **class** KruskalArcosOrdenados{

**private** ArrayList<ArcoED> arcos;

**private** **int** cantNodos;

EDDisjointSetCH DS;

**public** KruskalArcosOrdenados(EDGrafoListaAdyacencias grafo){

arcos = grafo.getArcos();

//Ordeno los arcos de peso menor a mayor

arcos = mergeSort (arcos);

cantNodos = grafo.getNodos().size();

DS = **new** EDDisjointSetCH(grafo.getNodos());

}

/\*\*

\* Metodoq ue se encarga de ejecutar el algoritmo de Kruskal sobre el grafo ingresado

\* **@return** un arraylist de arcos que generan el arbol de cubrimiento minimal sobre el grafo ingresado

\*/

**public** ArrayList<ArcoED> Kruskal(){

ArrayList<ArcoED> T = **new** ArrayList<ArcoED>();

ListIterator <ArcoED> iterator = arcos.listIterator();

//Obtengo un iterador y recorro los arcos para obtener el arbol de cubrimiento

**while**((T.size() != (cantNodos-1)) && iterator.hasNext () ){

ArcoED uv = iterator.next ();

Nodo compu = DS.findSet(uv.getSource());

Nodo compv = DS.findSet(uv.getTarget());

//Obtenidos los nodos del arco minimal actual, si estos NO estaban en un mismo conjunto, los uno

**if**(!(compu.equals(compv) )){

DS.union(uv.getSource(), uv.getTarget());

T.add(uv);

}

}

**return** T;

}

/\*\*

\* Metodo de ordenamiento para ordenar los arcos según el algoritmo mergeSort

\* **@param** A ArrayList de Arcos Pesados a ordenar

\* **@return** ArrayList de arcos ordenados por su peso

\*/

**public** ArrayList <ArcoED> mergeSort (ArrayList <ArcoED> A) {

**if** (A.size () == 1) {

**return** A;

}

**else** {

**int** mitad = A.size ()/2;

ArrayList <ArcoED> AIzquierdo = **new** ArrayList <ArcoED> ();

ArrayList <ArcoED> ADerecho = **new** ArrayList <ArcoED> ();

**for** (**int** I = 0; I < mitad; I++) {

AIzquierdo.add (A.get (I));

}

**for** (**int** I = mitad; I < A.size (); I++) {

ADerecho.add (A.get (I));

}

//Obtenidas las mitades del arreglo original, hago mergeSort con las mitades

AIzquierdo = mergeSort (AIzquierdo);

ADerecho = mergeSort (ADerecho);

//habiendo ordenado las mtiades recursivamente, procedo a combinar y obtener un nuevo arreglo donde todo esta ordenado

ArrayList <ArcoED> AOrdenado = **new** ArrayList <ArcoED> ();

merge (AIzquierdo, ADerecho, AOrdenado);

**return** AOrdenado;

}

}

/\*\*

\* Metodo de combinación del mergeSort

\* **@param** A1 arraylist izqueirda a combinar

\* **@param** A2 arraylist derecha a combinar

\* **@param** A arraylist que surge de combinar y ordenar los arreglos A1 y A2

\*/

**private** **void** merge (ArrayList <ArcoED> A1, ArrayList <ArcoED> A2, ArrayList <ArcoED> A) {

**int** IndiceA1 = 0;

**int** IndiceA2 = 0;

**while** (IndiceA1 < A1.size () && IndiceA2 < A2.size ()) {

**if** (A1.get (IndiceA1).getPeso () < A2.get (IndiceA2).getPeso ()) {

A.add(A1.get (IndiceA1));

IndiceA1++;

}

**else** {

A.add (A2.get (IndiceA2));

IndiceA2++;

}

}

**while** (IndiceA1 < A1.size ()) {

A.add(A1.get (IndiceA1));

IndiceA1++;

}

**while** (IndiceA2 < A2.size ()) {

A.add (A2.get (IndiceA2));

IndiceA2++;

}

}

}

Clase KruskalOrdenadoSH (es el algoritmo de Kruskal el cual usa MergeSort para ordenar los arcos del grafo y usa un Disjoint Set el cual NO usa las heurísticas vistas en clase)

**public** **class** KruskalOrdenadoSH{

**private** ArrayList<ArcoED> arcos;

**private** **int** cantNodos;

**private** EDDisjointSetSH DS;

**public** KruskalOrdenadoSH(EDGrafoListaAdyacencias grafo){

//inicializar todo aqui

arcos = grafo.getArcos();

arcos = mergeSort (arcos);

DS = **new** EDDisjointSetSH(grafo.getNodos());

}

/\*\*

\* Metodo que ejecuta el algoritmo de Kruskal sobre el grafo obtenido

\* **@return** una lista de arcos que construyen el arbol minimal de cubrimiento para ese grafo

\*/

**public** ArrayList<ArcoED> Kruskal(){

ArrayList<ArcoED> T = **new** ArrayList<ArcoED>();

ListIterator <ArcoED> iterator = arcos.listIterator();

**while**((T.size() != (cantNodos-1)) && iterator.hasNext () ){

ArcoED uv = iterator.next ();

Nodo compu = DS.findSet(uv.getSource());

Nodo compv = DS.findSet(uv.getTarget());

**if**(!(compu.equals(compv) )){

DS.union(uv.getSource(), uv.getTarget());

T.add(uv);

}

}

**return** T;

}

/\*\*

\* Metodo de ordenamiento para ordenar los arcos según el algoritmo mergeSort

\* **@param** A ArrayList de Arcos Pesados a ordenar

\* **@return** ArrayList de arcos ordenados por su peso

\*/

**public** ArrayList <ArcoED> mergeSort (ArrayList <ArcoED> A) {

**if** (A.size () == 1) {

**return** A;

}

**else** {

**int** mitad = A.size ()/2;

ArrayList <ArcoED> AIzquierdo = **new** ArrayList <ArcoED> ();

ArrayList <ArcoED> ADerecho = **new** ArrayList <ArcoED> ();

**for** (**int** I = 0; I < mitad; I++) {

AIzquierdo.add (A.get (I));

}

**for** (**int** I = mitad; I < A.size (); I++) {

ADerecho.add (A.get (I));

}

AIzquierdo = mergeSort (AIzquierdo);

ADerecho = mergeSort (ADerecho);

ArrayList <ArcoED> AOrdenado = **new** ArrayList <ArcoED> ();

merge (AIzquierdo, ADerecho, AOrdenado);

**return** AOrdenado;

}

}

/\*\*

\* Metodo de combinación del mergeSort

\* **@param** A1 arraylist izqueirda a combinar

\* **@param** A2 arraylist derecha a combinar

\* **@param** A arraylist que surge de combinar y ordenar los arreglos A1 y A2

\*/

**private** **void** merge (ArrayList <ArcoED> A1, ArrayList <ArcoED> A2, ArrayList <ArcoED> A) {

**int** IndiceA1 = 0;

**int** IndiceA2 = 0;

**while** (IndiceA1 < A1.size () && IndiceA2 < A2.size ()) {

**if** (A1.get (IndiceA1).getPeso () < A2.get (IndiceA2).getPeso ()) {

A.add(A1.get (IndiceA1));

IndiceA1++;

}

**else** {

A.add (A2.get (IndiceA2));

IndiceA2++;

}

}

**while** (IndiceA1 < A1.size ()) {

A.add ( A1.get (IndiceA1));

IndiceA1++;

}

**while** (IndiceA2 < A2.size ()) {

A.add ( A2.get (IndiceA2));

IndiceA2++;

}

}

}