UNIVERSIDAD DE MÁLAGA | uma.es

Examen enero 2025: Grafos

Estructuras de Datos

Descarga del Proyecto y Envío de Soluciones

1. **Descarga**:
    - Obtén los archivos del proyecto en Campus Virtual.
    - Descomprime el archivo descargado.
2. **Configuración**:
    - Abrir el proyecto `datastructures.bipartite.project` en IntelliJ IDEA.
3. **Desarrollo**:
    - Escribe **toda** tu solución en el archivo `BipartiteExam.java` que se encuentra en el paquete `org.uma.ed.exam`.
    - Incluye tu nombre y número de identificación (DNI para estudiantes españoles, pasaporte para estudiantes extranjeros) en los comentarios al inicio del archivo (reemplazar [Su Nombre] con tu nombre real y [Su DNI o Número de Pasaporte] con tu número de identificación real).
4. **Entrega**:
    - Cuando hayas completado el ejercicio, sube **SÓLO** el archivo `BipartiteExam.java` al Campus Virtual.
    - Enviar el archivo **SIN COMPRIMIR** (no uses zip ni otra técnica de compresión).
    - Utiliza para ello la tarea designada para la entrega de código disponible en Campus Virtual.
5. **Notas Importantes**:
    - Solo se calificarán archivos que **COMPILEN SIN ERRORES**. Si tu código no compila, recibirás una calificación de cero.
    - Si tu código no compila, coméntalo y déjalo sin definir, lanzando una `UnsupportedOperationException` como estaba inicialmente definido y luego asegúrate de que compile sin errores.
    - Asegúrate de que tu código cumple con las especificaciones y requisitos del problema.

Puedes realizar algunas pruebas ejecutando el método `main` en el archivo `BipartiteExam.java`.

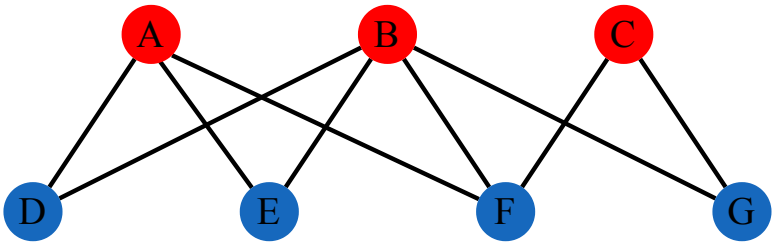
Te recuerdo que aunque pase todas las pruebas, revisa que tiene sentido lo que entregas, puedes suspender aunque las pases todas.

> Testing shows the presence, not the absence of bugs.
>
> — Edsger W. Dijkstra

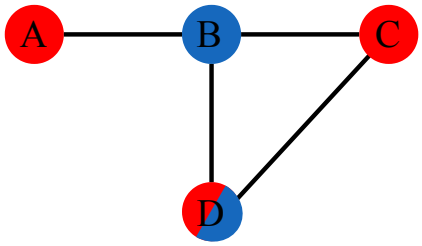
Grafos Bipartitos

Un grafo no dirigido se llama **bipartito** (o **2-coloreable**) si sus vértices pueden colorearse usando solo dos colores (por ejemplo, Rojo y Azul) de tal manera que dos vértices adyacentes siempre tengan colores diferentes. Un **coloreado compatible** es una asignación de colores a todos los vértices que cumple esta condición. Un grafo vacío (sin vértices) se considera bipartito.

Por ejemplo, el grafo en la parte izquierda de la figura es bipartito, ya que sus vértices pueden colorearse con dos colores diferentes, de modo que dos vértices adyacentes siempre tengan colores distintos. En contraste, el grafo a la derecha no es bipartito. En este grafo, el vértice D debería ser rojo por ser adyacente a un vértice azul (B) y, al mismo tiempo, debería ser azul por ser adyacente a un vértice rojo (C), lo cual es contradictorio.



Bipartite Graph



Non-Bipartite Graph

El objetivo de este ejercicio es implementar en Java un algoritmo para comprobar si un grafo conexo es bipartito, devolviendo una asignación de colores compatible a los vértices si es posible. El algoritmo a usar es una variante del recorrido en profundidad de un grafo. Utilizaremos una pila (stack) de pares para los vértices pendientes de visitar y colorear. Cada par en la pila indica un vértice y el color que debe tener, determinado como el opuesto a uno de sus adyacentes ya visitado. También usaremos un diccionario que asocia a cada vértice visitado el color asignado por el algoritmo.

Inicialmente, el diccionario estará vacío. Comenzamos inicializando una variable booleana (`bipartite`) a `true`. Esta variable indicará al final del algoritmo si el grafo es bipartito o no. También elegimos cualquiera de los vértices del grafo (lo llamaremos `source`) y le asignamos un color arbitrario (por ejemplo, Rojo), introduciendo en la pila el par con vértice `source` y color Rojo.

A continuación, se repetirá el siguiente bucle mientras sea necesario:

- Mientras que el grafo sea bipartito y la pila no esté vacía:
  - Desapilamos un par formado por un vértice (`v`) y el color que deberíamos asignarle (`c`).
  - Si el vértice (`v`) no ha sido visitado (no aparece en el diccionario):
    - Le asignamos el color (`c`) y lo visitamos, introduciendo en el diccionario la asociación `v` → `c` (clave `v`, valor `c`).
    - Apilamos en la pila pares para todos los vértices adyacentes de `v`, junto con el color que deberán tener (el opuesto a `c`).
  - En otro caso, si el vértice (`v`) ya fue visitado previamente y su color asignado en el diccionario es `cd`:
    - Si el color asociado en el diccionario (`cd`) no es igual al color en el par (`c`), hemos llegado a una contradicción y, por tanto, el grafo no es bipartito.
    - Si el color asociado en el diccionario (`cd`) y el color en el par (`c`) son iguales, ambas asignaciones de colores son compatibles y, por tanto, continuamos con la siguiente iteración del bucle.

- Al finalizar el bucle, si el grafo es bipartito, el diccionario contendrá un coloreado compatible.

El siguiente tipo enumerado puede ser usado para representar los colores rojo (`Color.Red`) y azul (`Color.Blue`):

```
public enum Color { Red, Blue;
  Color opposite() {
    return (this == Red) ? Blue : Red;
  }
}
```

El método `opposite` puede ser aplicado a un `Color` para obtener el color opuesto.

El siguiente `record` puede ser usado para representar los pares almacenados en la pila, formados por un vértice y un color:

```
private record Pair<V>(V vertex, Color color) {
  static <V> Pair<V> of(V vertex, Color color) { // factory method
    return new Pair<>(vertex, color);
  }
}
```

La clase `BipartiteExam` es la que se encargará de implementar el algoritmo:

```
public class BipartiteExam<V> {
  private boolean bipartite;
  private final Dictionary<V, Color> assignedColor;

  static <V> BipartiteExam<V> of(Graph<V> graph) {
    // factory method
    return new BipartiteExam<>(graph);
  }

  public BipartiteExam(Graph<V> graph) {
    // the algorithm must be implemented here
    throw new UnsupportedOperationException("Not implemented yet");
  }
  public boolean isBipartite() {
    // returns true if the graph is bipartite
    return bipartite;
  }

  public Dictionary<V, Color> assignedColor() {
    // returns the assigned colors if the graph is bipartite
    return bipartite ? assignedColor : null;
  }
}
```

Un programa ejemplo para probar la clase es:

```
class Test {
  public static void main(String[] args) {
    Graph<Integer> graph = DictionaryGraph.empty();
    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);
    graph.addVertex(4);
    graph.addVertex(5);
    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(2, 5);

    BipartiteExam<Integer> bipartiteExam = BipartiteExam.of(graph);
    if (bipartiteExam.isBipartite()) {
      System.out.println("Graph is bipartite");
      System.out.println("Colors assigned to vertices: " +
bipartiteExam.assignedColor());
    } else {
      System.out.println("Graph is not bipartite");
    }
  }
}
```

Completa la clase `BipartiteExam` para implementar el algoritmo descrito. Puedes usar las siguientes clases e interfaces que se proporcionan ya implementados en el proyecto:

- La interfaz `Graph<V>` que representa un grafo no dirigido y la clase `DictionaryGraph` que implementa esta interfaz.
- La interfaz `Dictionary<K, V>` que representa un diccionario y la clase `JDKHashDictionary` que implementa esta interfaz.
- La interfaz `Stack<T>` que representa una pila y la clase `JDKStack` que implementa esta interfaz.
