Ejercicio 5 – Grafos

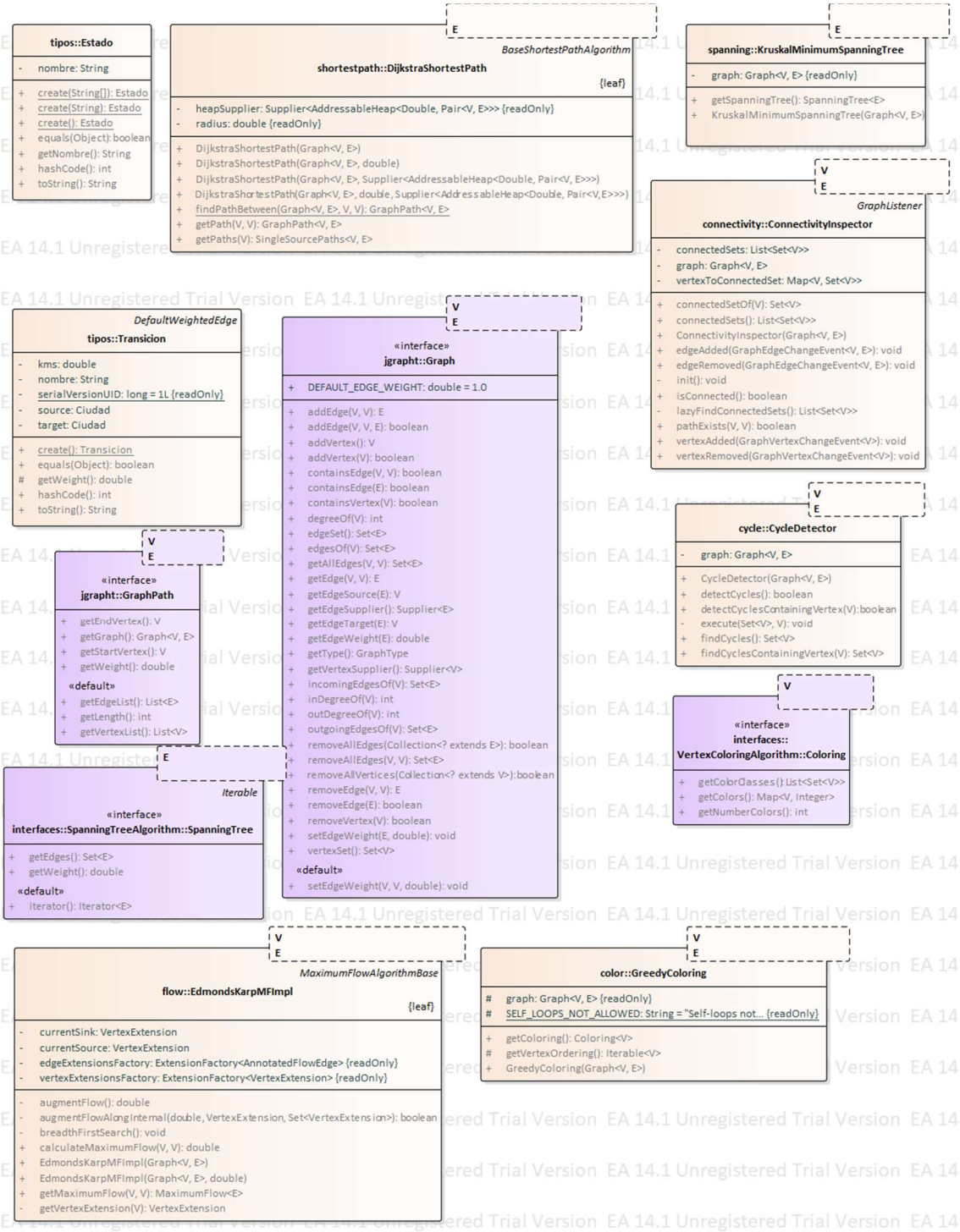
Se ha modelado en un grafo ponderado y dirigido de tipo Graph<Estado, Transicion> todos los estados en los que puede estar un reactor nuclear y las transiciones de un Estado a otro. El peso de las transiciones es el tiempo en minutos que se tarda en llegar de un Estado a otro.

SE PIDE:

Apartado A:
Sabiendo que existe un único Estado en el que el reactor funciona con completa normalidad (con nombre "Normalidad") y un único estado en el que el reactor explota (con nombre "Explota"), implemente el método public Tuple2<Integer, Integer> exploit(Graph<Estado, Transición> g) que devuelva una tupla con el número de estados y número de transiciones para hacer explotar el reactor desde el Estado de normalidad en el menor tiempo posible.

Apartado B:
Un inspector debe inspeccionar el reactor en cada uno de los estados. Un mismo inspector no puede inspeccionar dos estados adyacentes (conectados directamente por una Transición) pero sí puede inspeccionar varios estados que no estén conectados directamente por una Transición. Implemente el método public Integer minInspector(Graph<Estado, Transicion> g) que devuelva el número mínimo de inspectores necesarios para inspeccionar todos los estados.

Apartado C:
Dada una lista de estados posibles del reactor, implemente el método public List<Estado> noReturn(Graph<Estado, Transición> g, List<Estados> l) que reciba una lista de estados en el grafo y devuelva una lista con los estados de la lista de entrada que sean estados de no retorno, es decir, aquellos estados en los que, una vez alcanzado, es imposible volver al estado en el que el reactor funciona con completa normalidad.


tipos::Estado
- nombre: String
+ create(String[]): Estado
+ create(String): Estado
+ create(): Estado
+ equals(Object): boolean
+ getNombre(): String
+ hashCode(): int
+ toString(): String
E
BaseShortestPathAlgorithm
shortestpath::DijkstraShortestPath
{leaf}
- heapSupplier: Supplier<AddressableHeap<Double, Pair<V, E>>> {readOnly}
- radius: double {readOnly}
+ DijkstraShortestPath(Graph<V, E>)
+ DijkstraShortestPath(Graph<V, E>, double)
+ DijkstraShortestPath(Graph<V, E>, Supplier<AddressableHeap<Double, Pair<V, E>>>)
+ DijkstraShortestPath(Graph<V, E>, double, Supplier<AddressableHeap<Double, Pair<V,E>>>)
+ findPathBetween(Graph<V, E>, V, V): GraphPath<V, E>
+ getPath(V, V): GraphPath<V, E>
+ getPaths(V): SingleSourcePaths<V, E>
E
spanning::KruskalMinimumSpanningTree
- graph: Graph<V, E> {readOnly}
+ getSpanningTree(): SpanningTree<E>
+ KruskalMinimumSpanningTree(Graph<V, E>)
V
E
GraphListener
connectivity::ConnectivityInspector
- connectedSets: List<Set<V>>
- graph: Graph<V, E>
- vertexToConnectedSet: Map<V, Set<V>>
+ connectedSetOf(V): Set<V>
+ connectedSets(): List<Set<V>>
+ ConnectivityInspector(Graph<V, E>)
+ edgeAdded(GraphEdgeChangeEvent<V, E>): void
+ edgeRemoved(GraphEdgeChangeEvent<V, E>): void
- init(): void
+ isConnected(): boolean
- lazyFindConnectedSets(): List<Set<V>>
+ pathExists(V, V): boolean
+ vertexAdded(GraphVertexChangeEvent<V>): void
+ vertexRemoved(GraphVertexChangeEvent<V>): void
DefaultWeightedEdge
tipos::Transicion
- kms: double
- nombre: String
- serialVersionUID: long = 1L {readOnly}
- source: Ciudad
- target: Ciudad
+ create(): Transicion
+ equals(Object): boolean
# getWeight(): double
+ hashCode(): int
+ toString(): String
V
E
«interface»
jgrapht::Graph
+ DEFAULT_EDGE_WEIGHT: double = 1.0
+ addEdge(V, V): E
+ addEdge(V, V, E): boolean
+ addVertex(): V
+ addVertex(V): boolean
+ containsEdge(V, V): boolean
+ containsEdge(E): boolean
+ containsVertex(V): boolean
+ degreeOf(V): int
+ edgeSet(): Set<E>
+ edgesOf(V): Set<E>
+ getAllEdges(V, V): Set<E>
+ getEdge(V, V): E
+ getEdgeSource(E): V
+ getEdgeSupplier(): Supplier<E>
+ getEdgeTarget(E): V
+ getEdgeWeight(E): double
+ getType(): GraphType
+ getVertexSupplier(): Supplier<V>
+ incomingEdgesOf(V): Set<E>
+ inDegreeOf(V): int
+ 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
+ setEdgeWeight(E, double): void
+ vertexSet(): Set<V>
«default»
+ setEdgeWeight(V, V, double): void
V
E
cycle::CycleDetector
- graph: Graph<V, E>
+ CycleDetector(Graph<V, E>)
+ detectCycles(): boolean
+ detectCyclesContainingVertex(V):boolean
- execute(Set<V>, V): void
+ findCycles(): Set<V>
+ findCyclesContainingVertex(V): Set<V>
V
E
«interface»
jgrapht::GraphPath
+ getEndVertex(): V
+ getGraph(): Graph<V, E>
+ getStartVertex(): V
+ getWeight(): double
«default»
+ getEdgeList(): List<E>
+ getLength(): int
+ getVertexList(): List<V>
V
«interface»
interfaces::
VertexColoringAlgorithm::Coloring
+ getColorClasses(): List<Set<V>>
+ getColors(): Map<V, Integer>
+ getNumberColors(): int
E
Iterable
«interface»
interfaces::SpanningTreeAlgorithm::SpanningTree
+ getEdges(): Set<E>
+ getWeight(): double
«default»
+ iterator(): Iterator<E>
V
E
MaximumFlowAlgorithmBase
flow::EdmondsKarpMFImpl
{leaf}
- currentSink: VertexExtension
- currentSource: VertexExtension
- edgeExtensionsFactory: ExtensionFactory<AnnotatedFlowEdge> {readOnly}
- vertexExtensionsFactory: ExtensionFactory<VertexExtension> {readOnly}
- augmentFlow(): double
- augmentFlowAlongInternal(double, VertexExtension, Set<VertexExtension>): boolean
- breadthFirstSearch(): void
+ calculateMaximumFlow(V, V): double
+ EdmondsKarpMFImpl(Graph<V, E>)
+ EdmondsKarpMFImpl(Graph<V, E>, double)
+ getMaximumFlow(V, V): MaximumFlow<E>
- getVertexExtension(V): VertexExtension
V
E
color::GreedyColoring
graph: Graph<V, E> {readOnly}
SELF_LOOPS_NOT_ALLOWED: String = "Self-loops not... {readOnly}
+ getColoring(): Coloring<V>
# getVertexOrdering(): Iterable<V>
+ GreedyColoring(Graph<V, E>)

Apartado A:

```
public Tuple2<Integer, Integer> exploit(Graph<Estado, Transicion> g)  {
		int estados, transiciones;
		Estado scr = Estado.create("Normalidad");
		Estado tgt = Estado.create("Explota");

		DijkstraShortestPath alg = new DijkstraShortestPath(g);
		GraphPath<Estado, Transicion>  path = alg.getPath(scr, tgt);

		estados = path.getVertexList().size();
		transiciones = path.getLength();

		return Tuple.create(estados, transiciones);
	}
```

Apartado B:

```
public Integer minInspector(Graph<Estado, Transicion> g) {
	VertexColoringAlgorithm<Estado> alg  = new
GreedyColoring<Estado,Transicion>(g);
	VertexColoringAlgorithm.Coloring<Estado> vc = alg.getColoring();


	return vc.getNumberColors();
	 }
```

Apartado C:

```
public List<Estado> noReturn(Graph<Estado, Transicion> g, List<Estado> l)   {
	Estado tgt = Estado.create("Normalidad");
	DijkstraShortestPath alg = new DijkstraShortestPath(g);
	List<Estado> noReturn = new ArrayList<>();
	for (Estado src: l) {
		GraphPath<Estado, Transicion>   path = alg.getPath(src, tgt);
			if (path == null)
				noReturn.add(src);
		 }

	return noReturn;
}
```