

# Topological Sort y Componentes Fuertemente Conexas

Melanie Sclar

Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires

Problemas, Algoritmos y Programación

# Contenidos

- 1 Topological Sort
  - Motivación
  - Topological Sort con in-degree
  - Topological Sort con DFS
  - Problemas
- 2 Componentes fuertemente conexas
  - Motivación
  - Algoritmo de Kosaraju
  - Problemas

# Grafos hasta para vestirse

Muchas veces **tenemos un orden parcial que nos gustaría extender a un orden total**. Por ejemplo si construyéramos un grafo de precedencia para vestirnos, sabemos que:

medias  $\rightarrow$  zapatillas

ropa interior  $\rightarrow$  pantalón

pantalón  $\rightarrow$  zapatillas

Pero esto aún nos deja libertad respecto del orden total final, pues tenemos estos dos órdenes totales:

medias  $\rightarrow$  ropa interior  $\rightarrow$  pantalón  $\rightarrow$  zapatillas

ropa interior  $\rightarrow$  medias  $\rightarrow$  pantalón  $\rightarrow$  zapatillas

Este no es el único ejemplo: siempre que tengamos muchas tareas a realizar con algunas precedencias preestablecidas y debemos definir el orden total en el que las mismas se realizarán (no podemos paralelizarlas), será útil lo que veremos hoy.

## Topological Sort

Un *topological sort* de un DAG (Directed Acyclic Graph)  $G$  es un ordenamiento de todos sus nodos de manera tal que si  $G$  contiene un eje  $(u, v)$ , entonces  $u$  aparece antes que  $v$  en el ordenamiento.

# Contenidos

## 1 Topological Sort

- Motivación
- Topological Sort con in-degree
- Topological Sort con DFS
- Problemas

## 2 Componentes fuertemente conexas

- Motivación
- Algoritmo de Kosaraju
- Problemas

# Topological sort con in-degree

Existen varias formas de resolver este problema. La que más me gusta observa que cada nodo tiene un *in - degree* particular.

## in-degree de un nodo

Se define el *in - degree* de un nodo  $v \in V(G)$  como la cantidad de ejes incidentes en  $v$  en el grafo  $G$ . Lo notamos  $in[v]$ .

- Cuando  $in[v] = 0$ ,  $v$  no tiene ninguna dependencia que le impida ir en este momento en el ordenamiento total.
- Al agregar  $v$  al ordenamiento total, esto modifica el *in - degree* de todos los  $w$  tal que existe  $(v, w) \in E(G)$ .

# Topological sort con in-degree

- Con estas observaciones, podemos ver que si en cada momento elegimos un nodo con *indegree* igual a 0 y lo colocamos como próximo nodo del ordenamiento, cuando hagamos esto sobre todos los nodos tendremos un topological sort. Es importante recordar actualizar el *indegree* de los vecinos.
- Cuando hay varios candidatos posibles, podemos elegir cualquiera de ellos. En general, todos los candidatos se guardan en una cola, pero esto dependerá del problema que estemos resolviendo.
- Si en algún momento no tenemos más nodos con *indegree* = 0 pero aún no colocamos todos los nodos, significa que el grafo no era un DAG.

# Pseudocódigo del toposort con in-degree

```
1  L = Lista vacia que contendra los elementos ordenados
2  S = Conjunto de nodos sin nodos entrantes (in-degree = 0)
3
4  mientras S no sea vacio :
5      removemos un nodo v de S (cualquiera sirve)
6      agregamos v al final de L
7      para cada nodo w tal que existe el eje (v, w) :
8          removemos (v, w) del grafo ( $\text{indegree}[w] = \text{indegree}[w] - 1$ )
9          si no hay mas ejes incidentes en w ( $\text{indegree}[w] = 0$ ) :
10             insertar w en S
11
12 si el grafo aun tiene ejes retorno error (no era DAG)
13 si no, retorno L
```

Se puede ver que la complejidad será  $O(n + m)$ .



# Contenidos

## 1 Topological Sort

- Motivación
- Topological Sort con in-degree
- **Topological Sort con DFS**
- Problemas

## 2 Componentes fuertemente conexas

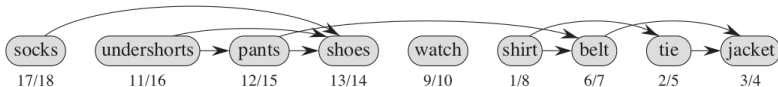
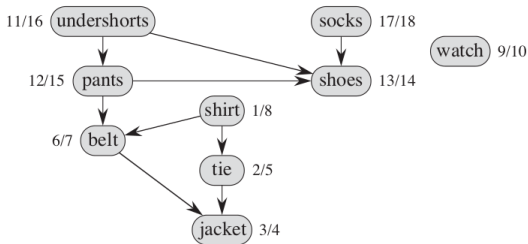
- Motivación
- Algoritmo de Kosaraju
- Problemas

# Topological Sort con DFS

Utilizando la notación de la clase pasada, recorreremos el grafo con DFS colocando dos números en cada nodo: el tiempo de descubrimiento y el de finalización.

A medida que vayamos finalizando cada nodo, lo iremos agregando al frente de la lista del orden topológico. Así, el que termina último deberá ir primero en el orden final (lo que es intuitivo, porque de él dependen muchos de los otros nodos).

# Ejemplo de Topological Sort con DFS



Ejemplo totalmente no robado del Cormen

# Pseudocódigo de Topological Sort con DFS

1. Correr DFS sobre  $G$  para computar los tiempos de finalización de cada nodo
2. A medida que cada vértice sea finalizado, lo inserto en el frente de la lista de respuesta
3. Devuelvo la lista, que representa un topological sort de  $G$

La complejidad será  $O(n + m)$ , pues sólo consiste en correr un DFS.

# Correctitud de Topological Sort con DFS

## Un DAG no contiene back edges

Un grafo  $G$  dirigido es acíclico si y sólo si un DFS de  $G$  no tiene *back edges*. Recordemos que una back edge es un eje  $(u, v)$  que conecta a  $u$  con  $v$ , un antecesor de  $u$  en el árbol de DFS.

# Correctitud de Topological Sort con DFS

## Un DAG no contiene back edges

Un grafo  $G$  dirigido es acíclico si y sólo si un DFS de  $G$  no tiene *back edges*. Recordemos que una back edge es un eje  $(u, v)$  que conecta a  $u$  con  $v$ , un antecesor de  $u$  en el árbol de DFS.

Idea de la demo: si tuviera un back edge, ese formaría un ciclo.

# Correctitud de Topological Sort con DFS

## Un DAG no contiene back edges

Un grafo  $G$  dirigido es acíclico si y sólo si un DFS de  $G$  no tiene *back edges*. Recordemos que una back edge es un eje  $(u, v)$  que conecta a  $u$  con  $v$ , un antecesor de  $u$  en el árbol de DFS.

Idea de la demo: si tuviera un back edge, ese formaría un ciclo.

Debemos ver que para todo  $(u, v) \in E(G)$ , vale que  $v.f < u.f$ . Si vemos esto entonces podemos observar que al momento de agregar un nodo al frente de lista respuesta todos los ejes incidentes en él tienen finishing time posterior. Esto implica que los ejes estarán bien ordenados en el toposort.

## Correctitud de Topological Sort con DFS (2)

Por suerte, es fácil ver que para todo  $(u, v) \in E(G)$ , vale que  $v.f < u.f$ :

Si al momento de recorrer  $(u, v)$  ya tuviéramos a  $v$  marcado de gris (es decir, ya tiene tiempo inicial pero no final) entonces  $v$  sería ancestro de  $u$  y  $(u, v)$  sería un back edge. ¡Absurdo!

Entonces  $v$  es o bien blanco o negro (nunca tocado, o ya finalizado).

- Si  $v$  fuera blanco,  $v$  es descendiente de  $u$  y luego  $v.f < u.f$
- Si  $v$  fuera negro,  $v$  ya se terminó y por lo tanto  $v.f$  ya está definido. Pero  $u.f$  todavía no está definido, y luego sabemos que  $v.f < u.f$ .



# Contenidos

## 1 Topological Sort

- Motivación
- Topological Sort con in-degree
- Topological Sort con DFS
- Problemas

## 2 Componentes fuertemente conexas

- Motivación
- Algoritmo de Kosaraju
- Problemas

# Problema: conservación de la Mona Lisa

La Mona Lisa debe ser conservada: este trabajo será realizado en dos laboratorios especializados en distintas tareas. El proceso de conservación se ha dividido en varias etapas y para cada una de ellas sabemos en qué laboratorio deberá realizarse.

Transportar la tan famosa pintura introduce un riesgo adicional, por lo que debemos evitarlo siempre que sea posible. Idealmente, primero se realizarían todos los trabajos en el primer laboratorio para luego mover la pintura para el segundo laboratorio y completar las tareas.

Sin embargo, hay numerosas dependencias entre etapas de conservación - algunas deben completarse antes que otras puedan comenzar. Debemos encontrar un orden para realizar las tareas tal que se minimice el número de veces que la pintura necesita ser movida entre laboratorios. La conservación puede comenzar en cualquier laboratorio.

# tareas  $\leq 100000$ , # dependencias  $\leq 1000000$

[https://icpcarchive.ecs.baylor.edu/index.php?option=com\\_onlinejudge&Itemid=8&category=567&page=show\\_problem&problem=4275](https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=567&page=show_problem&problem=4275)

## Solución: conservación de la Mona Lisa

- Básicamente, queremos un topological sort de las tareas pero siempre que sea posible tomaremos una tarea del mismo laboratorio donde nos encontramos actualmente.
- Cuando esto no sea posible, ahí tomaremos una tarea del otro laboratorio, contabilizando este cambio.
- Como no sabemos en qué laboratorio comenzamos, probamos ambos posibles comienzos y tomamos el que menos cambios entre laboratorios produzca.

# Combinando Topological Sort con otras técnicas

- Muchas veces el topological sort (o *toposort*, para los amigos) no soluciona completamente el problema, pero es un paso necesario en el camino a la solución.
- Por ejemplo, a partir de llevar al grafo a un orden topológico podemos utilizar técnicas que requieren de la noción de *anterior*, como programación dinámica (entre muchas otras).

## Problema: camino máximo en un DAG

Dado un DAG con pesos, queremos hallar un camino máximo en él. Es decir, entre todos los caminos posibles, queremos hallar uno cuya suma de los ejes sea máxima.

Veamos un ejemplo en el pizarrón.

¿Ideas?

## Solución: camino máximo en un DAG

Este problema es esencialmente programación dinámica, pero no podríamos aplicarla sin dar un orden para los nodos. Sin un orden adecuado, ¿cuál sería el caso recursivo? ¿cómo sé que es más pequeño?

Tomo un topological sort de  $G$ , que denotaré  $ordenTopologico(G)$ . Si estoy en  $w$ , todos los ejes que llegan a  $w$  partieron de un nodo anterior según  $ordenTopologico(G)$ . Así, para cada uno de ellos puedo asumir que la distancia máxima ya está calculada y fijarme si al agregar el costo de la arista obtengo un nuevo camino máximo terminando en  $w$  o no.

# Solución: camino máximo en un DAG

```
1 | para cada vertice v en ordenTopologico(G) :  
2 |     para cada eje (v, w) en E(G) :  
3 |         distancia[w] = max(distancia[w], distancia[v] + peso (G, (v, w)))  
4 |  
5 | devolver maxima distancia[v] entre todos los v en V(G)
```

¿A qué problema clásico de programación dinámica les suena este problema?

# Contenidos

- 1 Topological Sort
  - Motivación
  - Topological Sort con in-degree
  - Topological Sort con DFS
  - Problemas
- 2 Componentes fuertemente conexas
  - **Motivación**
  - Algoritmo de Kosaraju
  - Problemas



# Componentes fuertemente conexas

En grafos no dirigidos ya vimos la noción de componentes conexas: son subgrafos conexos maximales, es decir, subgrafos conexos donde al agregar un nodo más (si existiera alguno) el subgrafo dejaría de ser conexo.

En el párrafo anterior, un grafo conexo es un grafo donde entre existe un camino entre cualquier par de nodos que lo forman.

¿Cómo podemos extender esto a grafos dirigidos? Aquí no siempre que desde  $A$  podemos llegar a  $B$  significa que desde  $B$  podemos llegar a  $A$ . Veamos un ejemplo en el pizarrón.

# Componentes fuertemente conexas

## Grafo fuertemente conexo

Un grafo fuertemente conexo es un grafo  $G$  dirigido en el cual para todo par de nodos  $v, w$  ( $v \neq w$ ) existe un camino que parte de  $v$  y llega a  $w$  y viceversa.

La mayoría de los grafos no son fuertemente conexos, pero podemos separar los nodos en subgrafos tal que cada uno lo sea.

# Ejemplo visual

Dibujar ejemplo en el pizarrón donde marco las componentes fuertemente conexas. Usá tu creatividad.

# Checkear si un grafo es fuertemente conexo

Primero resolvamos una versión más sencilla del problema que queremos resolver. Supongamos que sólo nos interesara saber si un grafo es fuertemente conexo o no. ¿Qué podemos hacer?

# Checkear si un grafo es fuertemente conexo

Primero resolvamos una versión más sencilla del problema que queremos resolver. Supongamos que sólo nos interesara saber si un grafo es fuertemente conexo o no. ¿Qué podemos hacer?

Podemos partir de un nodo cualquiera  $u$  y hacer DFS desde  $u$ . Si no pude llegar a todos los nodos, no era fuertemente conexo (¿por qué?). Si pude, entonces invierto la dirección de todos los ejes de  $G$  y hago DFS desde  $u$  nuevamente. Si de nuevo pude llegar a todos los nodos, significa que en el grafo original puedo llegar desde cualquier nodo a  $u$ .

$$u \rightsquigarrow v \text{ y } v \rightsquigarrow u \text{ para todo } v \in V(G)$$

Luego, es un grafo fuertemente conexo.

# Contenidos

- 1 Topological Sort
  - Motivación
  - Topological Sort con in-degree
  - Topological Sort con DFS
  - Problemas
- 2 Componentes fuertemente conexas
  - Motivación
  - Algoritmo de Kosaraju
  - Problemas

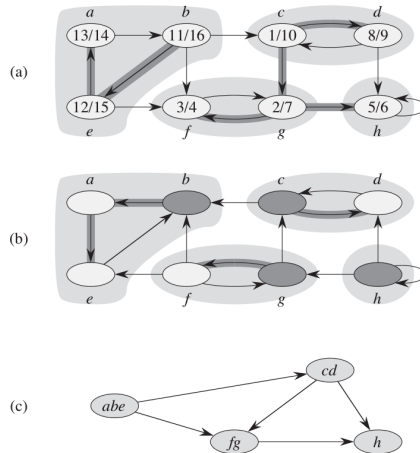
# Algoritmo de Kosaraju

1. Correr  $DFS(G)$  computando los tiempos de finalización  $u.f$  para cada nodo  $u \in V(G)$
2. Calcular  $G^T$
3. Correr  $DFS(G^T)$ , pero en el ciclo central del DFS consideramos los vértices en orden decreciente según el tiempo de finalización calculado en el ítem 1.
4. Imprimir los vértices de cada árbol del bosque de DFS formado en la línea 3 como una componente fuertemente conexa.

Corramos un ejemplo en el pizarrón para ganar intuición de por qué funciona el algoritmo. Luego, pensemos por qué funciona!

Nota: el ejemplo hecho a mano se adjunta en `kosaraju.jpg`





## Demo de Kosaraju: observaciones

- Una vez que separo el grafo en componentes fuertemente conexas, si pienso a cada componente como un nodo (les voy a decir “supernodos”), dicho grafo es un DAG. Esto es así porque si contuviera un ciclo, todos los supernodos formarían parte de una misma componente fuertemente conexa, lo que sería absurdo.
- $G$  y  $G^T$  tienen las mismas componentes fuertemente conexas ( $G^T$  es  $G$  transpuesto, o sea, todos los ejes invierten su dirección).

# Demo de Kosaraju

Veremos por inducción que cada árbol del  $DFS(G^T)$  se corresponde con una componente fuertemente conexa de  $G$  (o  $G^T$ , pues son las mismas). Supongamos que los primeros  $k$  árboles explorados en

$DFS(G^T)$  son componentes fuertemente conexas (SCC para abreviar). Veamos que el  $k + 1$ -ésimo lo será (el caso base con  $k = 0$  es trivial).

## Demo de Kosaraju

Como elijo las raíces del  $DFS(G^T)$  en orden descendiente de finishing time, sé que los tiempos de las raíces de los árboles todavía no visitados son menores al que voy a analizar ahora.

Más aún, definamos  $f(C)$  = máximo finishing time de la componente  $C$ .

Sea  $u$  es la raíz del árbol que estamos analizando, y  $u \in C$  ( $C$  es SCC). Entonces  $u.f = f(C) > f(C')$  para todo  $C'$  componente conexa aún no hallada (por como elegimos las raíces del DFS).

## Demo de Kosaraju (2)

### Lema que me van a creer

Sea  $G = (V, E)$ ,  $(u, v) \in E$ , y sean  $C, C'$  SCC's tal que  $u \in C$  y  $v \in C'$ . Entonces,  $f(C) > f(C')$ .

### Corolario del lema que me creyeron

Sea  $G = (V, E)$ ,  $(u, v) \in E^T$ , y sean  $C, C'$  SCC's tal que  $u \in C$  y  $v \in C'$ . Entonces,  $f(C) < f(C')$ .

## Demo de Kosaraju (3)

Así, todos los ejes que “salen” del árbol que estamos explorando van para componentes anteriores, que ya fueron analizados. ¿Por qué?

## Demo de Kosaraju (3)

Así, todos los ejes que “salen” del árbol que estamos explorando van para componentes anteriores, que ya fueron analizados. ¿Por qué?

¡Por el corolario anterior! Si hay un eje  $(u, v) \in E^T$  con  $u \in C$ , luego  $f(C) < f(C')$  siendo  $v \in C'$ . Pero ya comentamos que los árboles que vienen después tienen finishing times menores por como es nuestro algoritmo, luego los únicos que cumplen con  $f(C) < f(C')$  son los  $C'$  ANTERIORES a  $C$ .

# Demo de Kosaraju (4 - gran final)

## Conclusiones maravillosas

- Todos los nodos de  $C$  se visitan al explorar el árbol de DFS enraizado en  $u$  (si no, significa que en  $G$  no vale que  $v \rightsquigarrow u$ , una de las condiciones de SCC).
- En la exploración del árbol de DFS que comienza en  $u$ , no se visitan nodos que no pertenezcan a la SCC: todos los vértices que puedo alcanzar son o bien pertenecientes a  $C$  o son de componentes anteriores, que ya terminé de explorar por hipótesis inductiva (y son de otra SCC).

Entonces, como no faltan ni sobran nodos, el  $k + 1$ -ésimo árbol es una SCC de  $G$ !



# Contenidos

- 1 Topological Sort
  - Motivación
  - Topological Sort con in-degree
  - Topological Sort con DFS
  - Problemas
  
- 2 Componentes fuertemente conexas
  - Motivación
  - Algoritmo de Kosaraju
  - Problemas

# Componentes semiconexas

Un grafo dirigido  $G$  es semiconexo si para todo par de vértices  $u, v \in V(G)$  vale que o bien  $u \rightsquigarrow v$  o bien  $v \rightsquigarrow u$ . Dar un algoritmo eficiente para determinar si  $G$  es semiconexo o no.

## Solución: Componentes semiconexas

- Separo  $G$  en componentes fuertemente conexas. Lo que me queda es un DAG. ¿Cómo debe ser este DAG para que se cumpla la propiedad de semiconexión?

## Solución: Componentes semiconexas

- Separo  $G$  en componentes fuertemente conexas. Lo que me queda es un DAG. ¿Cómo debe ser este DAG para que se cumpla la propiedad de semiconexión?
- Básicamente, debe ser un DAG que tenga un único topological sort.

# Solución: Componentes semiconexas

- Separo  $G$  en componentes fuertemente conexas. Lo que me queda es un DAG. ¿Cómo debe ser este DAG para que se cumpla la propiedad de semiconexión?
- Básicamente, debe ser un DAG que tenga un único topological sort.
- O dicho de otra forma, un grafo palito con posibles aristas agregadas (pensar por qué).

## Idea para ver por qué el DAG debe ser único

- Si hubiera 2 o más nodos a los cuales sólo inciden ejes (no salen ejes desde ellos) entonces no será semiconexo (no tengo cómo deducir uno de estos nodos del otro).
- Si hubiera 0 nodos a los cuales sólo inciden ejes, no era un DAG sobre lo que partimos. Sabemos que lo es pues el grafo que queda luego de las componentes fuertemente conexas.
- Entonces hay exactamente un nodo al cual sólo le inciden ejes. Ese es claramente el último del topological sort. Lo quito y aplico el mismo principio recursivamente.

## Problema: 2-SAT con SCC

Queremos resolver 2-SAT utilizando componentes fuertemente conexas. Esto nos permitirá tener una solución lineal. Recordemos 2-SAT:

### 2-SAT

El problema de 2-SAT consiste en determinar si a una colección de variables con dos valores posibles y restricciones entre pares de variables se le puede asignar un valor satisfaciendo todas las restricciones.

Veamos un ejemplo.

Esta podría ser una posible entrada para el problema. Cada variable puede tener dos valores posibles, que de ahora en más llamaremos *True* y *False*, y queremos que toda la expresión sea verdadera.

$$\begin{aligned} & (x_0 \vee x_2) \wedge (x_0 \vee \neg x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge \\ & (x_2 \vee \neg x_4) \wedge (x_0 \vee \neg x_5) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_5) \wedge \\ & (x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6) \end{aligned}$$

Nos gustaría poder modelar las restricciones con un grafo. ¿Ideas?



## 2-SAT: modelado del grafo

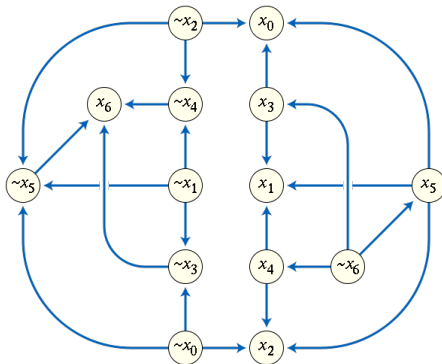
- Podemos pensar que un eje dirigido  $(u, v)$  significa que  $u$  implica  $v$ .
- Modelamos los nodos como cada variable y su negación: tendremos un nodo  $x_i$  y otro nodo  $\neg x_i$  para todo  $i$ .
- ¿Cómo deben ser los ejes que modelen las fórmulas binarias de la entrada?

## 2-SAT: modelado del grafo

- Podemos pensar que un eje dirigido  $(u, v)$  significa que  $u$  implica  $v$ .
- Modelamos los nodos como cada variable y su negación: tendremos un nodo  $x_i$  y otro nodo  $\neg x_i$  para todo  $i$ .
- ¿Cómo deben ser los ejes que modelen las fórmulas binarias de la entrada?

$$a \vee b \equiv \neg a \rightarrow b \equiv a \rightarrow \neg b$$

$$\begin{aligned}
 &(x_0 \vee x_2) \wedge (x_0 \vee \neg x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge \\
 &(x_2 \vee \neg x_4) \wedge (x_0 \vee \neg x_5) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_5) \wedge \\
 &(x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6)
 \end{aligned}$$



## 2-SAT: observaciones clave

- La fórmula es insatisfactible si y sólo si  $x_i$  implica  $\neg x_i$  y viceversa. ¿Por qué no alcanza sólo uno de los dos?
- Pensando en nuestro modelado con grafos, esto significa que la fórmula es insatisfactible si  $x_i$  y  $\neg x_i$  están en la misma componente fuertemente conexa (existe un camino de implicaciones que parte de uno y llega al otro para ambos).

## Solución 2-SAT (existencia)

- Así, corriendo Kosaraju detecto las componentes fuertemente conexas y chequeo que no haya ningún  $x_i$  y  $\neg x_i$  en la misma componente.
- Supongamos que no hay, entonces existe una asignación de valores a los  $x_i$  que resuelve el problema. ¿Pero cómo hallo dicha asignación?

## Solución 2-SAT (construcción de la respuesta)

Notemos que si una componente conexa contiene los elementos  $a_1, a_2, \dots, a_k$  entonces existe otra que contiene la negación de todos estos:  $\neg a_1, \neg a_2, \dots, \neg a_k$ .

Esto es así porque si  $a_1 \rightarrow a_2$  es equivalente a su contrarrecíproco  $\neg a_2 \rightarrow \neg a_1$  y el grafo de 2-SAT contiene todas las implicaciones entre elementos.

Entonces tenemos un DAG de las componentes fuertemente conexas: por ser DAG, qué podemos hacer con sus nodos?

## Solución 2-SAT (construcción de la respuesta)

Notemos que si una componente conexa contiene los elementos  $a_1, a_2, \dots, a_k$  entonces existe otra que contiene la negación de todos estos:  $\neg a_1, \neg a_2, \dots, \neg a_k$ .

Esto es así porque si  $a_1 \rightarrow a_2$  es equivalente a su contrarrecíproco  $\neg a_2 \rightarrow \neg a_1$  y el grafo de 2-SAT contiene todas las implicaciones entre elementos.

Entonces tenemos un DAG de las componentes fuertemente conexas: por ser DAG, qué podemos hacer con sus nodos? ¡Un topological sort!

- Notemos que podemos pensar que cada componente fuertemente conexa tiene una pareja, que es aquella con los mismos elementos pero negados.
- Si  $C_1$  y  $C_2$  son pareja, entonces  $u \in C_1 \rightsquigarrow \neg u \in C_2$  o viceversa, pues  $x \rightsquigarrow \neg x \vee \neg x \rightsquigarrow x$  siempre es cierto.
- Así, tomamos un topological sort y lo empezamos a analizar en orden inverso. Si la componente  $C_i$  no tenía valores asignados, coloco todos sus nodos en *True*. Como consecuencia, todos los de la pareja de  $C_i$  (que tiene todos los mismos elementos pero negados) irán en *False*.
- Notemos que esto no generará implicaciones del estilo  $True \rightarrow False$ , que son las que queremos evitar, pues tomamos el toposort en orden inverso.



# ¿Preguntas?