Ejercicio 4. Grafos

Se dispone de la información del callejero de un pueblo, cuyas calles son todas de tierra. Modelaremos el callejero mediante un grafo, donde sus vértices serán de tipo `Cruce` y sus aristas de tipo `Calle`. Los atributos de `Calle` son:

- `longitud: int` - Longitud en metros
- `superficie: int` - Superficie en metros cuadrados
- `tiempoCoche: int` - Tiempo de recorrido en coche en segundos
- `tiempoPie: int` - Tiempo de recorrido a pie en segundos

Para mejorar el servicio de recogida de basuras se quiere situar contenedores en todos los cruces de calles y se necesita llegar a ellos por calles asfaltadas. Para lograr que un solo vehículo pueda llegar a todos los contenedores se decide asfaltar las calles necesarias con el mínimo coste posible.

1. Implementar el método

   ```
   Set<Calle> callesAsfaltado(Graph<Cruce,Calle> pueblo)
   ```

   que debe devolver el conjunto de las calles seleccionadas para conseguir el asfaltado de mínimo coste y permitir el servicio de recogida de basuras descrito, suponiendo un coste de K €/m2.

Tras el asfaltado de las calles, se logra disminuir los tiempos de recorrido en coche de las calles asfaltadas a la mitad.

2. Se quiere obtener el impacto en el tiempo de recorrido en coche de las diferentes rutas comenzando en cada uno de los cruces y terminando en los demás. Implementar el método

   ```
   double informeReduccionTiemposRutas(Graph<Cruce,Calle>
    pueblo, Set<Calle> callesAsfaltadas, Cruce c1, Cruce c2)
   ```

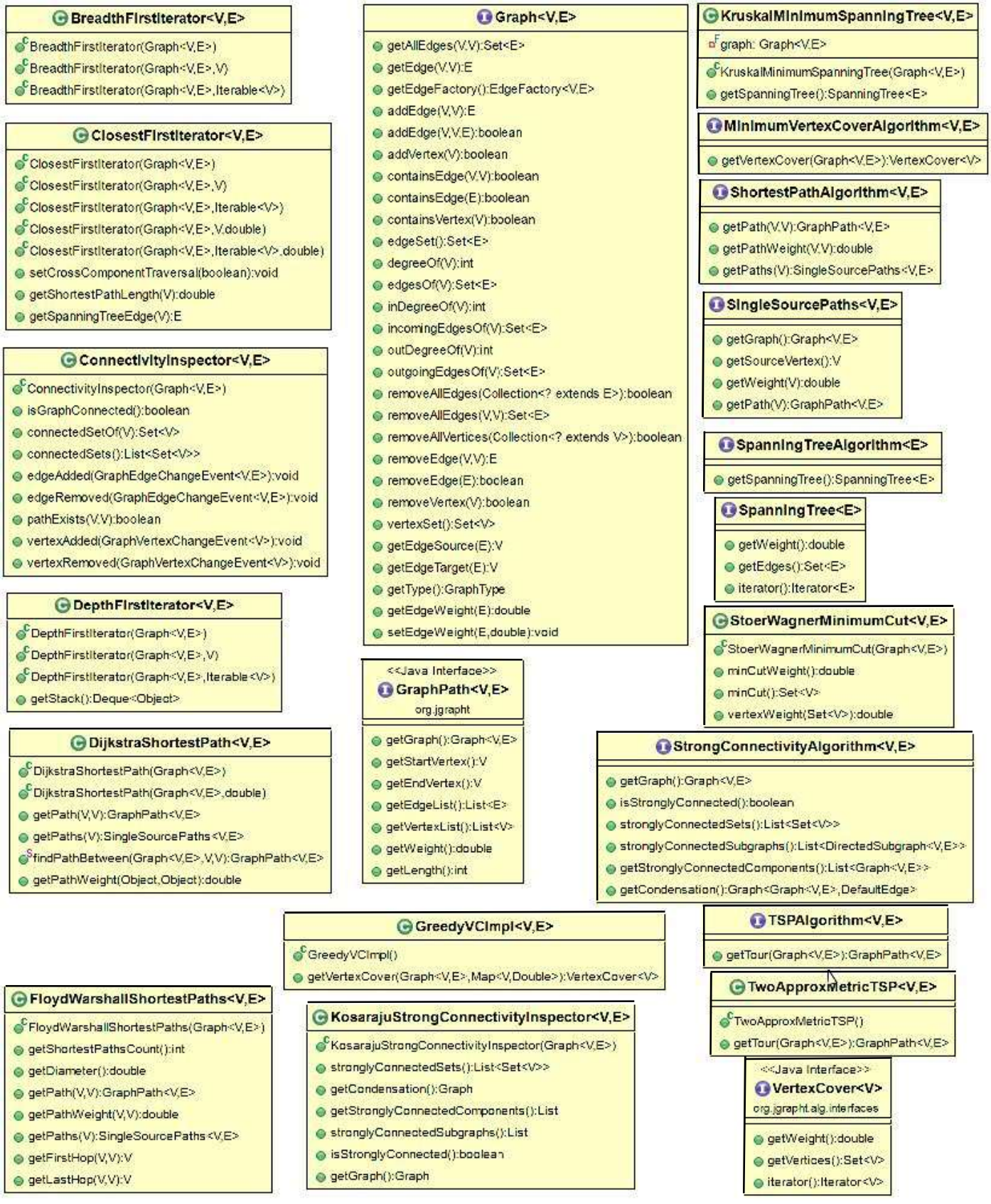
   que devolverá para cada ruta el tiempo de recorrido que se gana en coche al ir desde el cruce `c1` al cruce `c2` tras el asfaltado.

3. Se desean plantear rutas que sólo incluyan calles de tierra. Tras el asfaltado óptimo descrito, se desea averiguar si es posible una ruta que pueda ir desde cualquier cruce a cualquier otro. Para ello, implemente el método

   ```
   List<Set<Cruce>> gruposRutasPorTierra(
     Graph<Cruce,Calle> pueblo, Set<Calle> callesAsfaltadas)
   ```

   que devuelva los grupos de cruces conectados entre sí por calles de tierra.

NOTA: En cada caso debe tenerse en cuenta la medida adecuada del peso de las aristas.


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
<<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
GreedyVCImpl<V,E>
GreedyVCImpl()
getVertexCover(Graph<V,E>,Map<V,Double>):VertexCover<V>
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

1.
```
static Set<Calle> callesAsfaltado(Graph<Cruce,Calle> pueblo) {
    for (Calle c: pueblo.edgeSet()) {
        pueblo.setEdgeWeight(c, c.getSuperficie());
    }
    SpanningTreeAlgorithm<Calle> ast = new
                            KruskalMinimumSpanningTree<Cruce,Calle>(pueblo);
    return ast.getSpanningTree().getEdges();
}
```

2.
```
static double informeReduccionTiemposRutas(Graph<Cruce,Calle> pueblo,
                    Set<Calle> callesAsfaltadas, Cruce c1, Cruce c2) {
    for (Calle c: pueblo.edgeSet()) {
        pueblo.setEdgeWeight(c, c.getTiempoCoche());
    }
    ShortestPathAlgorithm<Cruce,Calle> acm = new
                                DijkstraShortestPath<Cruce,Calle>(pueblo);
    double tiempoReducido = acm.getPathWeight(c1,c2);
    for (Calle c: callesAsfaltadas) {
        pueblo.setEdgeWeight(c, c.getTiempoCoche()/2.0);
    }
    ShortestPathAlgorithm<Cruce,Calle> acm1 = new
                                DijkstraShortestPath<Cruce,Calle>(pueblo);
    tiempoReducido -= acm1.getPathWeight(c1,c2);
    return tiempoReducido;
}
```

3.
```
static List<Set<Cruce>> gruposRutasPorTierra(Graph<Cruce,Calle> pueblo,
                                                Set<Calle> callesAsfaltadas) {
    pueblo.removeAllEdges(callesAsfaltadas);
    return new ConnectivityInspector<>(pueblo).connectedSets();
}
```