# *Clasificación topológica y camino crítico*

Por N. Aguilera

Mayo de 2019

En ocasiones aparece naturalmente un *orden parcial* entre objetos. Wirth (1987, p. 202) propone algunos ejemplos interesantes, y agregamos alguno más:

1. En un diccionario, las palabras se definen en términos de otras, y podemos indicar que la palabra $u$ depende o está definida en términos de la palabra $v$ mediante $v \prec u$ ($v$ precede a $u$).
2. Para concretar un proyecto, posiblemente sea necesario realizar distintas tareas, algunas de ellas en un orden dado. Por ejemplo, buscar la escalera antes de subir al techo. Si la tarea $u$ debe realizarse antes que la tarea $v$, podemos poner $u \prec v$.
3. En un currículum universitario, ciertos materias deben realizarse antes que otras. Pondremos entonces $u \prec v$ si el curso $u$ tiene que aprobarse antes de cursar $v$.
4. En un programa de computación posiblemente se usen funciones que a su vez dependen de otras funciones, y podemos indicar con $u \prec v$ que la función $v$ depende de la función $u$.
5. En una organización, hay toda una jerarquía de dependencias y podemos poner $u \prec v$ si el empleado $u$ tiene bajo su responsabilidad al empleado $v$.

Como vemos, la precedencia $\prec$ puede referirse a una variedad de categorías, más allá de las cuestiones temporales.

Recordemos que un orden parcial $\prec$ en un conjunto $A$ es una relación en $A \times A$ con las propiedades:

*a*) (transitividad) $a \prec b$ y $b \prec c$ entonces $a \prec c$.

*b*) (antisimetría) si $a \neq b$ y $a \prec b$ entonces $b \not\prec a$.

Dependiendo de lo que se quiera hacer, a veces se pide que la relación sea reflexiva:

*c*) $a \prec a$

o justamente todo lo contrario, como propone Wirth (1987, p. 203):

*b'*) $a \not\prec a$.

✍ En matemáticas la inclusión entre conjuntos y la relación de divisibilidad en $\mathbb{N}$ son órdenes parciales que además tienen otras propiedades que los transforman en *retículos* o *reticulados* (*lattice* en inglés): dados $a$ y $b$ no comparables entre sí existen $c$ y $d$ tales que

*i*) $c \prec a \prec d$, $c \prec b \prec d$,

*ii*) si $u \prec a$ y $u \prec b$ entonces $u \prec c$,

*iii*) si $a \prec v$ y $b \prec v$ entonces $d \prec v$.

$c = a \wedge b$ es el *ínfimo* entre $a$ y $b$ y $d = a \vee b$ es el *supremo* de $a$ y $b$. En en el caso de conjuntos, $\vee$ es la unión y $\wedge$ es la intersección. En el caso de divisibilidad, $\wedge$ es el máximo común divisor y $\vee$ es el mínimo común múltiplo.

En un orden parcial arbitrario $a \wedge b$ y $a \vee b$ pueden no existir.

Recordemos que un *orden total* o *lineal* es un orden parcial en la que todo par de elementos es comparable, y que el orden lineal $\preccurlyeq$ es una extensión del orden parcial $\prec$ si lo que lo que antes era válido en $\prec$ sigue siendo válido en $\preccurlyeq$: $a \prec b$ implica $a \preccurlyeq b$.

Por distintos motivos a veces es conveniente extender el orden parcial a un orden total compatible.

Esto es claro en el ejemplo de las tareas cuando hay una única máquina o un único trabajador para hacerlas: queremos poner las tareas en orden secuencial.

El ejemplo del diccionario, en cambio, nos muestra que no siempre es posible hacer la extensión: seguramente habrá una palabra definida en términos de otra que a su vez, tal vez con varias palabras intermedias, depende de la primera. Por ejemplo, según el diccionario de la Real Academia Española, *verdad* es «la cualidad de veraz», mientras que *veraz* es «que dice, usa o profesa siempre la verdad». Se trata de definiciones *circulares*, que abundan en los diccionarios.

✍ Euclides trató de evitar estos problemas dando «definiciones», «nociones comunes» y «axiomas»: por algún lado hay que empezar.

En el medio dejó cosas que hoy nos resultan al menos curiosas, como «punto es aquello que no tiene parte» o «línea recta es aquella que yace por igual respecto de los puntos que están en ella».

Si el orden parcial $\prec$ definido en $A$ satisface las condiciones *a)*, *b)* y *b')*, podemos definir un grafo dirigido a partir de él en el que los nodos son los elementos de $A$ y $(a, b)$ es un arco si $a \prec b$. El ejemplo del diccionario nos dice que si queremos extender el orden parcial a uno total, este grafo no debe tener ciclos.

El algoritmo en Wirth (1987) consiste en crear, para cada nodo, una lista de sucesores y la cantidad de predecesores. El orden lineal se construye tomando los que no tienen predecesores, borrándolos del grafo y actualizando los datos de sus sucesores. Si no se eliminan todos los nodos quiere decir que hay un ciclo, que construimos en nuestra implementación en Python.

Para construir el ciclo, tomamos ventaja de que todos los nodos que quedan tienen grado de entrada positivo, por lo que revirtiendo el sentido de los arcos cualquier camino debe cerrarse repitiendo un nodo.

✍ Podría haber más de un ciclo, nosotros construimos sólo uno.

Wirth denomina al método *clasificación topológica*, por lo que nuestro programa se llama *toposort*. Si bien se basa en la presentación de Wirth, la nuestra es una versión simplificada, usando las listas de Python.

✍ Esperemos haber mantenido el espíritu del algoritmo.

Técnicamente usamos «arreglos paralelos», evitando «clases con objetos», «registros» y «listas encadenadas».

Si agregamos tiempos de ejecución a cada tarea del ejemplo 2, nos interesará encontrar el tiempo mínimo en que se pueden terminar todas las tareas, idealmente identificando aquellas que son *críticas*, cuyo comienzo y ejecución no pueden retrasarse sin retrasar todo el proyecto, y dando un tiempo de *holgura* o *flotante* en la nomenclatura de Biggs (1998, p. 261) a las tareas no críticas, lo que puede ayudar a decidir si destinar más recursos a las tareas críticas o menos a las no críticas.

Por supuesto que suponemos que si no hay precedencias entre dos tareas, estas pueden desarrollarse simultáneamente: en otro caso, el tiempo mínimo será la suma de todos los tiempos. Es decir, ahora no interesa poner las tareas en un orden lineal.

Para resolver este problema usaremos el *método del camino crítico*, que abreviaremos *CPM* por *Critical Path Method* en inglés.

Es usual interpretar a las tareas como arcos en un grafo dirigido (en vez de nodos), asignando como longitud al tiempo que requiere la tarea. Curiosamente, el tiempo mínimo para completar todas las tareas corresponde al camino más largo en este grafo (y de allí el nombre de «camino» crítico).

A fin de aprovechar las ideas del algoritmo para orden lineal, nosotros pensaremos en las tareas como nodos y no arcos (en inglés, *activity on node* en vez de *activity on arc*), asignando el tiempo al nodo como si se tratara de una cabina de peaje. Esto es equivalente a pensar que cada tarea $a$ da lugar a un arco de la forma $(a^-, a^+)$, con la longitud igual al tiempo que demanda la tarea, y las relaciones $a \prec b$ o $c \prec a$ se traducen en los arcos $(a^+, b^-)$ y $(c^+, a^-)$ respectivamente, de longitudes nulas.

Nuestra versión del método CPM, en el programa *cpm*, repite esencialmente los pasos del algoritmo para orden lineal, agregando nodos $s$ y $t$ de salida y llegada, guardando el tiempo acumulado para llegar a cada nodo desde $s$.

Finalmente, los tiempos de holgura o "slack" se obtienen repitiendo el procedimiento, pero ahora "de atrás hacia adelante". Las tareas críticas corresponden exactamente a aquéllas que tienen holgura 0.

# Referencias


N. L. Biggs, 1998. *Matemática discreta*. Vicens Vives.

N. Wirth, 1987. *Algoritmos y Estructuras de Datos*. Prentice-Hall Hispanoamericana.

# Apéndice: programas en Python

- Recordar que *en Python, los subíndices de las listas empiezan en* 0.
- Dependiendo del lector de pdf (Adobe Reader no tiene problemas), es posible extraer los módulos *toposort.py* y *cpm.py* como archivos de texto haciendo «click» con el botón derecho del ratón en los respectivos enlaces y eligiendo la opción de «guardar a disco» o similar (cambiar la extensión de .txt a .py para su ejecución con Python).

## toposort.py

```
"""Orden lineal.

- Dadas precedencias entre pares de nodos, se trata de
  encontrar un orden total que extienda al orden
  parcial dado.

  El resultado es una lista de los nodos en un orden
  lineal compatible y los nodos en un ciclo si no es
  posible incluir a todos los nodos (el ciclo es vacío
  cuando todos los nodos están en el orden lineal).

- Wirth lo llama "clasificación topológica" y de ahí el
  nombre "toposort".

"""

#-----------------------------------------------

def toposort(n, arcos):
    """Clasificación topológica.

    - n es la cantidad de nodos (numerados a partir
      de 1), y arcos es una lista de elementos de
      la forma (a,b) indicando que a precede a b.

    - No deben haber arcos repetidos ni bucles
      (no se comprueba si los datos son correctos).

    - Se retornan dos listas:

      - Los nodos en un orden lineal

      - Los nodos de un ciclo.

      - Si existe un orden lineal compatible para
        todos los nodos, el ciclo es vacío.

    - Seguimos a Wirth, p. 202, agregando la
```

```
  búsqueda de un ciclo, y simplificando las
  estructuras de datos usando listas de Python
  en vez de listas encadenadas con punteros.

"""

nodos = range(n + 1)
nodos1 = nodos[1:]

# encontrar vecinos y cantidad de predecesores
vecs = [[] for x in nodos]
vecs[0] = None
preds = [0 for x in nodos]
for x, y in arcos:
    vecs[x].append(y)
    preds[y] = preds[y] + 1

# cola inicial de nodos a procesar
aprocesar = []
for x in nodos1:
    if preds[x] == 0:
        aprocesar.append(x)

# construcción del orden lineal
ordenlineal = []
while len(aprocesar) > 0:
    x = aprocesar.pop(0)  # o x = aprocesar.pop()
    ordenlineal.append(x) # ponerlo en orden lineal
    for y in vecs[x]:     # actualizar predecesores
        preds[y] = preds[y] - 1
        if preds[y] == 0:
            aprocesar.append(y)

# acá no quedan nodos para procesar

if len(ordenlineal) == n:
    return ordenlineal, []

# En el grafo residual, después de haber
# eliminado los nodos en el orden lineal
# debe haber un ciclo.

# los nodos del nuevo grafo
quedan = [x for x in nodos1 if preds[x] > 0]

# buscamos un ciclo dando vuelta los arcos para
# que los vértices que quedan tengan todos
# grado de salida > 0, facilitando la búsqueda

vecs2 = [[] for x in nodos]
```

```
    vecs2[0] = None
    for x in quedan:
        for y in vecs[x]:
            vecs2[y].append(x)

    # Hacemos un recorrido en profundidad sencillo
    padre = [None for x in nodos]
    x = quedan[0]  # raíz
    padre[x] = x
    while True:
        # basta un vecino de cada visitado
        y = vecs2[x][0]
        if padre[y] == None:
            padre[y] = x
            x = y
        else: # se armó un ciclo
            break
    ciclo = [y]
    while x != y:
        ciclo.append(x)
        x = padre[x]
    # no hay que dar vuelta el orden del ciclo
    # porque habíamos invertido los arcos

    return ordenlineal, ciclo
```

### cpm.py

```
"""Camino crítico.

- Los nodos representan tareas que tienen precedencias
  entre sí y cada una requiere de cierto tiempo para
  realizarla.

  Deben estar numerados a partir de 1 a n.

  El algoritmo agrega nodos s = 0 y t = n + 1.

- Se retorna (lista, ccrit), donde

  - lista tiene entradas de la forma [k, tmin, tmax,
  dif], con

    - k es el número de tarea (entre 0 y n+1),

    - tmin es el tiempo mínimo en el cual puede
      terminar la tarea,

    - tmax es el tiempo máximo en el cual puede
```

```
      terminar la tarea.

    - dif es el "slack", diferencia entre los tiempos
      anteriores.

  - ccrit es un camino crítico.

- Podría haber más de un camino crítico, acá se retorna
  uno solo.

- Copiamos las idesa de toposort agregando los tiempos.

- No construimos ciclo (como en toposort).

"""

#---------------------------------------------

def cpm(n, tmp, rel):
    """Encontrar el camino crítico de tareas.

    - n es el número de tareas.

    - Para cada tarea k, se da tmp[k], el tiempo que
      tarda en completarse.

    - Se dan las relaciones rel: [a,b] significa que
      la tarea a viene antes de la tarea b.

    - Siguiendo a Biggs (p. 259), usamos un nodo
      inicial s y uno final t, que no usan tiempo.

      Resultará s = 0 y t = n + 1.

    - Consideramos tareas sobre nodos y no sobre arcos.

    """

    # Agregamos s = 0 adelante y t = n + 1 atrás.

    s = 0
    t = n + 1  # ahora hay n + 2 nodos
    nodos = range(n + 2)

    # El tiempo total acumulado para realizar una
    # tarea es el tiempo propio más el acumulado de
    # las tareas que lo preceden.
    # Inicialmente ponemos sólo el tiempo propio.
    tmpn = [0] + tmp + [0] # tiempos de n + 2 nodos
    tmin = tmpn[:]  # tiempos mínimos para terminar
```

```
# agregamos los arcos s-nodo y nodo-t
# el arco s-t no es necesario
reln = ([[s,a] for a in range(1,n+1)]
        + rel
        + [[a,t] for a in range(1,n+1)])

# repetimos lo hecho en toposort, agregando
# tiempos acumulados
preds = [0 for x in nodos]
sucs = [[] for x in nodos]
for x, y in reln:
    sucs[x].append(y)
    preds[y] = preds[y] + 1

# sólo s no tiene predecesores inicialmente
aprocesar = [s]
padre = [0 for x in nodos]  # para camino crítico
while len(aprocesar) > 0:
    x = aprocesar.pop(0)    # o aprocesar.pop()
    # actualizar datos de los sucesores
    for y in sucs[x]:
        preds[y] = preds[y] - 1
        if preds[y] == 0:
            aprocesar.append(y)
        if tmin[y] < tmpn[y] + tmin[x]:
            tmin[y] = tmpn[y] + tmin[x]
            padre[y] = x

# el último procesado debe ser t,
# que tiene n + 1 predecesores

if x != t:  # t no se procesó
    print("*** Error: posible ciclo")
    return

# else;
# fabricamos un camino crítico
crit = [x]
while x != s:
    x = padre[x]
    crit.append(x)  # poner adelante
crit.reverse()  # dar vuelta la lista

# El tiempo máximo de finalización se obtiene
# yendo p'al otro lao.

tfin = tmin[t]
tmax = [tfin for x in nodos]
sucs = [0 for x in nodos]
```

```
    preds = [[] for x in nodos]
    for x, y in reln:
        preds[y].append(x)
        sucs[x] = sucs[x] + 1

    # sólo t no tiene sucesores inicialmente
    aprocesar = [t]
    while len(aprocesar) > 0:
        x = aprocesar.pop(0)    # o aprocesar.pop()
        # actualizar datos
        for y in preds[x]:
            sucs[y] = sucs[y] - 1
            if sucs[y] == 0:
                aprocesar.append(y)
            if tmax[y] > tmax[x] - tmpn[x]:
                tmax[y] = tmax[x] - tmpn[x]

    sal = [(k, tmin[k], tmax[k], tmax[k] - tmin[k])
           for k in nodos]

    return sal, crit
```