Topological Sort y Componentes Fuertemente Conexas

Melanie Sclar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Algoritmos y Estructuras de Datos III

Contenidos

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 debamos 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

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 $\mathit{indegree} = 0$ pero aún no colocamos todos los nodos, significa que el grafo no era un DAG.

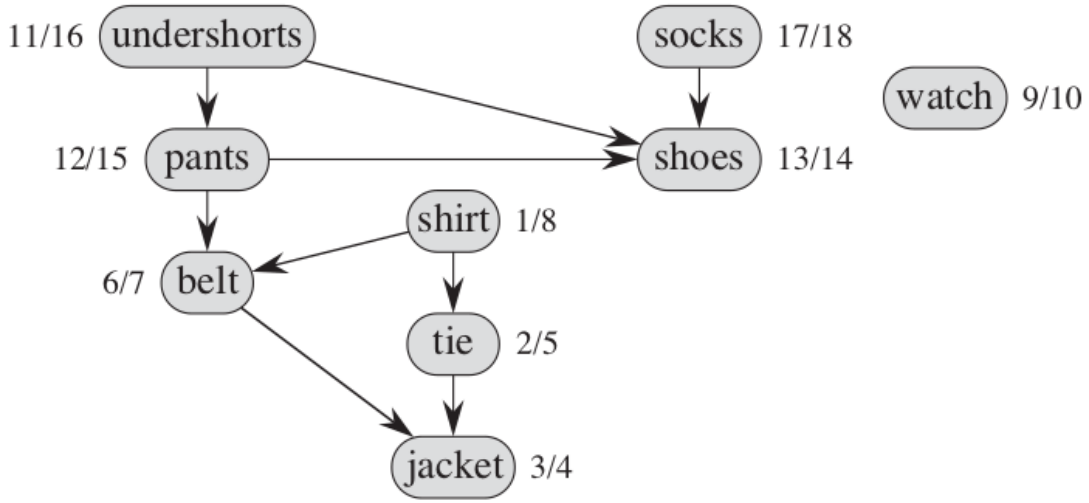
Pseudocódigo del toposort con in-degree

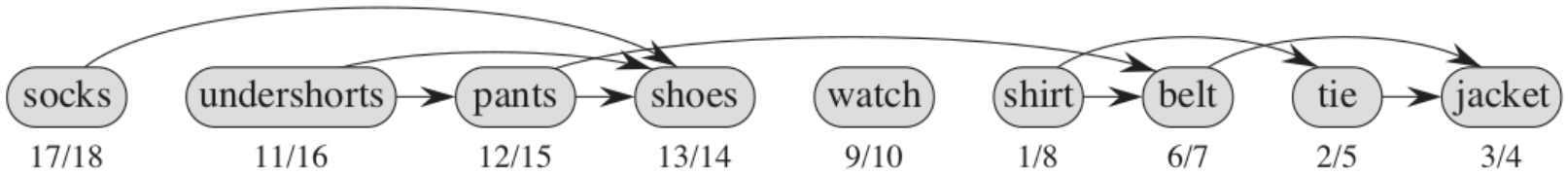
```
1  L = Lista vacia que contendra los elementos ordenados
2  S = Conjunto de nodos sin nodos entrantes (in-degree = 0)
3
4  mientras S no sea vacio:
5      removemos un nodo v de S (cualquiera sirve)
6      agregamos v al final de L
7      para cada nodo w tal que existe el eje (v, w):
8         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

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

Existen otras formas de calcular el Topological Sort que no veremos hoy. Para los interesados, pueden ver las diapositivas de PAP o el capítulo 22 del Cormen.

Ejemplo de Topological Sort con DFS





Ejemplo totalmente no robado del Cormen

Contenidos

Problema: Dudu Service Maker

Dudu necesita un documento para finalizar una tarea en su trabajo. Después de mucho buscar, encontró que este documento requería otros documentos, que también necesitaban otros documentos y así siguiendo.

Dudu hizo una lista con todos los documentos que necesitará. Él sospecha que esta lista contiene ciclos. Por ejemplo, si un documento A depende del B y a su vez el B también depende del A, la tarea sería imposible de terminar.

Dada la lista de dependencias, ayudalo a decidir si puede obtener todos los documentos, o sea, si hay un ciclo en la lista de dependencias o no.

`https://www.urionlinejudge.com.br/judge/en/problems/view/1610`

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`

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

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

Supongamos que sólo nos interesara saber si un grafo es fuertemente conexo o no. ¿Qué podemos hacer?

Checkear si un grafo es fuertemente conexo

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 recorrerlo 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 recorro el grafo 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.

¡A programar!

En una ciudad hay N intersecciones conectadas por calles de mano única y/o doble mano. Es evidente que toda ciudad razonable debería cumplir que dadas dos intersecciones cualesquiera v y w se debe poder viajar de v a w y de w a v.

Tu tarea es escribir un programa que lea la descripción de la ciudad y determine si esta ciudad es razonable o no.

Ya comenzamos a programar el problema, pero nos falta la lógica principal. El código se llama `1128_ComeAndGo.cpp`.

`https://www.urionlinejudge.com.br/judge/en/problems/view/1128`

Hoy no alcanzaremos a ver un algoritmo para detectar las componentes fuertemente conexas de un grafo.

Uno de los más populares se llama Kosaraju, y para los interesados pueden leerlo de las diapositivas de PAP (notar que este algoritmo excede lo que veremos en la materia).

¿Preguntas?

Para los interesados, acá dejo dos problemas que se resuelven con lo visto anteriormente.

Contenidos

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.

$$(x_0 \vee x_2) \wedge (x_0 \vee \neg x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge$$
$$(x_2 \vee \neg x_4) \wedge (x_0 \vee \neg x_5) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_5) \wedge$$
$$(x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6)$$

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

2-SAT: modelado del grafo

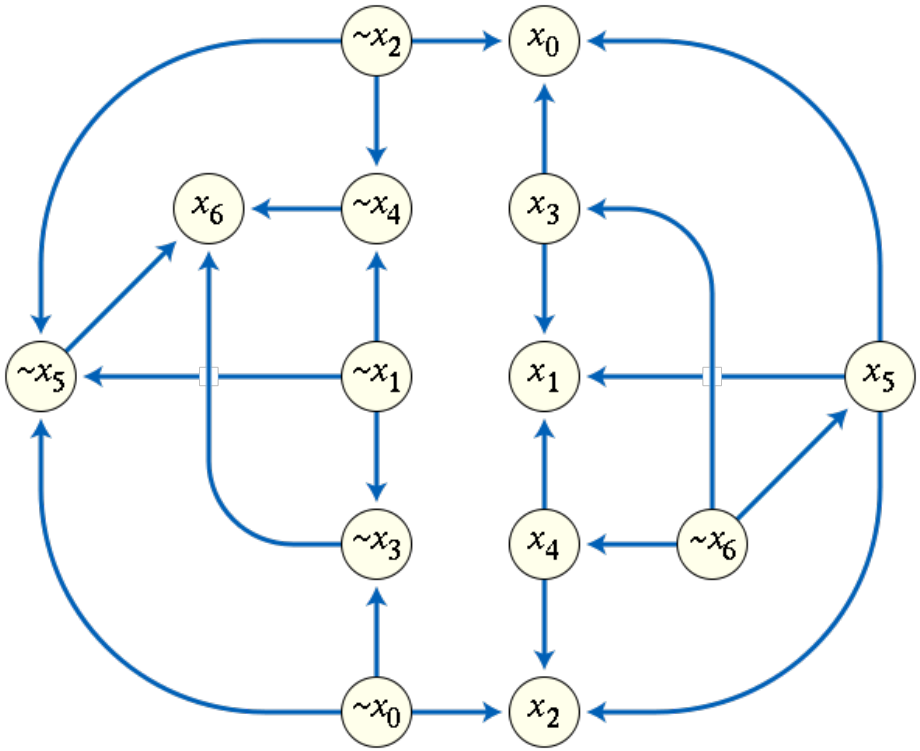
- Podemos pensar que un eje dirigido (u, v) significa que u implica v.
- Modelamos los nodos como cada variable y su negación: tendremos un nodo x_i y otro nodo $\neg x_i$ para todo i.
- ¿Cómo deben ser los ejes que modelen las fórmulas binarias de la entrada?

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?

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

$$(x_0 \vee x_2) \wedge (x_0 \vee \neg x_3) \wedge (x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_4) \wedge$$
$$(x_2 \vee \neg x_4) \wedge (x_0 \vee \neg x_5) \wedge (x_1 \vee \neg x_5) \wedge (x_2 \vee \neg x_5) \wedge$$
$$(x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6)$$



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 checkeo 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 $a_1, a_2, ..., a_k$ entonces existe otra que contiene la negación de todos estos: $\neg a_1, \neg a_2, ..., \neg a_k$.

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 $a_1, a_2, ..., a_k$ entonces existe otra que contiene la negación de todos estos: $\neg a_1, \neg a_2, ..., \neg a_k$.

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.