Grafos y ciclos

Clase 17

IIC 2133 - Sección 3

Prof. Eduardo Bustos

Sumario

Programación dinámica

A partir de este ejemplo, planteamos la estrategia de **programación dinámica** para resolver un problema

1. El número de subproblemas es (idealmente) polinomial
2. La solución al problema original puede calcularse a partir de subsoluciones
3. Hay un **orden natural** de los subproblemas (*del más pequeño al más grande*) y una recurrencia *sencilla* (★)
4. Recordamos las soluciones a subproblemas

La recurrencia es la clave para plantear el algoritmo base

Problema de dar vuelto

Consideremos ahora el problema de dar S pesos de vuelto usando el menor número posible de monedas

- Suponemos que los valores de las monedas, ordenados de mayor a menor, son $\{v_1, v_2 \ldots, v_n\}$
- Tenemos una cantidad ilimitada de monedas de cada valor

Posible estrategia codiciosa:

Asignar tantas monedas *grandes* como sea posible, antes de avanzar a la siguiente moneda *grande*

Ejemplo

Si $\{v_1, v_2, v_3, v_4\} = \{10, 5, 2, 1\}$, la estrategia codiciosa **siempre** produce el menor número de monedas para un vuelto S cualquiera

Problema de dar vuelto

Ejemplo

Sin embargo, la estrategia no funciona para un conjunto de valores cualquiera. Si $\{v_1, v_2, v_3\} = \{6, 4, 1\}$ y $S = 8$, entonces la estrategia produce

$$8 = 6 + 1 + 1$$

pero el óptimo es

$$8 = 4 + 4$$

Lo atacamos con programación dinámica

Problema de dar vuelto

Dado un conjunto de valores ordenados $\{v_1, \ldots, v_n\}$, definimos $z(S, n)$ como el problema de encontrar el menor número de monedas para totalizar S

Ejercicio

Proponga una recurrencia para resolver el problema $z(S, n)$ y plantee un algoritmo a partir de ella.

Problema de dar vuelto

Ejercicio

Sea $Z(S, n)$ la solución óptima al problema $z(S, n)$. Para buscar intuición, notamos que hay dos opciones respecto a las monedas de valor v_n

- Si se incluye una moneda de valor v_n,

$$Z(S, n) = Z(S - v_n, n) + 1$$

- Si no se usan monedas de valor v_n,

$$Z(S, n) = Z(S, n - 1)$$

Problema de dar vuelto

Ejercicio

Luego, generalizamos esta idea

- Para las monedas de de valor v_n,

$$Z(S, n) = \min\{Z(S - v_n, n) + 1,\ Z(S, n-1)\}$$

- Luego, generalizamos para el subconjunto de los primeros k valores de monedas

$$Z(T, k) = \min\{Z(T - v_k, n) + 1,\ Z(T, k-1)\}$$

donde $Z(T, 0) = +\infty$ si $T > 0$, y $Z(0, k) = 0$

Problema de dar vuelto

Ejercicio

Con esto, podemos plantear el siguiente algoritmo iterativo

$\texttt{Change}(S)$**:**

- **for** $T = 1, \ldots, S$ **:**
  - $Z[T][0] \leftarrow +\infty$
- **for** $k = 0, \ldots, n$ **:**
  - $Z[0][k] \leftarrow 0$
- **for** $k = 1, \ldots, n$ **:**
  - **for** $T = 1, \ldots, S$ **:**
    - $Z[T][k] \leftarrow Z[T][k-1]$
    - **if** $T - v_k \geq 0$ **:**
      - $Z[T][k] \leftarrow \min\{Z[T][k],\ Z[T - v_k, k]\}$

Un nuevo problema: proyectos con requisitos

Consideremos un proyecto complejo dividido en varias tareas

- Cada tarea es indivisible
- Algunas tareas tienen como requisito otras tareas
- No nos preocupamos del tiempo que toma cada tarea

Ejemplo

Un proyecto G tiene las siguientes tareas con su especificación de requisitos

- T0: no tiene requisitos
- T1: requiere T0
- T2: requiere T0
- T3: requiere T1 y T2

¿Cómo sabemos si el proyecto especificado es viable?

Viabilidad de proyectos con requisitos

Solo nos interesa la ejecución de tareas con requisitos

- El proyecto no será viable si hay una referencia **circular**

Para dos tareas A y B, si B tiene como requisito la tarea A entonces escribiremos

$$A \to B$$

Con esto, el proyecto es inviable si existe una secuencia de tareas

$$X \to T_1 \to \cdots \to T_n \to X$$

¿Es la única condición que debemos verificar para decidir?

Viabilidad de proyectos con requisitos

La ausencia de secuencias circulares es condición **necesaria y suficiente** para asegurar la viabilidad del proyecto

- Si hay una secuencia circular, no podemos **ordenar** las tareas
- Si no hay secuencia circular, ordenamos

El orden de las tareas nos indica en qué orden **ejecutarlas** para completar el proyecto

¿Cómo verificar la viabilidad de un proyecto de manera algorítmica?

Representación de la dependencia de tareas

Ejemplo

Un proyecto G tiene las siguientes tareas con su especificación de requisitos

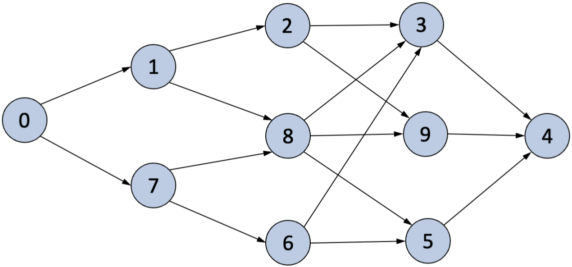
- T0: no tiene requisitos
- T1: requiere T0
- T2: requiere T1
- T3: requiere T1, T6, T8
- T4: requiere T3, T5 y T9
- T5: requiere T6 y T8
- T6: requiere T7
- T7: requiere T0
- T8: requiere T1 y T7
- T9: requiere T2 y T8

¿Cómo podemos representar de manera eficiente este escenario?

Representación de la dependencia de tareas

Ejemplo

- T0: no tiene requisitos
- T1: requiere T0
- T2: requiere T1
- T3: requiere T1, T6, T8
- T4: requiere T3, T5 y T9
- T5: requiere T6 y T8
- T6: requiere T7
- T7: requiere T0
- T8: requiere T1 y T7
- T9: requiere T2 y T8



Objetivos de la clase

- Comprender el concepto de grafo y sus uso para modelar
- Identificar un algoritmo para detección de ciclos
- Identificar estrategias para evitar loops en algoritmos en grafos

Sumario

Grafos

La representación anterior se conoce como **grafo dirigido**

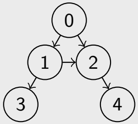
Definición

Un **grafo dirigido** es un par $G = (V, E)$, donde V es el conjunto de **nodos** y E es el conjunto de **aristas** de la forma (u, v), con $u, v \in V$

Ejemplo

$$V(G) = \{0, 1, 2, 3, 4\}$$

$$E(G) = \{(0, 1), (0, 2), (1, 2), (1, 3), (2, 4)\}$$



Usaremos grafos dirigidos cuando exista
un nodo de partida (origen) y uno de llegada (destino)

Grafos

De forma análoga definimos los grafos **no dirigidos**

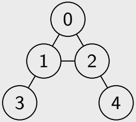
Definición

Un **grafo no dirigido** es un par $G = (V, E)$, donde V es el conjunto de **nodos** y E es el conjunto de **aristas** de la forma $\{u, v\}$, con $u, v \in V$ y $u \neq v$.

Ejemplo

$$V(G) = \{0, 1, 2, 3, 4\}$$

$$E(G) = \{\{0, 1\}, \{0, 2\}, \{1, 2\}, \{1, 3\}, \{2, 4\}\}$$



Usaremos grafos no dirigidos cuando importe agrupar pares de nodos vecinos

Grafos

Cuando hablemos de complejidad de algoritmo sobre grafos

- usaremos $|V|$ y $|E|$ como parámetros de tamaño
- los abreviaremos como V y E

Con esto, expresaremos la complejidad en términos de los elementos que definen el grafo

¿Es mejor un algoritmo $\mathcal{O}(V)$ o $\mathcal{O}(E)$?

Depende! Hay dos situaciones extremas

- $|E| = 0$
- Grafo completo (todas las aristas posibles)

Representación de grafos

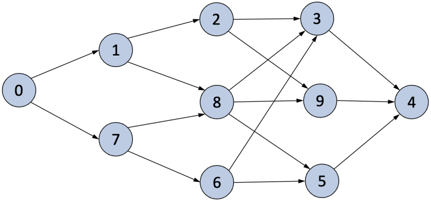
Necesitamos codificar los grafos y almacenarlos

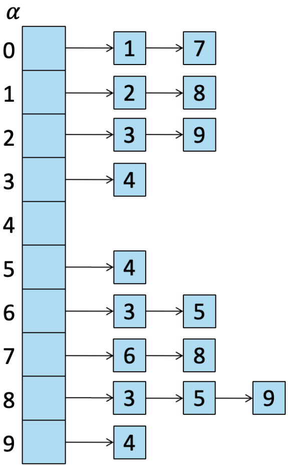
1. **Listas de adyacencias** en que la celda i de la lista tiene una lista con los vecinos j tales que (i, j) es arista en el grafo
2. **Matriz de adyacencias** en que la celda $[i][j]$ indica si la arista (i, j) existe en el grafo

Ambas almacenan la información de nodos en sus dimensiones, mientras que las aristas son los datos adicionales

La complejidad en grafos puede depender de qué representación se usa

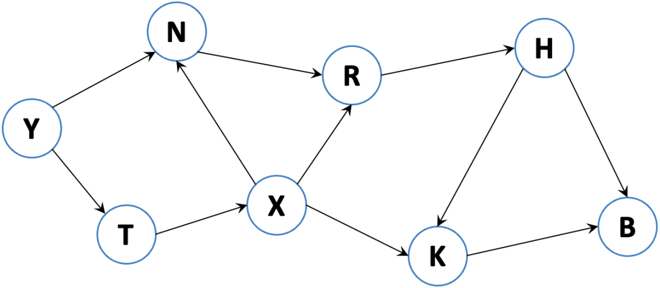
Representación de grafos





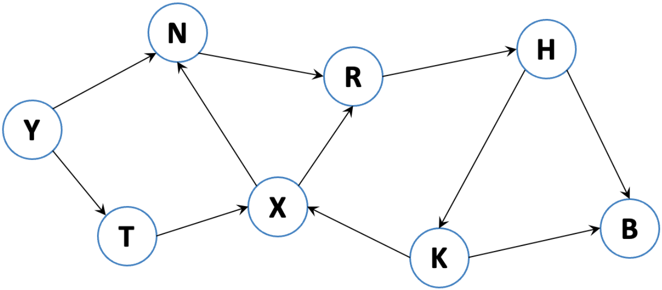
La complejidad en grafos puede depender de qué representación se usa

Volviendo al problema



¿Algún problema con este proyecto?

Volviendo al problema



¿Algún problema con este proyecto?

Ciclos

Definición

Dado un grafo dirigido $G = (V, E)$, un **camino** π es una secuencia de nodos $v_0, v_1, \ldots, v_n$ tal que

$$\{v_i, v_{i+1}\} \in E \qquad \text{para cada } i < n$$

Decimos que $\pi := v_0, v_1, \ldots, v_n$ es de largo n. Si $n > 0$ y $v_0 = v_n$ diremos que π es un **ciclo**. Un grafo que posee un ciclo se dice **cíclico**.

Para nuestro problema, determinar si hay una secuencia circular de tareas equivale a determinar si el grafo de representación contiene un ciclo

¿Podemos definir un algoritmo para detectar ciclos?

Sumario

Ciclos

Dadas dos tareas X e Y, diremos que Y es **posterior** a X si

- $X \rightarrow Y$ o
- existe una tarea Z tal que $X \rightarrow Z$ e Y es posterior a Z

En esencia: Y es posterior a X si X debe realizarse antes que Y

Esta propiedad es clave para definir un algoritmo de detección de ciclos

Ciclos

input : Grafo G, nodo $X \in V(G)$

$\texttt{Posteriors}(G, X)$:

$\quad P \leftarrow \varnothing$

$\quad$ **for** Y *tal que* $X \to Y$ **:**

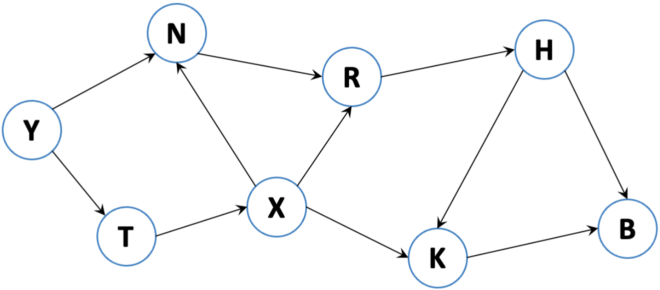
$\quad\quad P \leftarrow P \cup \{Y\}$

$\quad\quad P \leftarrow P \cup \texttt{Posteriors}(G, Y)$

$\quad$ **return** P

¿Qué complejidad tiene este algoritmo?

Ciclos

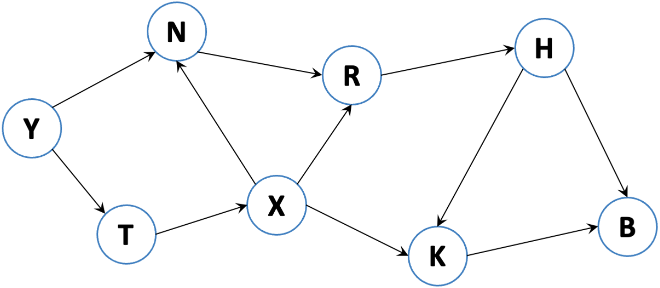


Ejercicio

Para el grafo anterior, determine los nodos posteriores a X usando el algoritmo `Posteriors`

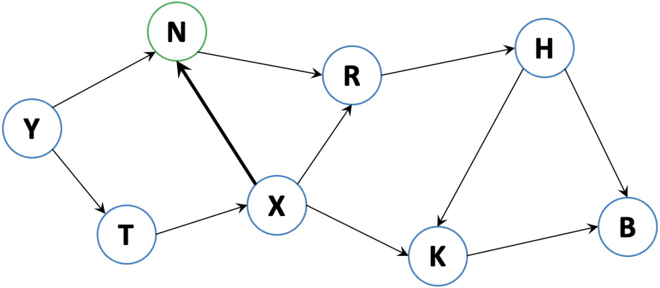
Ciclos

Exploramos cada arista de salida desde X



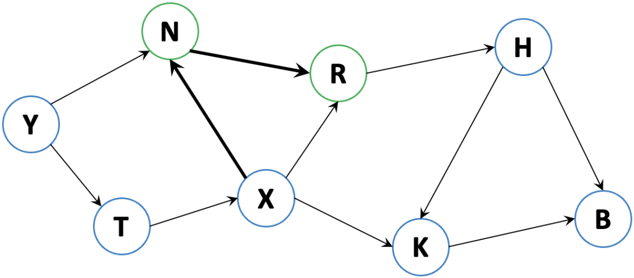
Ciclos

Encontramos primero a N



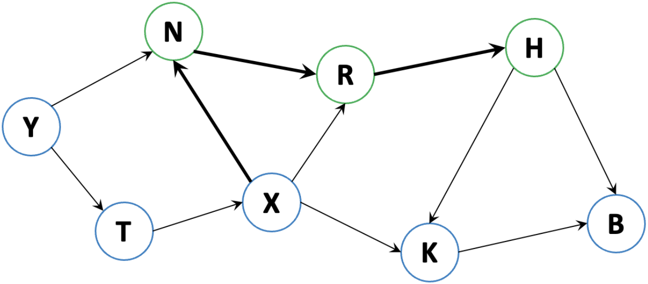
Ciclos

Ojo, ahora corresponde revisar las aristas de N recursivamente: encontramos a R

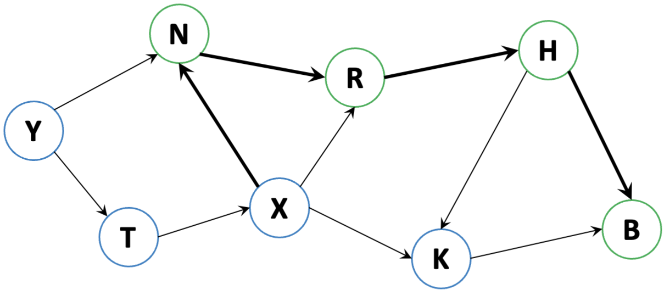


Ciclos

Seguimos, comenzando en R y encontrando H

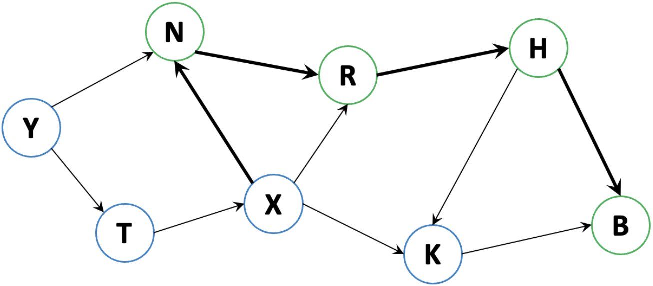


Seguimos, comenzando en H y encontrando B



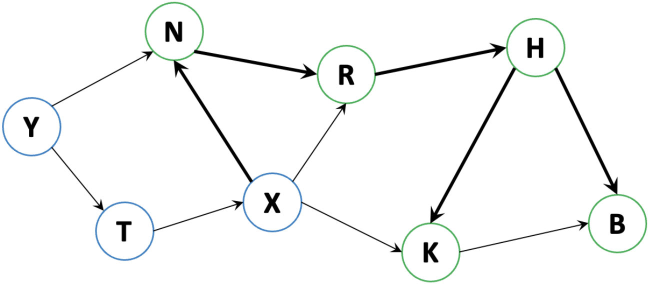
Ciclos

Comenzando en B no tenemos aristas de salida. **Retornamos** hacia atrás



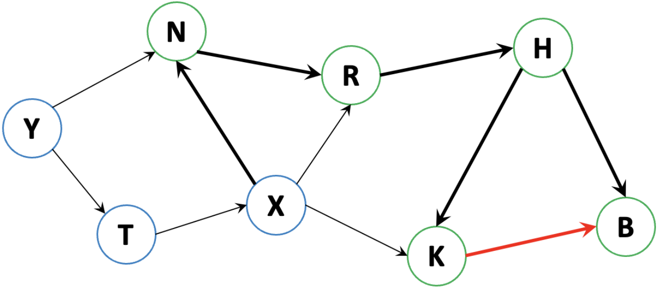
Ciclos

Comenzando en H, exploramos la otra arista de salida, encontrando K



Ciclos

Desde K volvemos a encontrarnos con B y retornamos hacia atrás



¿Qué problema tiene el algoritmo actual?

Ciclos

El algoritmo `Posteriors` no evita nodos ya visitados

- esto no solo impacta en la complejidad práctica
- si hay un ciclo en el grafo, el algoritmo **no termina**

Necesitamos poder distinguir a los ya visitados

Agregamos un atributo a los nodos

Ciclos

input : Grafo G, nodo $X \in V(G)$

$\texttt{Posteriors}(G, X)$:

if $X.visited = 1$ **: return** $\varnothing$

$X.visited \leftarrow 1$

$P \leftarrow \varnothing$

for Y *tal que* $X \to Y$ **:**

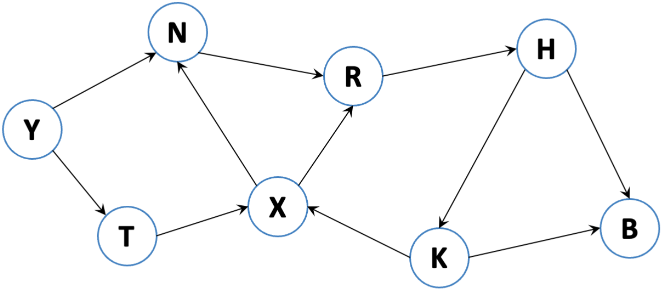
$P \leftarrow P \cup \{Y\}$

$P \leftarrow P \cup \texttt{Posteriors}(G, Y)$

return P

Esta versión termina siempre

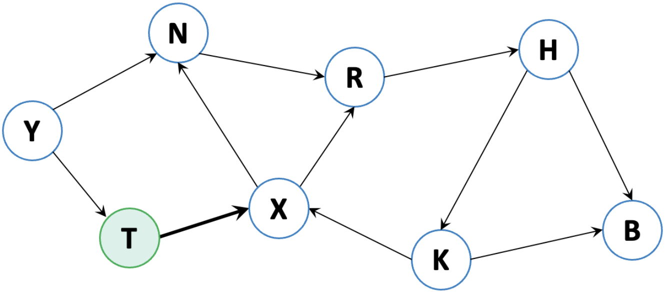
Ciclos



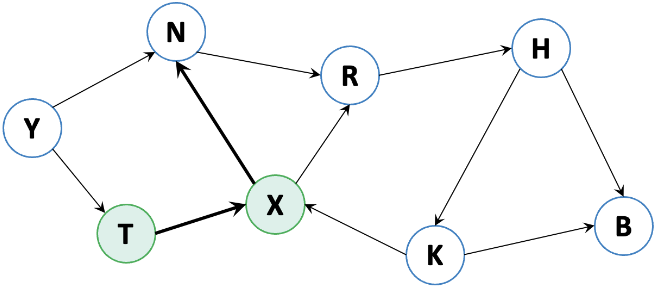
Ejercicio

Para el grafo anterior, determine los nodos posteriores a T usando el algoritmo `Posteriors` que detecta visitados

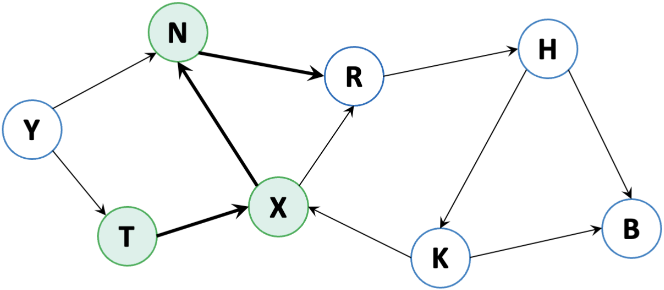
Ciclos



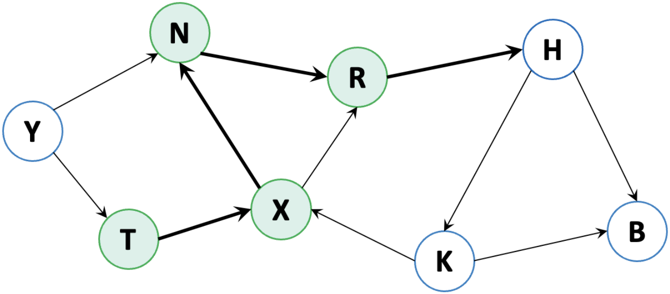
Ciclos



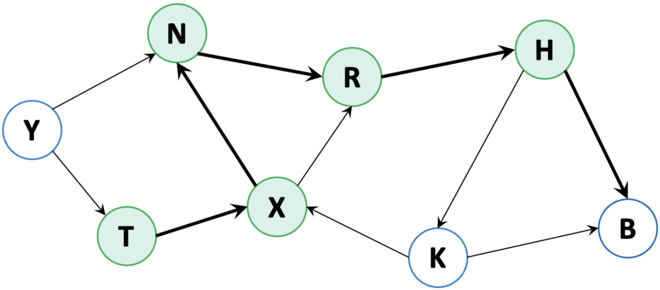
Ciclos



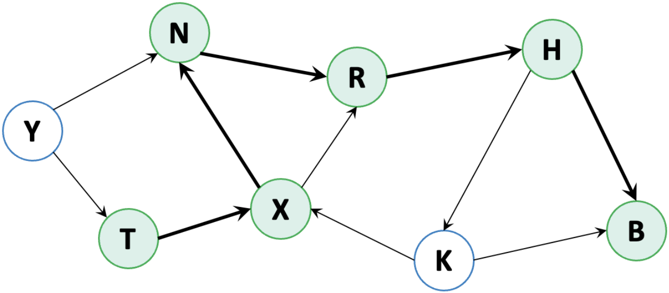
Ciclos



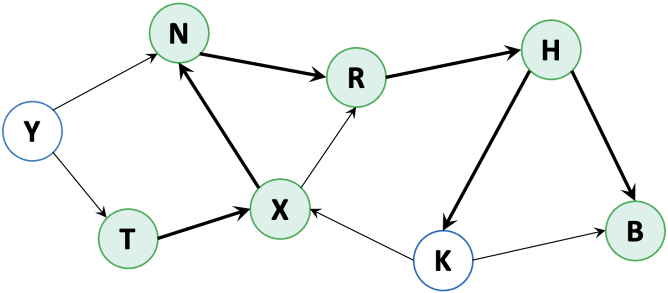
Ciclos



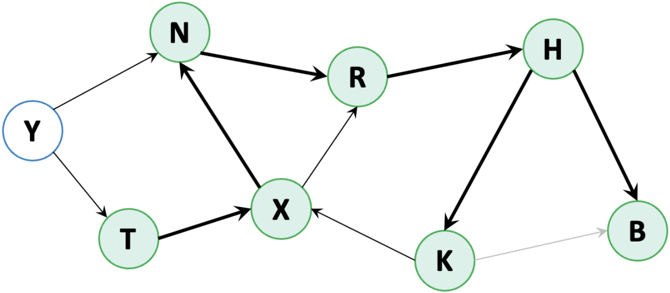
Ciclos



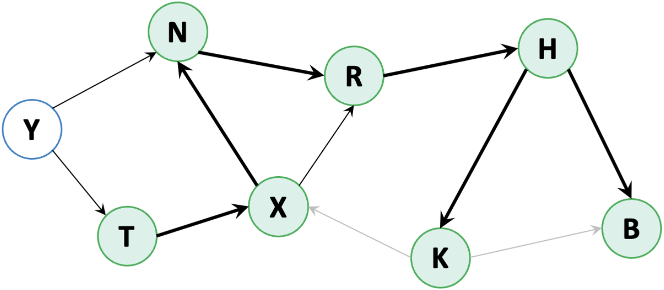
Ciclos



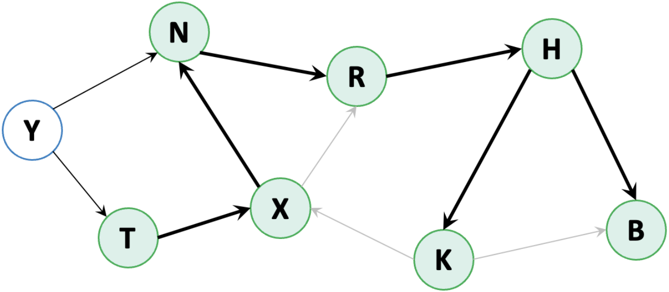
Ciclos



Ciclos

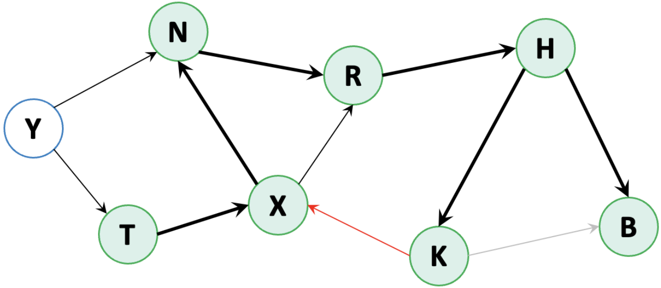


Ciclos



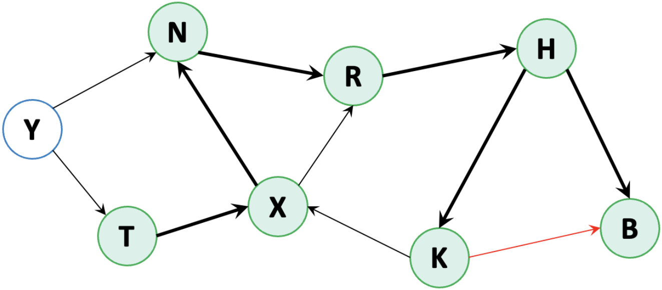
¿El algoritmo *se dio cuenta* que habían ciclos?
¿Basta decir que hay ciclo si se llama `Posteriors`(G, W) para W visitado?

Ciclos



Queremos que esto sea detectado como ciclo

Ciclos



Esto **NO DEBE** ser considerado un ciclo

Detección de ciclos

Proponemos la siguiente regla para detección de ciclos usando `Posteriors`

- Si el nodo Y que vamos a visitar ya fue visitado
  1. Si estamos en un nodo posterior a Y, **hay ciclo**
  2. Si no, entonces **esta arista no forma ciclo** (pueden haber en otras zonas)

Importante: hasta que $\texttt{Posteriors}(G, X)$ retorne, todos los visitados que se marcan son posteriores a X

Agregamos colores para distinguir el **tipo de visitado**:
blanco=no visitado, gris=visitado posterior, negro=visitado no posterior

Detección de ciclos

input : Grafo G, nodo $X \in V(G)$

`cycleAfter`(G, X)**:**

    if $X.color = gris$ **: return true**
    if $X.color = negro$ **: return false**
    $X.color \leftarrow$ gris
    for Y *tal que* $X \rightarrow Y$ **:**
        if `cycleAfter`(G, Y) **: return true**
    $X.color \leftarrow$ negro
    return false

Este algoritmo decide si hay un ciclo **partiendo desde** X

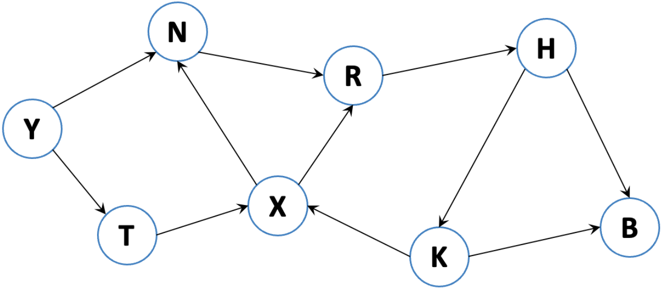
Detección de ciclos

input : Grafo G

```
isCyclic(G):
    for X ∈ V(G) :
        if X.color ≠ blanco :
            continue
        if CycleAfter(G, X) :
            return true
    return false
```

Este algoritmo decide si hay un ciclo **en el grafo**

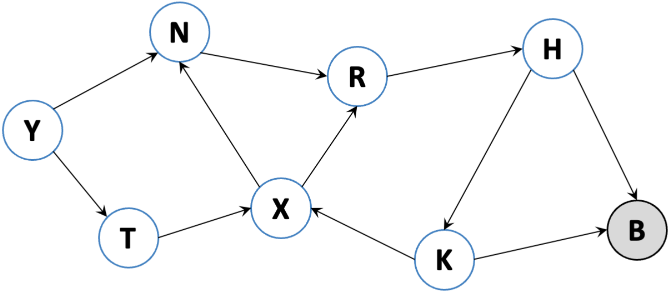
Detección de ciclos



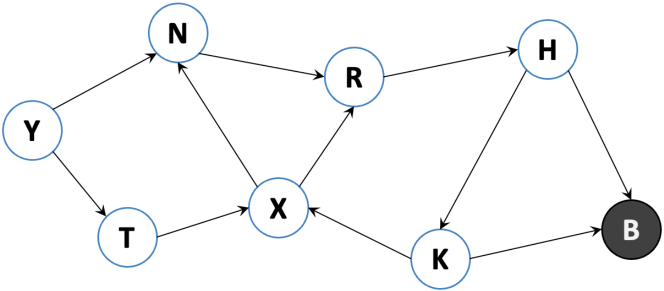
Ejercicio

Para el grafo anterior, determine si posee algún ciclo usando el algoritmo `isCyclic`. Asuma que el **for** de la línea 1 de `isCyclic` escoge los nodos en orden alfabético

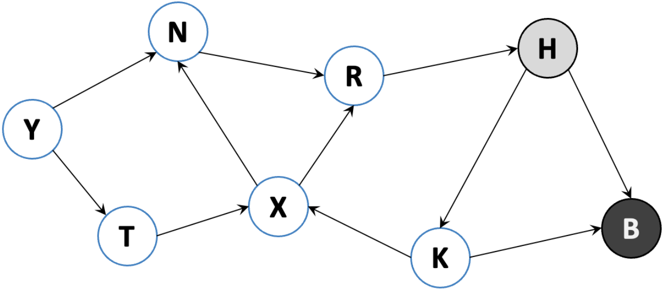
Detección de ciclos



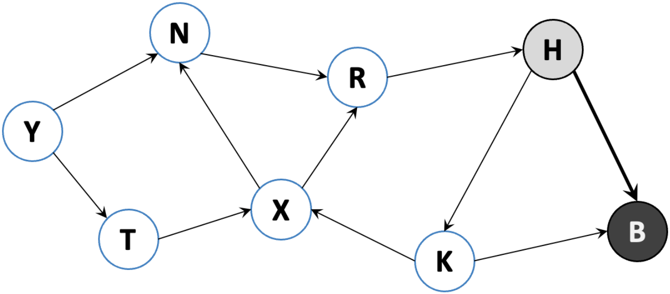
Detección de ciclos



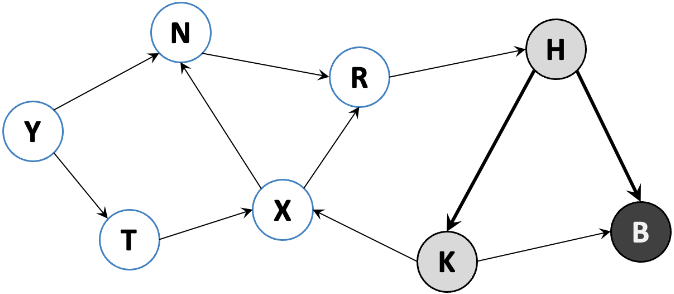
Detección de ciclos



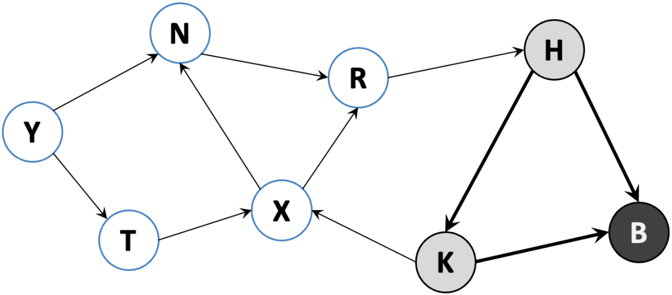
Detección de ciclos



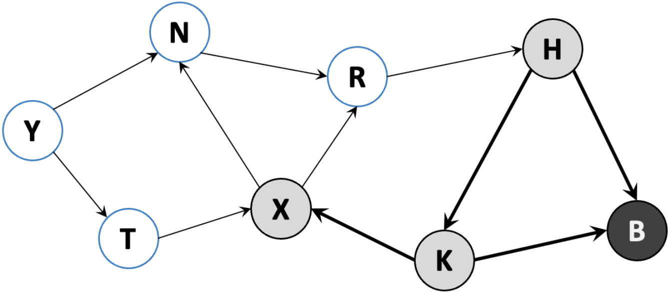
Detección de ciclos



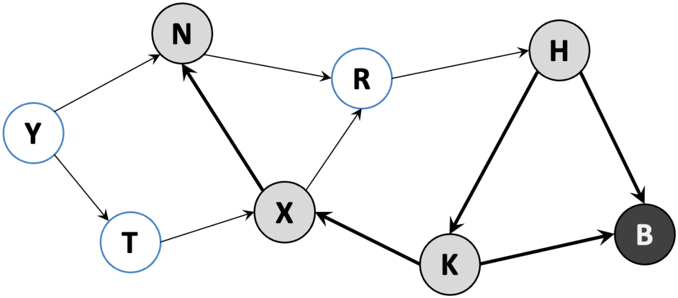
Detección de ciclos



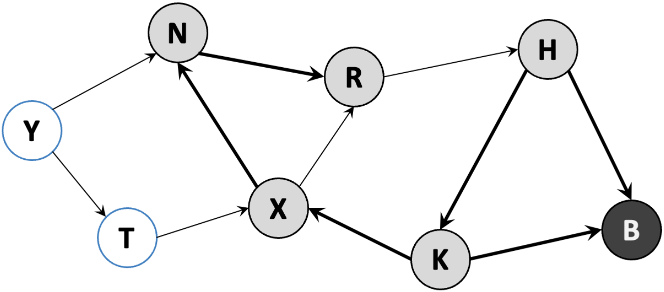
Detección de ciclos



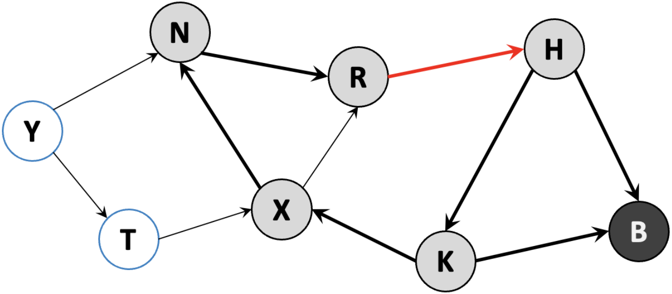
Detección de ciclos



Detección de ciclos



Detección de ciclos



Búsqueda en profundidad

El algoritmo `isCyclic` sigue una estrategia de **búsqueda en profundidad**

- también abreviada **DFS** por las siglas de *Depth First Search*
- la estrategia es avanzar por aristas adyacentes hasta más no poder
- luego de agotar esa ruta, se retrocede y se exploran otras
- todas las rutas se exploran **en profundidad**

¿Cuál es la complejidad de estos algoritmos?

Sumario

Objetivos de la clase

- ☐ Comprender el concepto de grafo y sus uso para modelar
- ☐ Identificar un algoritmo para detección de ciclos
- ☐ Identificar estrategias para evitar loops en algoritmos en grafos