Sugerencias de Implantación P5 ADSOF

Apartado 1

- Implementar Grafos con Nodos y Conectores.
- No cerrar el tipo de los Nodos y Conectores (Genericidad/Colecciones)
  - Ejemplos
  - public class Graph<N, E> …
  - Que tendrá atributos como….
    - List<Node<N>> nodes …
    - Map<Node<N>, List<Edge<E, N>>> edges ….
  - public class Node<N>
  - Que tendrá atributos como….
    - N value;
    - Graph<N, ?> graph = null;
    - ….
  - public class Edge<E, N> {
  - Que tendrá atributos como….
    - Node<N> source;
    - E value;
    - Node<N> target;
- Ojo con la gestión de ids en los Node

Apartado 2

- Funciones para Reglas en ConstrainedGraph
  - UNIVERSALES: boolean forAll(Predicate<Node<N>> pred)
  - EXISTENCIALES: boolean exists(Predicate<Node<N>> pred)
  - UNITARIAS: boolean one(Predicate<Node<N>> pred)
- Recorrer los nodos del grafo y comprobar cumplimiento
  - for (Node<N> n : this.nodes) {
  - if (pred.test(n)) {

- “…devolverá dicho nodo, o null. Para unificar ambas posibilidades el método deberá devolver un objeto Optional.”
- public Optional<Node<N>> getWitness() {
- return Optional.ofNullable(this.nodeWitness);
- }

- “…clase comparadora de grafos llamada BlackBoxComparator”
- public class BlackBoxComparator<N, E> implements Comparator<ConstrainedGraph<N, E>>{
- …que tendrá una lista de criterios, que puede ser un mapa de criterios
- private Map<Criteria, List<Predicate<Node<N>>>> criteria = new EnumMap<Criteria, List<Predicate<Node<N>>>>(Criteria.class);
- … y un compare (para que sea el Comparator)
- public int compare(ConstrainedGraph<N, E> g0, ConstrainedGraph<N, E> g1) { ….
- Hacer una enum “Criteria”
- public enum Criteria {
- Existential{
- @Override public <N, E> boolean eval(ConstrainedGraph<N, E> g, Predicate<Node<N>> p) {
- return g.exists(p); ...

Apartado 3

- “…. Una vez que hemos definido un conjunto de reglas, podemos evaluarlas sobre un stream de objetos del tipo correspondiente. …”
  - rs.getExecutionContext().stream().map(rs)……..

- Crear clases Rule y RuleSet
  - public class Rule<U>
    - Que tendrá los métodos when, exec, accept
  - public class RuleSet<U>
    - Que tendrá los métodos de procesamiento de la lista de reglas. (muy poco código)

Apartado 4

- El de menos código de todos. Según el contexto de ejecución (getExecutionContext y setExecutionContext por ejemplo en RuleSet, decidiré si es Sequence y AsLongAsPossible.
- Las clases Sequence y AsLongAsPossible
  - @Override
  - public void process(RuleSet<U> rs) {
    - Se diferenciarán en la implementación de este método

Diferencia entre estrategias “Sequence” y “As Long As Possible”

- Muy simple. Si elegimos “Sequence” aplicará el predicado al primer elemento de un conjunto que lo cumpla.
  - rs.getExecutionContext().stream().map(rs). …
- Si elegimos “As Long As Possible” aplicaremos las reglas a todos los elementos del stream que lo cumplan:
  - while (rs.getExecutionContext().stream().anyMatch(rs)) { … (aplicar sequence a esos subconjuntos)
    - Nótese que el “algoritmo de Dijkstra” planteado consiste en la variante de obtener la distancia mínima de todos los nodos a uno inicial y busca aplicar la regla a todas las posibles conexiones.