Algoritmos y Estructuras de Datos

Grafos

Guillermo Román Díez
groman@fi.upm.es

Lars-Åke Fredlund
lfredlund@fi.upm.es

Universidad Politécnica de Madrid

Curso 2022/2023

Motivación

- Un grafo es una forma de representar las relaciones que existen entre pares de objetos

Grafo

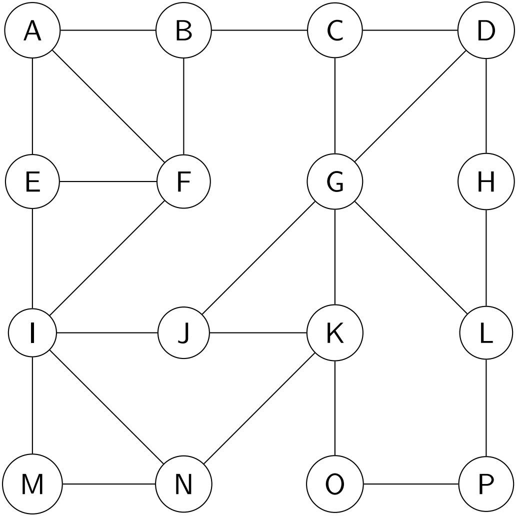
" *Un grafo es un conjunto de objetos, llamados vértices (vertices), y una colección de aristas (edges), donde cada arista conecta dos vértices*"

- Podemos verlo como un conjunto de vértices $V = \{u, v, w, x \dots\}$ y una colección de aristas $E = [(u, v), (u, x), \dots]$
- Los grafos son de aplicación en mútiples dominios: mapas, transporte, instalaciones eléctricas, redes de computadores, conexiones en redes sociales, ...

Tipos de Grafos

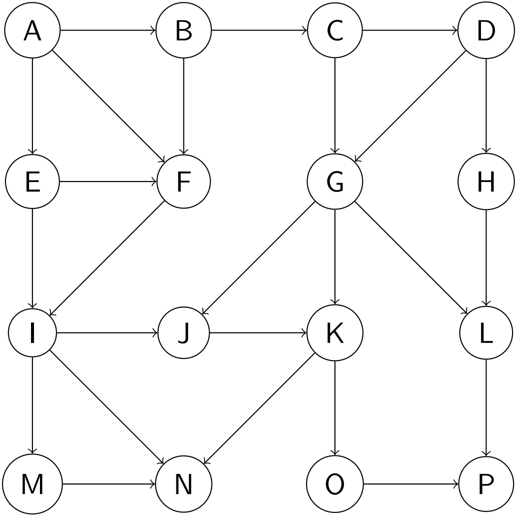
- Las aristas que conectan los vértices (o nodos) de un grafo pueden ser de dos tipos
- **Aristas no dirigidas**: Decimos que una arista es no dirigida cuando el par (u, v) no está ordenado
  - La arista te lleva de u a v y de v a u
  - El par (u, v) sería lo mismo que el par (v, u)
- **Aristas dirigidas**: Decimos que una arista es dirigida cuando el par (u, v) está ordenado
  - La arista únicamente te lleva de u a v, pero no de v a u
- Si todas las arista de un grafo son aristas no dirigidas, decimos que el grafo es no dirigido
- Si hay alguna arista dirigida, el grafo es un grafo dirigido

Ejemplos de grafos



Grafo no dirigido

Ejemplos de grafos

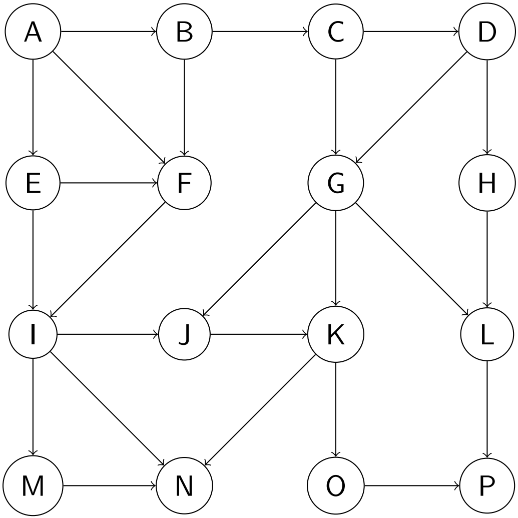


Grafo dirigido

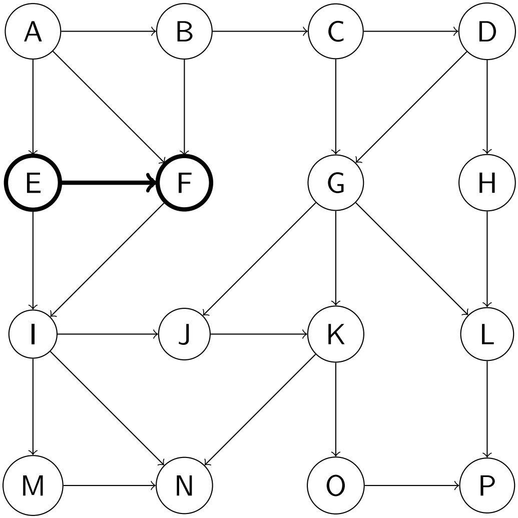
Definiciones

- Dos vértices son **adyacentes (adjacent)** si hay una arista que los conecta
- El **origen** y **destino** son los vértices inicial y final de una arista dirigida
- Un nodo puede tener **aristas salientes (outgoing edges)** que tienen como origen el nodo y **aristas entrantes (incoming edges)**, que tienen el nodo como destino
- El **grado** de un nodo es el número de aristas que entran y salen del nodo
  - Podemos distinguir entre el grado *entrante* y el grado *saliente*
- Un **camino (path)** es una sencuencia de vértices y aristas que empieza en y acaba en un vértice, de forma que cada arista del camino es adyacente con su vértice anterior y su vértice siguiente del camino
- Un **ciclo (cycle)** es un camino cuyo primer y último nodo son el mismo
- Un **camino simple** es un camino que no repite vértices (no contiene ciclos)
- Un **bosque** es un grafo sin ciclos

Ejemplos de grafos

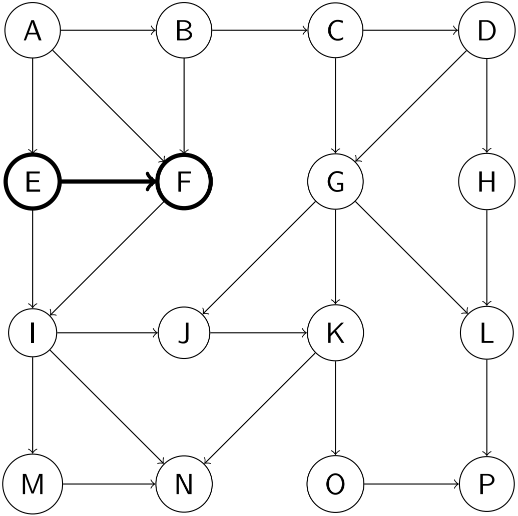


Ejemplos de grafos



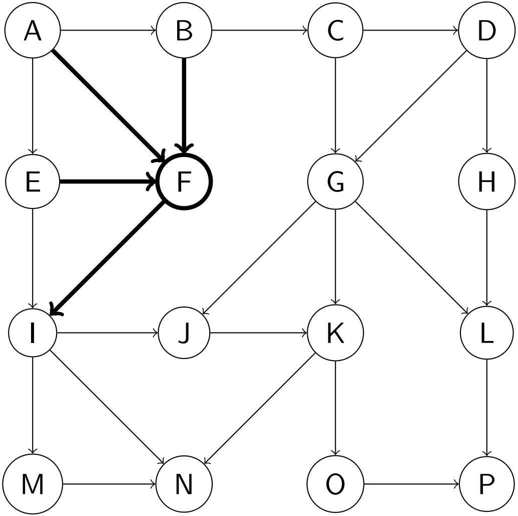
Los vértices E y F son adyacentes

Ejemplos de grafos



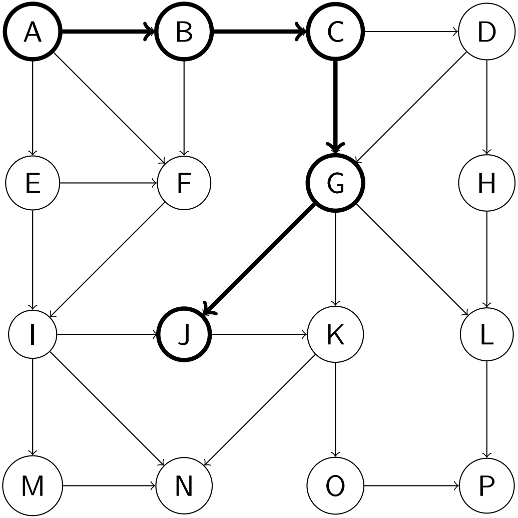
E es el vértice origen y F el vértice destino

Ejemplos de grafos



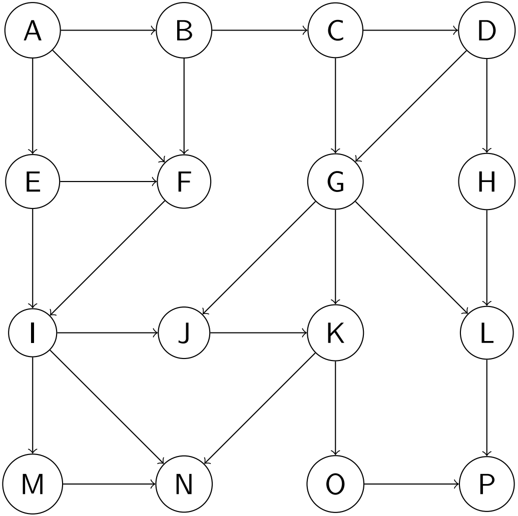
El grado F es 4, el grado entrante 3 y el grado saliente 1

Ejemplos de grafos



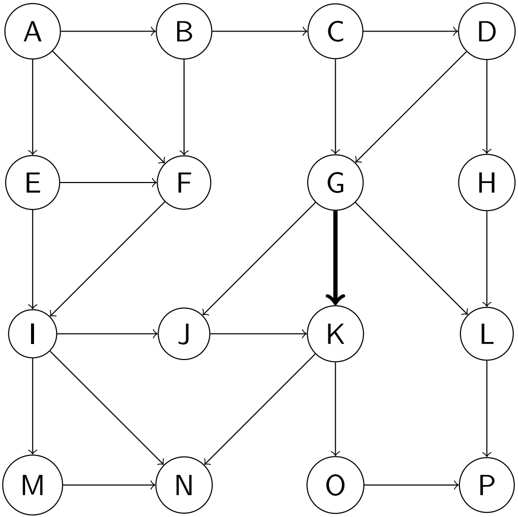
[A, (A,B), B, (B,C), C, (C,G), G, (G,J), J] es un camino (simple)

Ejemplos de grafos



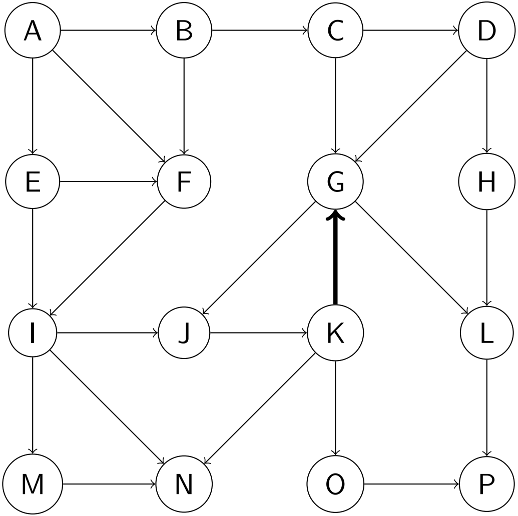
Este grafo es un bosque ya que NO tiene ciclos

Ejemplos de grafos



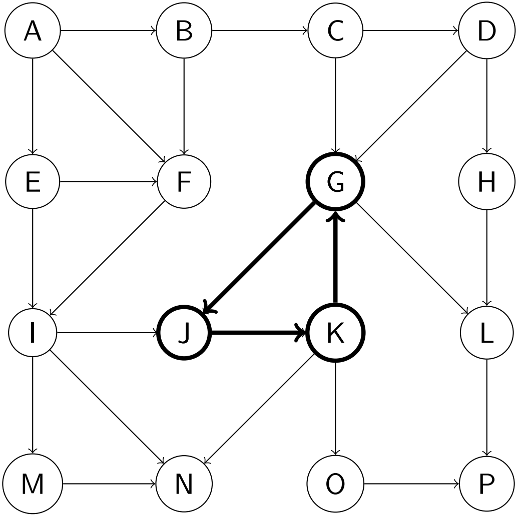
Modificando ligeramente el grafo hacemos un ciclo

Ejemplos de grafos



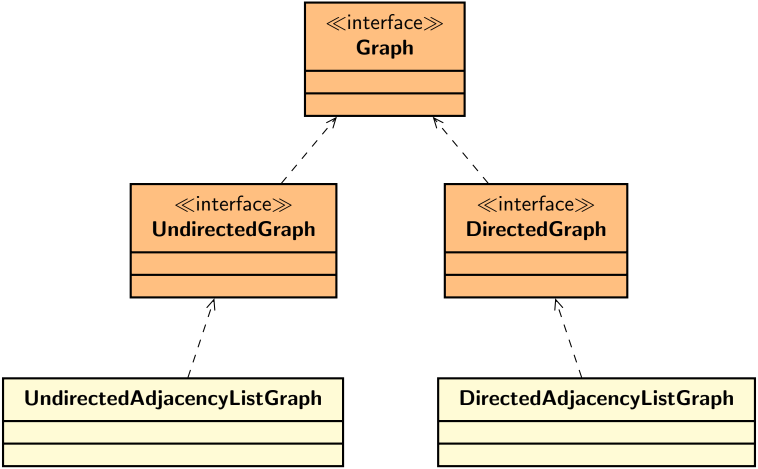
Modificando ligeramente el grafo hacemos un ciclo

Ejemplos de grafos

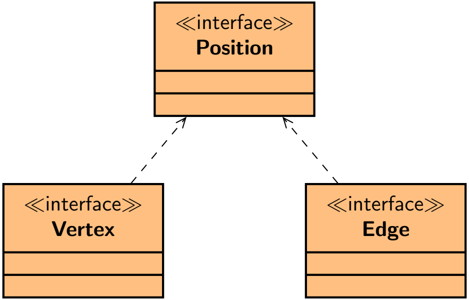


[G, (G,J), J, (J,K), K, (K,G), G] es un ciclo

Jerarquía de clases e interfaces en aedlib



Jerarquía de clases e interfaces en aedlib



Interfaz Graph<V,E>

```
public interface Graph<V, E> {
  public int size();
  public boolean isEmpty();
  public int numVertices();
  public int numEdges();

  public int degree(Vertex<V> v) throws IAE;

  public Iterable<Vertex<V>> vertices();
  public Iterable<Edge<E>> edges();

  public V set(Vertex<V> p, V o) throws IAE[1];
  public E set(Edge<E> p, E o) throws IAE;

  public Vertex<V> insertVertex(V o);

  public V removeVertex(Vertex<V> v) throws IAE;
  public E removeEdge(Edge<E> e) throws IAE;
}
```

[1] IllegalArgumentException

Interfaz `Graph<V,E>`

- Un grafo dispone de dos genéricos `<V,E>` para almacenar información en los vértices (`V`) y en las aristas (`E`)
- Los métodos `size`, `isEmpty`, `numVertices` y `numEdges` permiten consultar el número elementos del grafo
- `degree` devuelve el número de aristas incidentes en un vértice
- `vertices` y `edges` permiten recorrer los vértices y las aristas que componen el grafo mediante un `Iterable`
- `insertVertex` permite insertar un vértice. La inserción de las aristas se deja para las clases especializadas para grafos dirigidos y no dirigidos
- `removeVertex` borran un vértice y las aristas que llegan y salen de él
- `removeEdge` borra una arista, pero no los vértices que la formaban
- Todos los métodos que reciben un vértice o una arista como parámetro pueden lanzar `IllegalArgumentException`

Interfaz UndirectedGraph<V,E>

```
public interface UndirectedGraph<V,E> extends Graph<V,E> {
  public Iterable<Vertex<V>> endVertices(Edge<E> e) throws
       IAE²;

  public Edge<E> insertUndirectedEdge(Vertex<V> u,
                                      Vertex<V> v, E o)
        throws IAE;

  public Vertex<V> opposite(Vertex<V> v, Edge<E> e)
        throws IAE;

  public boolean areAdjacent(Vertex<V> u, Vertex<V> v)
        throws IAE;

  public Iterable<Edge<E>> edges(Vertex<V> v) throws IAE;
}
```

[2] IllegalArgumentException

Interfaz `UndirectedGraph<V,E>`

- `UndirectedGraph<V,E>` define el interfaz de un grafo no dirigido
- `insertUndirectedEdge` permite crear aristas conectando los nodos `u` y `v` y asociar a la arista un objeto
- Dado un nodo y una arista, el método `opposite` devuelve el nodo que está "al otro lado de la arista"
- El método `areAdjacent` permite saber si dos nodos están conectados mediante alguna arista (son adyacentes)
- `edges` (sobrecargado) recibe un vértice y devuelve todas las aristas incidentes en él
- Todos los métodos que reciben un vértice o una arista como parámetro pueden lanzar `IllegalArgumentException`

Interfaz DirectedGraph<V,E>

```
public interface DirectedGraph<V,E> extends Graph<V,E> {

  public Vertex<V> startVertex(Edge<E> e) throws IAE;

  public Vertex<V> endVertex(Edge<E> e) throws IAE;

  public Edge<E> insertDirectedEdge(Vertex<V> from,
                                   Vertex<V> to, E o) throws IAE;

  public Iterable<Edge<E>> outgoingEdges(Vertex<V> v)
       throws IAE;

  public Iterable<Edge<E>> incomingEdges(Vertex<V> v)
       throws IAE;

  public int inDegree(Vertex<V> v) throws IAE;

  public int outDegree(Vertex<V> v) throws IAE;
}
```

Interfaz DirectedGraph<V,E>

- `DirectedGraph<V,E>` define el interfaz de un grafo dirigido
- `insertDirectedEdge` permite crear aristas dirigidas conectando los nodos `from` y `to` y asociar a la arista un objeto
- Dada una arista, los métodos `startVertex` y `endVertex` permiten obtener los nodos participantes en una arista
- Los métodos `outgoingEdges` y `incomingEdges` permiten obtener las aristas entrantes y salientes de un vértice
- `inDegree`, `outDegree` devuelven el número de entrantes o salientes de un vértice
- Todos los métodos que reciben un vértice o una arista como parámetro pueden lanzar `IllegalArgumentException`

Problemas para que se usa grafos

Hay muchos problemas conocidos para los que se usan grafos, por ejemplo: "travelling salesman":

- "Dado un mapa (un grafo) con ciudades, las distancias entre ellos y una ciudad inicial, devuelve **la ruta mas corta** que visita todas las ciudades y vuelve al ciudad inicial.

Ejemplo Travelling Salesman

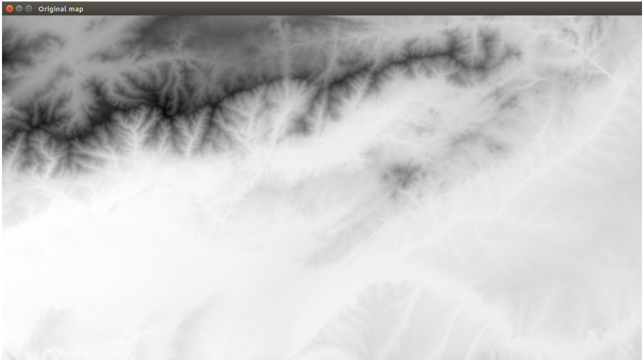
- Rutas para visitar los capitales de España:

Ejemplo Travelling Salesman

- Una ruta posible:

Encontrando Caminos Óptimos en Grafos

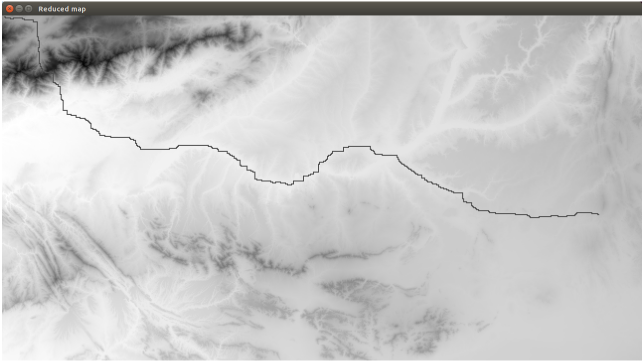
Habitualmente queremos encontrar un camino "óptimo" entre dos puntos en un mapa (grafo) – por ejemplo en videojuegos o en un GPS

- 

- El color indica la altura
- Dimensiones (número de puntos): 722x1288 = 929 936 puntos

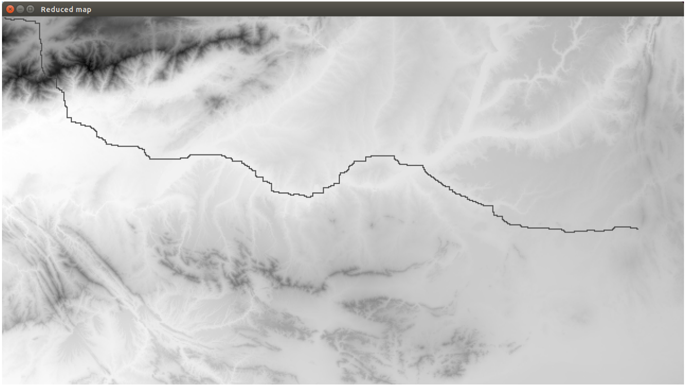
Encontrando Caminos Óptimos en Grafos

Habitualmente queremos encontrar un camino "óptimo" entre dos puntos en un mapa (grafo) – por ejemplo en videojuegos o en un GPS

- 

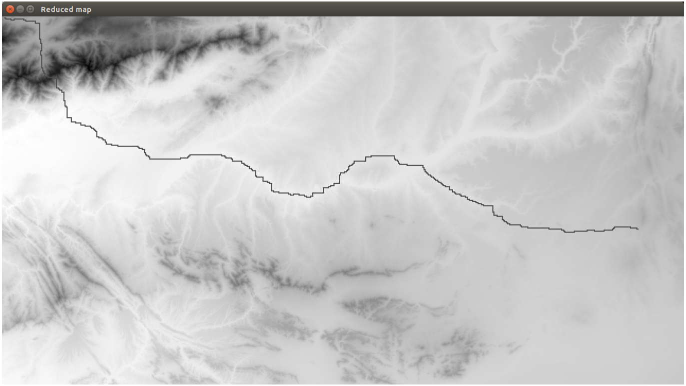
- El color indica la altura
- Dimensiones (número de puntos): 722x1288 = 929 936 puntos
- ¿Cómo podemos encontrar el camino "óptimo", con menos variaciones en altitud y menor distancia?

Encontrando Caminos Optimos en Grafos



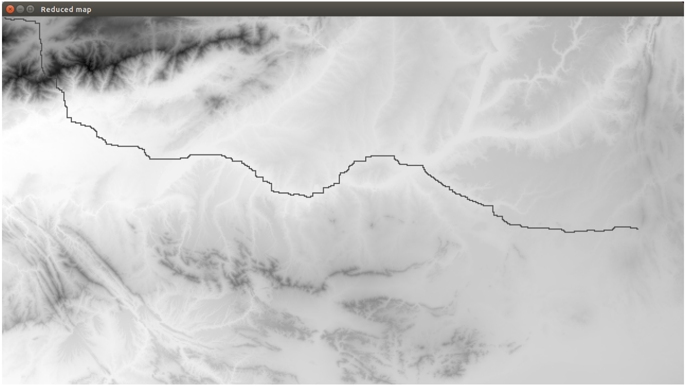
- ¿Podemos enumerar todos los caminos?

Encontrando Caminos Optimos en Grafos



- ¿Podemos enumerar todos los caminos?
- **No. Son demasiados. Necesitamos un algoritmo mejor.**

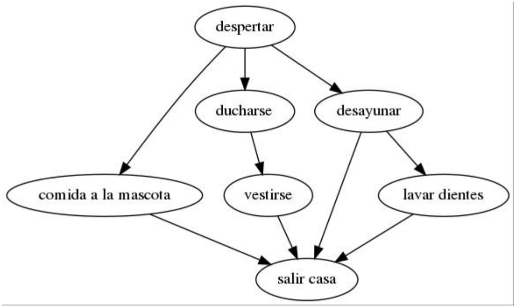
Encontrando Caminos Optimos en Grafos



- ¿Podemos enumerar todos los caminos?
- **No. Son demasiados. Necesitamos un algoritmo mejor.**
- Podemos usar "Dijkstra's shortest path algorithm"

Decidir en que orden realizar tareas: “topological sort”

- Tenemos que realiar las siguientes tareas cada mañana:



- Nota que el grafo es dirigido, sin ciclos – un **“directed acyclic graph” (dag)**.
- ¿En que orden podemos realizar estas tareas, cumpliendo las dependencias: $t_1, \ldots, t_n$? (un “topological ordering”)
- Usaremos un “topological sort” algoritmo.

Algoritmo "Topological sort"

1. *Dado un grafo (dag) g*:
2. *Crea estructuras de datos*:
   - un conjunto s
   - un mapa $incounter : vertice \mapsto \mathcal{N}$
   - una lista *result*
3. *Inicialización*: para cada vertice $v \in g$:
   - añade $\langle v, g.indegree(v)\rangle$ a *incounter*
   - si $indegree(v) == 0$ añade v a s
4. *Bucle – repite hasta que s es vacio*:

   Saca un vertice v de s, y:
   - añade v al final de *result*
   - calcula el conjunto de vertices $dest(v)$ que son los destinos de una arista originando en v
   - para cada vertice $v' \in dest(v)$:
     - ★ cambia $incounter(v') = incounter(v') - 1$
     - ★ si $incounter(v') = 0$ añade v' a s
5. *Final*: *result* contiene los vertices de g en orden "topologico"