

Actividad 8. Esquema Algorítmico Voraz

Objetivo

Resolver problemas que siguen el esquema algorítmico voraz.

Procedimiento

1. Ver el vídeo o leer la presentación sobre esquemas algorítmicos que están disponibles en Moodle, Tema 4/Sección 4.1 Esquema Algorítmico: Voraz/ Recursos didácticos.
2. Resolver los ejercicios que se proponen en esta actividad, para reforzar el esquema voraz. Para ello haz uso de la clase `GreedyAlgorithm` situada en el paquete `es.uvigo.esei.aed2.activity8`, la cual contiene las cabeceras de los métodos.
3. Para probar que la implementación realizada funciona correctamente se proporcionan la clase de prueba `GreedyAlgorithmTestCase`.

Evaluación

Estos contenidos serán evaluados mediante una prueba individual el 19 de enero de 2026.

Tiempo estimado

5 horas y media

Ejercicios

1. Algoritmo del **Viajante**. Consideremos un mapa de carreteras, con dos tipos de componentes: las ciudades (*nodos*) y las carreteras que las unen. Cada tramo de carreteras (*arco*) está señalado con su longitud. “Un viajante debe recorrer una serie de ciudades interconectadas entre sí, de manera que recorra todas ellas con el menor número de kilómetros posible, comenzando en una ciudad determinada”. Emplea una estrategia **voraz** para resolver el ejercicio.

```
public static <T> Graph<T, Integer> traveller(Graph<T, Integer> graph,  
Vertex<T> vertex)
```

2. Algoritmo de **Prim**. Es un algoritmo perteneciente a la teoría de los grafos para encontrar un árbol recubridor mínimo en un grafo conexo, **no** dirigido y cuyas aristas están etiquetadas. En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo. Consideremos un mapa de carreteras, con dos tipos de

componentes: las ciudades (*nodos*) y las carreteras que las unen. Cada tramo de carreteras (*arco*) está señalado con su longitud. Se desea implantar un tendido eléctrico siguiendo los trazos de las carreteras de manera que conecte todas las ciudades y que la longitud total sea mínima. Emplea una estrategia **voraz** para resolver el ejercicio.

```
public static <T> Graph<T, Integer> prim(Graph<T, Integer> graph, Vertex<T> vertex)
```

3. Escribe el **algoritmo de Dijkstra**, también llamado **algoritmo de caminos mínimos**, es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo dirigido y con pesos en cada arista.

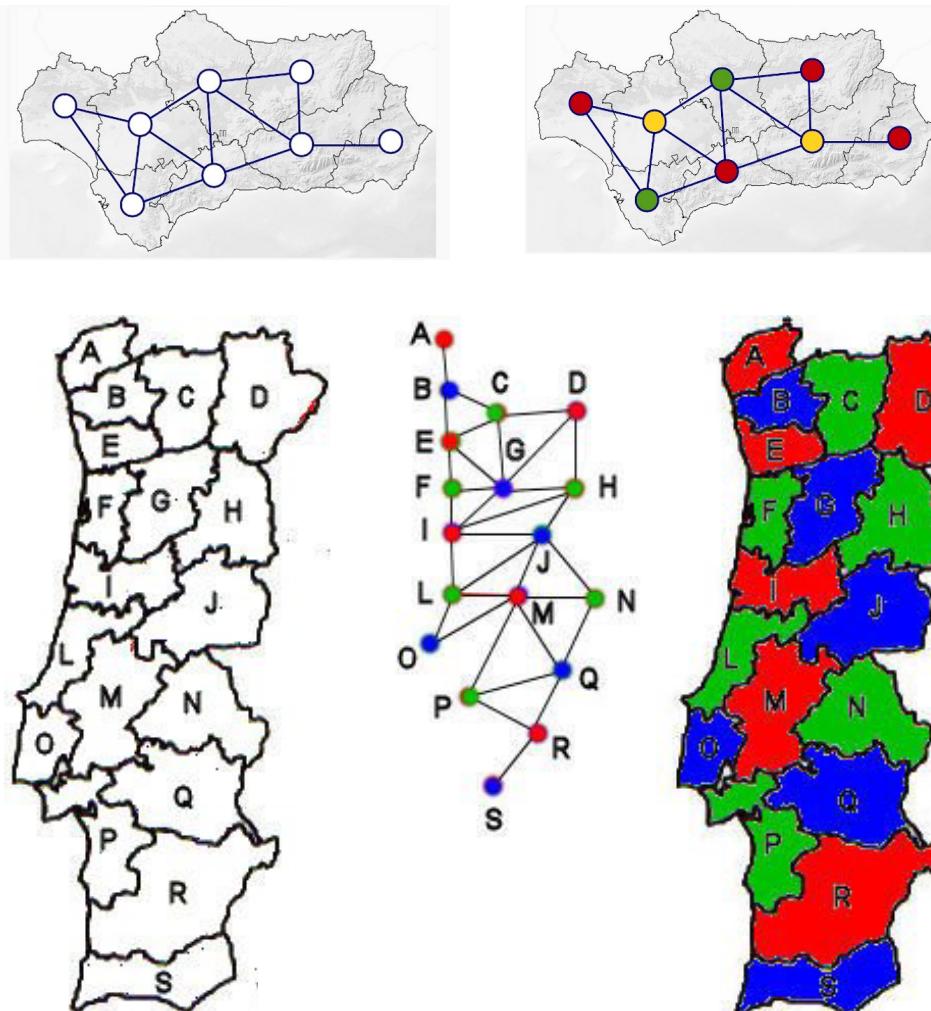
La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a los demás vértices; cuando se obtiene el camino más corto desde el vértice origen, al resto de vértices que componen el grafo, el algoritmo se detiene. Emplea una estrategia **voraz** para resolver el ejercicio.

```
public static <T> Map<Vertex<T>, Integer> dijkstra(Graph<T, Integer> graph, Vertex<T> vertex)
```

4. El **algoritmo de número cromático** indica el número de colores mínimo necesario para colorear los vértices de un grafo. El *teorema de los cuatro colores* justifica que el número cromático de un grafo plano es menor o igual que cuatro.

El objetivo de este ejercicio consiste en, dado un grafo plano que representa un mapa político, indicar cómo deben colorearse cada uno de los vértices de dicho grafo, empleando el menor número posible de colores (según el teorema citado anteriormente como máximo cuatro). Emplea una estrategia **voraz** para resolver el ejercicio.

```
public static <T> Map<Vertex<T>, String> colourMap(Graph<T, Integer> g, String[] colours)
```



5. Supongamos que sólo están disponibles los siguientes billetes y monedas: 100, 50, 20, 10, 5 y 1 euro (con un número limitado de cada tipo de moneda). Nuestro problema consiste en diseñar un algoritmo para pagar una cierta cantidad a un cliente, utilizando el *menor* número posible de billetes y monedas. Por ejemplo, si tenemos que pagar 289 euros y tenemos 5 billetes y monedas de cada tipo, la mejor solución consiste en dar al cliente los siguientes billetes/monedas: 2 de 100, 1 de 50, 1 de 20, 1 de 10, 1 de 5 y 4 de 1 euro. Implementa este algoritmo siguiendo una estrategia **voraz**.

```
public static Map<Integer, Integer> giveChange(int amountReturned, Map<Integer, Integer> changeAvailable)
```

6. Considere que tenemos n programas distintos para grabar en un disco, pero el espacio de memoria que necesitan excede la capacidad del disco. Cada programa P_i requiere m_i kilobytes de memoria, la capacidad del disco es de C kilobytes y $C < \sum_{i=1}^n m_i$

Diseñe un algoritmo, utilizando un esquema **voraz**, que calcule una colección de estos programas para grabar en el disco, de manera que se utilice la *máxima* capacidad posible del disco en la grabación de estos programas.

El método principal tendrá esta cabecera:

```
public static Set<String> burnCD(int maximumCapacity, Map<String, Integer> espacePrograms)
```

donde:

maximumCapacity: es la capacidad del disco

espacePrograms: es un Map que contiene el nombre de cada programa que queremos almacenar (identificado por un String) y el espacio que ocupa (Integer)

El método retornará un conjunto que contiene el nombre de cada programa que se eligió para almacenar en el disco.

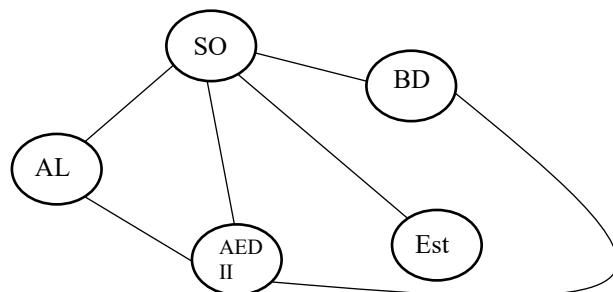
7. Problema de la mochila. Se desea llenar una mochila hasta un volumen máximo V, y para ello se dispone de n objetos, en cantidades limitadas c_1, \dots, c_n y cuyos valores por unidad de volumen son v_1, \dots, v_n , respectivamente. Debe seleccionarse de cada objeto una cantidad máxima k_i con tal de que $k_i * v_i \leq V$. El problema consiste en determinar las cantidades k_1, \dots, k_n que llenan la mochila *maximizando* el valor total $\sum v_i * k_i$, $i \in \{1, \dots, n\}$. Emplea una estrategia **voraz** para resolver el ejercicio.

```
public static Map<String, Integer> fillRucksack(int maxVol, Map<String, Integer> amounts, Map<String, Integer> volumes)
```

8. Dado un grafo con las asignaturas (vértices) y las conexiones entre ellas (arcos), las cuales indican que algún profesor imparte ambas asignaturas, implementa el método que se describe a continuación para establecer los días de examen de dichas asignaturas. El método devuelve un mapa con las asignaturas y el día de la semana en el que se realizará la prueba de dicha asignatura. Los exámenes se realizarán en una semana (de lunes a viernes de 10:00 a 14:00).

El objetivo es minimizar el número de días para la realización de todas las pruebas, garantizando que un profesor no puede examinar en diferentes asignaturas el mismo día; para lo cual se debe emplear una **estrategia voraz**.

```
public static Map<Vertex<String>, String> examSchedule(Graph<String, Integer> g, String[] daysWeek)
```



En el ejemplo, hay conexión entre SO y el resto: BD, Est, AEDII y AL (lo cual significa que hay algún profesor que da clase en todas); BD con SO y AEDII; Est con SO; AEDII con BD, SO, AL y por último AL con SO y AEDII.

Si tenemos un array con diasSemana = {Lunes, Martes, Miércoles, Jueves, Viernes}

Una posible solución, que sólo utilizaría los tres primeros días de la semana, sería:

SO – Lunes; BD – Martes; Est- Martes; AEDII- Miércoles; AL- Martes

9. La Escuela Superior de Ingeniería Informática (ESEI) ha organizado para el día del patrón “Beato Aparicio” un montón de actividades. Cada actividad tiene un nombre, una hora de inicio y hora de fin (ver clase Activity). Se desea asistir al mayor número posible de actividades, pero no se puede estar en más de una actividad al mismo tiempo, para lo cual se ha desarrollado un planificador de actividades.

Por ejemplo, si las actividades organizadas por la ESEI son:

{(Act1, <9,12>), (Act2, <8,10>), (Act3, <10,13>), (Act4, <12,14>), (Act5, <10,12>), (Act6 <12, 15>)}

El planificador determinará que asistas:

Actividades seleccionadas: {Act2, Act5, Act4}

Siguiendo una estrategia **voraz** implementa el algoritmo que planifique la asistencia al mayor número posible de actividades.

public static Set<String> plannerActivityes(List<Activity> listActivities)