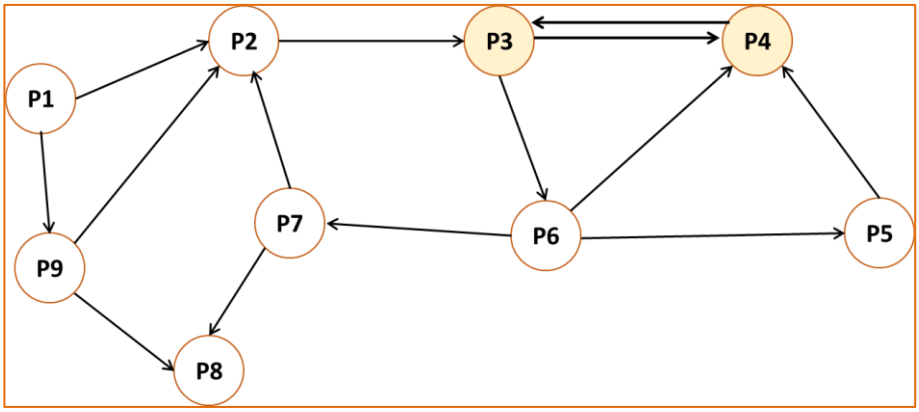
Ejercicio 3 – GRAFOS

Enunciado:
Nos contratan para diseñar una pequeña urbanización de adosados. La urbanización está formada por distintas calles y el prerrequisito inicial es que los cruces de las calles sean pequeñas placitas que favorezcan el encuentro entre los vecinos y den a la vez sensación de amplitud. Uno de los puntos a estudiar es el sentido de la circulación vial en cada uno de los tramos para que se puedan cumplir el resto de requisitos en el contrato. El concejal de urbanismo de la localidad nos propone la siguiente opción para la intercomunicación de las distintas calles (P significa "plaza"):

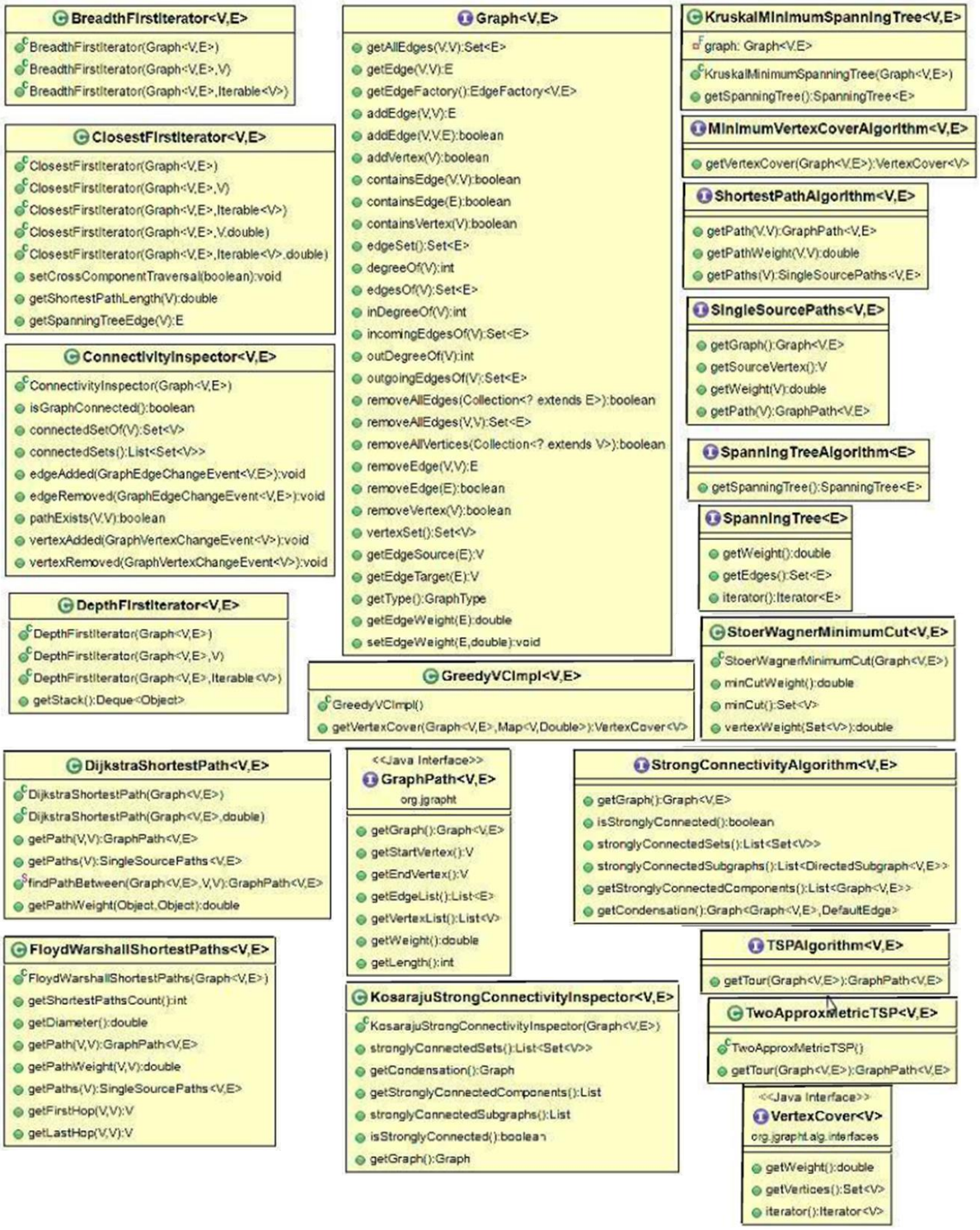


SE PIDE:

Requisito 1: Garantizar que todas las calles tengan pasos de peatones.
¿Cuántos pasos de peatones habría que pintar en cada calle si deseamos que haya uno al final de cada calle antes de llegar a cada plaza?

Requisito 2: Accesibilidad para la recogida de basuras.
Suponga que en cada plaza hay un contenedor. Justifique si la propuesta del concejal garantiza que el camión de basuras pueda llegar a todas las plazas de la urbanización. Si no fuera así, indique cómo podríamos conocer las plazas que quedarían excluidas del servicio de recogida de basuras, por lo que no podrían tener el contenedor.

Requisito 3: Carril bici.
Se desea construir un carril bici para comunicar las dos plazas principales de la urbanización, la P3 y la P4, pero se tiene el inconveniente de que es la calle con más tránsito de coches al ser la única de doble dirección. Proponer al concejal una solución alternativa diseñando la ruta más segura entre ambas plazas, que será aquella por la que transite el menor número de coches.


BreadthFirstIterator<V,E>
BreadthFirstIterator(Graph<V,E>)
BreadthFirstIterator(Graph<V,E>,V)
BreadthFirstIterator(Graph<V,E>,Iterable<V>)
ClosestFirstIterator<V,E>
ClosestFirstIterator(Graph<V,E>)
ClosestFirstIterator(Graph<V,E>,V)
ClosestFirstIterator(Graph<V,E>,Iterable<V>)
ClosestFirstIterator(Graph<V,E>,V,double)
ClosestFirstIterator(Graph<V,E>,Iterable<V>,double)
setCrossComponentTraversal(boolean):void
getShortestPathLength(V):double
getSpanningTreeEdge(V):E
ConnectivityInspector<V,E>
ConnectivityInspector(Graph<V,E>)
isGraphConnected():boolean
connectedSetOf(V):Set<V>
connectedSets():List<Set<V>>
edgeAdded(GraphEdgeChangeEvent<V,E>):void
edgeRemoved(GraphEdgeChangeEvent<V,E>):void
pathExists(V,V):boolean
vertexAdded(GraphVertexChangeEvent<V>):void
vertexRemoved(GraphVertexChangeEvent<V>):void
DepthFirstIterator<V,E>
DepthFirstIterator(Graph<V,E>)
DepthFirstIterator(Graph<V,E>,V)
DepthFirstIterator(Graph<V,E>,Iterable<V>)
getStack():Deque<Object>
DijkstraShortestPath<V,E>
DijkstraShortestPath(Graph<V,E>)
DijkstraShortestPath(Graph<V,E>,double)
getPath(V,V):GraphPath<V,E>
getPaths(V):SingleSourcePaths<V,E>
findPathBetween(Graph<V,E>,V,V):GraphPath<V,E>
getPathWeight(Object,Object):double
FloydWarshallShortestPaths<V,E>
FloydWarshallShortestPaths(Graph<V,E>)
getShortestPathsCount():int
getDiameter():double
getPath(V,V):GraphPath<V,E>
getPathWeight(V,V):double
getPaths(V):SingleSourcePaths<V,E>
getFirstHop(V,V):V
getLastHop(V,V):V
Graph<V,E>
getAllEdges(V,V):Set<E>
getEdge(V,V):E
getEdgeFactory():EdgeFactory<V,E>
addEdge(V,V):E
addEdge(V,V,E):boolean
addVertex(V):boolean
containsEdge(V,V):boolean
containsEdge(E):boolean
containsVertex(V):boolean
edgeSet():Set<E>
degreeOf(V):int
edgesOf(V):Set<E>
inDegreeOf(V):int
incomingEdgesOf(V):Set<E>
outDegreeOf(V):int
outgoingEdgesOf(V):Set<E>
removeAllEdges(Collection<? extends E>):boolean
removeAllEdges(V,V):Set<E>
removeAllVertices(Collection<? extends V>):boolean
removeEdge(V,V):E
removeEdge(E):boolean
removeVertex(V):boolean
vertexSet():Set<V>
getEdgeSource(E):V
getEdgeTarget(E):V
getType():GraphType
getEdgeWeight(E):double
setEdgeWeight(E,double):void
GreedyVCImpl<V,E>
GreedyVCImpl()
getVertexCover(Graph<V,E>,Map<V,Double>):VertexCover<V>
<<Java Interface>>
GraphPath<V,E>
org.jgrapht
getGraph():Graph<V,E>
getStartVertex():V
getEndVertex():V
getEdgeList():List<E>
getVertexList():List<V>
getWeight():double
getLength():int
KosarajuStrongConnectivityInspector<V,E>
KosarajuStrongConnectivityInspector(Graph<V,E>)
stronglyConnectedSets():List<Set<V>>
getCondensation():Graph
getStronglyConnectedComponents():List
stronglyConnectedSubgraphs():List
isStronglyConnected():boolean
getGraph():Graph
KruskalMinimumSpanningTree<V,E>
graph: Graph<V,E>
KruskalMinimumSpanningTree(Graph<V,E>)
getSpanningTree():SpanningTree<E>
MinimumVertexCoverAlgorithm<V,E>
getVertexCover(Graph<V,E>):VertexCover<V>
ShortestPathAlgorithm<V,E>
getPath(V,V):GraphPath<V,E>
getPathWeight(V,V):double
getPaths(V):SingleSourcePaths<V,E>
SingleSourcePaths<V,E>
getGraph():Graph<V,E>
getSourceVertex():V
getWeight(V):double
getPath(V):GraphPath<V,E>
SpanningTreeAlgorithm<E>
getSpanningTree():SpanningTree<E>
SpanningTree<E>
getWeight():double
getEdges():Set<E>
iterator():Iterator<E>
StoerWagnerMinimumCut<V,E>
StoerWagnerMinimumCut(Graph<V,E>)
minCutWeight():double
minCut():Set<V>
vertexWeight(Set<V>):double
StrongConnectivityAlgorithm<V,E>
getGraph():Graph<V,E>
isStronglyConnected():boolean
stronglyConnectedSets():List<Set<V>>
stronglyConnectedSubgraphs():List<DirectedSubgraph<V,E>>
getStronglyConnectedComponents():List<Graph<V,E>>
getCondensation():Graph<Graph<V,E>,DefaultEdge>
TSPAlgorithm<V,E>
getTour(Graph<V,E>):GraphPath<V,E>
TwoApproxMetricTSP<V,E>
TwoApproxMetricTSP()
getTour(Graph<V,E>):GraphPath<V,E>
<<Java Interface>>
VertexCover<V>
org.jgrapht.alg.interfaces
getWeight():double
getVertices():Set<V>
iterator():Iterator<V>

Solución:

```
public class Requisitos {

	// Requisito 1: Garantizar que todas las calles tengan pasos de peatones antes
de cada plaza
	// Solución a: contar las calles
	public static Integer pasosPeatones1(Graph<String,DefaultEdge> grafo) {
		Integer res = 0;

		res = grafo.edgeSet().size();
		return res;
	}

	// Solución b: contar el grado de entrada de cada vértice
	public static Integer pasosPeatones2(Graph<String,DefaultEdge> grafo) {
		Integer res = 0;

		res = grafo.vertexSet().stream()
					.mapToInt(v ->grafo.inDegreeOf(v)).sum();
		return res;
	}

	// Requisito 2: Accesibilidad para recogida de basuras
	// El objetivo 1º es comprobar si el grafo es fuertemente conexo
	public static Boolean recogerBasuraTodasPlazas(Graph<String,DefaultEdge>
grafo) {

	KosarajuStrongConnectivityInspector<String,DefaultEdge> g =
		new KosarajuStrongConnectivityInspector<String, DefaultEdge>(grafo);

		return g.isStronglyConnected();
	}

	// El objetivo 2ª es obtener la lista de plazas no fuertemente conexas
	public static List<Set<String>> plazasExcluidas(Graph<String,DefaultEdge>
grafo){
	List<Set<String>> res = new ArrayList<>();
	KosarajuStrongConnectivityInspector<String,DefaultEdge> g =
		new KosarajuStrongConnectivityInspector<String, DefaultEdge>(grafo);
	List<Set<String>> lista = g.stronglyConnectedSets();
	lista.stream().filter(v -> v.size() == 1)
			.forEach(v -> res.add(v));
		return res;
	}

	// Requisito 3: Carril bici
	public static List<String> getCarrilBici(Graph<String,DefaultEdge> grafoP,
String v1, String v2){
		ShortestPathAlgorithm<String,DefaultEdge> a =
				new DijkstraShortestPath<String,DefaultEdge>(grafoP);
		GraphPath<String,DefaultEdge> gp =  a.getPath(v1,v2);
		return gp.getVertexList();
	}

}
public class Test {

    public static void main(String[] args) {

	// 0) Crear el grafo
	Graph<String,DefaultEdge> grafo = creaGrafo();

	// 1) Pasos de peatones
```

```
        System.out.println("Número de pasos de peatones (contando calles) = "+
Requisitos.pasosPeatones1(grafo));
        System.out.println("Número de pasos de peatones (contando grado de entrada de
cada vértice) = "+ Requisitos.pasosPeatones2(grafo));

        // 2) Recogida de basuras
    if (Requisitos.recogerBasuraTodasPlazas(grafo)) {
        System.out.println("El camión de basuras puede llegar a todas las plazas.");
    }
    else {
        System.out.println("Plazas excluidas de la recogida de basura: "+
                Requisitos.plazasExcluidas(grafo));
    }

        // 3) Carril bici
    Graph<String,DefaultEdge> grafoP = creaGrafoConPesos();
    System.out.println("Carril bici entre P3 y P4: " +
                Requisitos.getCarrilBici(grafoP, "P3","P4"));

}

        // Grafo que representa la urbanización
        private static Graph<String,DefaultEdge> creaGrafo(){
         Graph<String,DefaultEdge> res=
                    new SimpleDirectedGraph<String, DefaultEdge>(DefaultEdge.class);

         res.addVertex("P1");res.addVertex("P2");res.addVertex("P3");
         res.addVertex("P4");res.addVertex("P5");res.addVertex("P6");
         res.addVertex("P7");res.addVertex("P8");res.addVertex("P9");

         res.addEdge("P1", "P2");res.addEdge("P2", "P3");res.addEdge("P3", "P4");
         res.addEdge("P3", "P6");res.addEdge("P4", "P3");res.addEdge("P5", "P4");
         res.addEdge("P6", "P4");res.addEdge("P6", "P5");res.addEdge("P6", "P7");
         res.addEdge("P7", "P2");res.addEdge("P7", "P8");res.addEdge("P9", "P8");
         res.addEdge("P9", "P2");res.addEdge("P1", "P9");

            return res;
        }

        // Subgrafo con pesos no dirigido para el carril bici
        private static Graph<String,DefaultEdge> creaGrafoConPesos(){

         Graph<String,DefaultEdge> res=
                    new SimpleWeightedGraph<String, DefaultEdge>(DefaultEdge.class);
         res.addVertex("P3");res.addVertex("P4");
         res.addVertex("P5");res.addVertex("P6");

         res.addEdge("P3", "P4");res.addEdge("P3", "P6");res.addEdge("P4", "P3");
         res.addEdge("P5", "P4");res.addEdge("P6", "P4");res.addEdge("P6", "P5");

         res.setEdgeWeight(res.getEdge("P3", "P4"), 100);
         res.setEdgeWeight(res.getEdge("P3", "P6"), 40);
         res.setEdgeWeight(res.getEdge("P5", "P4"), 60);
         res.setEdgeWeight(res.getEdge("P6", "P4"), 10);
         res.setEdgeWeight(res.getEdge("P6", "P5"), 20);

         return res;
        }
}
```