Resumen TDA

Created	@September 19, 2024 7:35 PM
Class	TDA

Complejidad Computacional:

▶ En el contexto de la teoría de complejidad computacional, llamamos *problema* a la descripción de los datos de entrada y la respuesta a proporcionar para cada dato de entrada.

▶ Una *instancia* de un problema es un conjunto válido de datos de entrada.

▶ Ejemplo:

1. **Entrada**: Un número n entero no negativo.
2. **Salida**: ¿El número n es primo?
3. **Instancia :** numero entero no negativo.

▶ En este ejemplo, una instancia está dada por un número entero no negativo.

Complejidad :

Dicho de una forma informal, la complejidad de un algoritmo es una función que representa el tiempo de ejecución en función del tamaño de la entrada.

$$Dadas\ dos\ funciones\ f, g : \mathbb{N} \to \mathbb{R},\ decimos\ que:$$

$$\blacktriangleright f(n) = O(g(n))\ si\ existen\ c \in \mathbb{R}^+\ y\ n0 \in \mathbb{N}\ tales\ que\ f(n) \leq c\ g(n)\ para\ todo\ n \geq n0.$$

$$\blacktriangleright f(n) = \Omega(g(n))\ si\ existen\ c \in \mathbb{R}^+\ y\ n0 \in \mathbb{N}\ tales\ que\ f(n) \geq c\ g(n)\ para\ todo\ n \geq n0$$

$$\blacktriangleright f(n) = \Theta(g(n))\ si\ f = O(g(n))\ y\ f = \Omega(g(n)).$$

$$\blacktriangleright Cualquier\ función\ exponencial\ es\ peor\ que\ cualquier\ función\ polinomial:\ si\ k \in R > 1\ y\ d \in N, entonce$$

$$\blacktriangleright La\ función\ logarítmica\ es\ mejor\ que\ la\ función\ lineal\ (no\ importa\ la\ base), es\ decir,\ \log n\ es\ O(n), pero\ no\ a\ la\ inversa.$$

OPTIMIZACIÓN

Un problema de optimización consiste en encontrar la mejor solución dentro de un conjunto:

$$z^* = \max_{x \in S} f(x) \quad \text{o bien} \quad z^* = \min_{x \in S} f(x)$$

$$(maximización\ o\ minimización)$$

$$\blacktriangleright La función f : S \to \mathbb{R}\ se\ denomina\ función\ objetivo\ del\ problema.$$

$$\blacktriangleright El\ conjunto\ S\ es\ la\ región\ factible\ y\ los\ elementos\ x \in S\ se\ llaman\ soluciones\ factibles.$$

$$\blacktriangleright El\ valor\ z^* \in \mathbb{R}\ es\ el\ valor\ óptimo\ del\ problema, y\ cualquier\ solución\ factible\ x^* \in S\ tal\ que\ f(x*) = z*\ se\ llama\ un\ óptimo\ del\ problema.$$

Problemas de optimización combinatoria

▶ Un problema de optimización combinatoria es un problema de optimización cuya región factible es un conjunto definido por consideraciones combinatorias.Es decir,la región factible contiene subconjuntos o permutaciones de un conjunto finito de elementos.

Algoritmos que resuelven problemas de optimización combinatoria.

Fuerza Bruta :

Consiste en generar todas las soluciones factibles y descartar las que no cumplen con las restricciones del problema.

Backtracking :

Consiste en recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones del problema buscando aquellas que cumplen con las propiedades esperadas,y descartando las configuraciones parciales que no las cumplen.

La idea es extender la solución parcial (configuraciones parciales) hasta obtener una solución completa.Generalmente se utiliza un vector a = (a1,...,an) para representar a la solución candidata,y cuando ya no puede ser extendida,se **retrocede** hasta hallar una configuración que permita avanzar de forma diferente.

Cuando se descarta una configuración parcial que no cumple con las propiedades del problema,decimos que **podamos esa rama del árbol,**porque sabemos que si seguimos explorando esa rama,no llegaremos de igual forma a una solución válida.

¿Por qué hablamos de árboles y ramas? Porque un algoritmo de backtracking se toman decisiones sucesivas y cada nodo representa una decisión tomada. La arista actual en un árbol de backtracking representa la solución parcial hasta el nodo actual.

Las podas pueden realizarse por :

- Factibilidad : La solución parcial no cumple con las restricciones del problema,y por lo tanto,ninguna extensión de esta lo hará.
- Optimalidad : Hallé anteriormente una solución más óptima que la solución parcial actual.

```
Backtracking: Esquema General
BT(a,k)
    entrada: a = (a1, . . . , ak) solucion parcial
    salida: sol = (a1, . . . , ak, . . . , an) solucion valida
    si k == n + 1 entonces
        sol ← a
        encontro ← true
    sino
    para cada a
            0 ∈ Sucesores(a, k)
            BT(a,0, k + 1)
            si encontro entonces
                        retornar
            fin si
    fin para
    fin si
retornar
```

Para demostrar la correctitud de un algoritmo de backtracking, debemos demostrar que se enumeran todas las posibles configuraciones validas. Es decir, que las ramificaciones y podas son correctas.

Programación dinámica :

Un problema a resolverse con programación dinámica cumple :

Se divide el problema en subproblemas de tamaños menores que se resuelven recursivamente. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.

▶ Superposición de estados: El árbol de llamadas recursivas resuelve el mismo subproblema varias veces.Por lo tanto,la cantidad de llamadas a la función es mayor o igual a la cantidad de estados dintintos dentro de la función.

▶ El problema tiene **subestructura óptima** si la solución óptima de un problema más grande puede construirse a partir de las soluciones óptimas de sus subproblemas más pequeños.

Esto implica que puedes dividir el problema en subproblemas independientes, resolver esos subproblemas, y luego combinar sus soluciones para obtener la solución del problema original.

▶ Alternativamente, podemos decir que se realizan muchas veces llamadas a la función recursiva con los mismos parámetros.
▶ Un algoritmo de programación dinámica evita estas repeticiones con alguno de estos dos esquemas:

1. Enfoque **top-down**. Se implementa recursivamente, pero se guarda el resultado de cada llamada recursiva en una estructura de datos (memorización). Si una llamada recursiva se repite, se toma el resultado de esta estructura.
2. Enfoque **bottom-up**. Resolvemos primero los subproblemas más pequeños y guardamos (habitualmente en una tabla) todos los resultados.

Receta para resolver un problema con PD :

1. Definir función recursiva f: Elegir parámetros,casos base y pasos recursivos.
2. Explicar la semántica de f: Establecer una relación entre la naturaleza recursiva del problema y nuestra función f,explicar casos bases,restricciones...
3. Definir que llamado/s a f resuelven el problema.
4. Probar que f cumple con la propiedad de superposición de problemas: Queremos contar las combinaciones posibles de los parámetros de la función recursiva y llamados que hace mi función recursiva, y determinar condiciones para que haya superposición.
   La receta es:
   ● Determinar el mejor caso de llamados recursivos de f si es fácil (sino un estimado de peor caso que se asemeje a algún caso real de f):
   ○ Vemos cuantos llamados recursivos se hacen en cada momento, y cómo cambian los parámetros de f en el mejor caso.
   ● Determinar la cantidad de combinaciones de los parámetros de mi función
   ○ Para los casos que tratamos basta con multiplicar los rangos de valores de cada parámetro de f.
   ● Entender en qué casos hay muchos más llamados que formas de llamar a la función
   ○ Determinar cómo deben ser los parámetros para que sus combinaciones sean mucho menos que los llamados.
5. Definir un algoritmo para f.
6. Determinar complejidad del algoritmo: La complejidad de un algoritmo con programación dinámica es equivalente a: (cantidad de estados) * (costo de calcular estado)

Greedy

La idea es construir una solución paso por paso, de manera de hacer la mejor elección posible localmente según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección. Es decir, en cada etapa se toma la decisión que parece mejor basándose en la información disponible en ese momento, sin tener en cuenta las consecuencias futuras. Una decisión tomada nunca es revisada y no se evalúan alternativas.

Estos algoritmos son fáciles de desarrollar e implementar y, cuando funcionan, son eficientes. Sin embargo, muchos problemas no pueden ser resueltos mediante esta técnica. En esos casos, proporcionan heurísticas sencillas que en general permiten construir soluciones razonables, pero subóptimas.

Una heurística es un procedimiento computacional que intenta obtener soluciones de buena calidad para un problema,intentando que su comportamiento sea lo más preciso posible.

Para que un problema admita una solución greedy,debe cumplir con las siguientes propiedades:

▶ **Greedy Choice : La elección greedy que se toma en cada paso debe ser parte de la solución óptima final.**

▶ **Subestructura** : Un problema tiene subestructura óptima si una solución óptima al problema contiene dentro soluciones óptimas a sus subproblemas . Es decir,Después de realizar una greedy choice, la solución restante debe ser

un subproblema similar, y la solución óptima para el problema original incluye la solución óptima del subproblema.

Para demostrar que un algoritmo greedy es correcto,deben probarse esas dos propiedades.

Demostración de correctitud para algoritmos greedy : Usualmente la **factibilidad** (validez) de la solución greedy se demuestra por inducción,mostrando en cada iteración que la solución no incumple las restricciones del problema y termina con una solución válida.

Con la técnica **Greedy Stays Ahead**,podemos demostrar que la solución del algoritmo greedy es al menos tan buena como la solución óptima durante cada iteración del algoritmo.

Luego,podemos utilizar este hecho para demostrar que la solución greedy es la **óptima**,usualmente el argumento es realizado por contradicción asumiendo que greedy no es el óptimo y utilizando Greedy Stays Ahead para llegar a una contradicción.

Dinámica y greedy comparten la caracteristica de **subestructura óptima,**y según el Cormen,detrás de cualquier algoritmo greedy,hay un algoritmo de programación dinámica más engorroso (?

Diferencias entre PD y Greedy : PD resuelve subproblemas y luego toma decisiones,y Greedy toma decisiones y luego resuelve subproblemas.

Divide & Conquer

Dividir y conquistar es una de las técnicas más utilizadas. Se basa en la descomposición de un problema en subproblemas. Si un problema es demasiado difícil para resolverlo directamente, una alternativa es dividirlo en partes más pequeñas que sean más fáciles de resolver y, luego, uniendo todas las soluciones parciales, obtener la solución final. Es un caso particular de los algoritmos recursivos.

Formalmente, dado un problema a resolver para una entrada de tamaño *n*, se divide la entrada en *r* subproblemas. Estos subproblemas se resuelven de forma independiente y después se combinan sus soluciones parciales para obtener la solución del problema original. En esta técnica, los subproblemas deben ser de la misma clase que el problema original, permitiendo una resolución recursiva. Por supuesto, deben existir algunos casos sencillos cuya solución pueda calcularse directamente.

Se distinguen 3 partes en un algoritmo de dividir y conquistar:

1. Una forma directa de resolver casos sencillos.
2. Una forma de dividir el problema en 2 o más subproblemas de menor tamaño.
3. Una forma de combinar las soluciones parciales para llegar a la solución completa.

El esquema general es:

```
    d&c(x)
  entrada: x
  salida: y
 si x es suficientemente fácil entonces
      y ← calcular directamente
 sino
    descomponer x en instancias más chicas x1, . . . , xr
    para i = 1 hasta r hacer
          yi ← d&c(xi)
    fin para
    y ← combinar los yi
 fin si
retornar y
```

La cantidad de subproblemas, r, generalmente es chica e independiente de la instancia particular que se resuelve. Para obtener un algoritmo eficiente, la medida de los subproblemas debe ser similar y no necesitar resolver más de una vez el mismo subproblema.

Complejidad de un algoritmo Divide & Conquer :

Estos algoritmos son algoritmos recursivos, por lo que vamos a seguir el mismo razonamiento para el cálculo de su complejidad. Generalmente las instancias que son caso base toman tiempo constante c (es O(1)) y para los casos recursivos podemos identificar tres puntos críticos:

▶ Cantidad de llamadas, que llamaremos r (que no son la cantidad de llamadas totales en la funcion,si no la cantidad de llamadas por subproblema,por ej en MergeSort dividimos el arreglo en 2,log n veces **pero la cantidad de llamadas es 2,no log n.**
▶ Medida de cada subproblema, n/b para alguna constante b (en cuanto dividimos la primer instancia)
▶ Tiempo requerido por D&C para descomponer y combinar para una instancia de tamaño n, f(n)
Entonces, tiempo total T(n) consumido por el algoritmo está definido por la siguiente ecuación de recurrencia:

T(n) = c si n es caso base
rT(n/b) + f(n) si n es caso recursivo

Y esa ecuación de recurrencia se resuelve con el Teorema Maestro :

$$Si\ f(n) = O(n^{\log_c^{a-\epsilon}}) \text{ para } \epsilon > 0, \text{ entonces } T(n) = \Theta(n^{\log_c^a})$$

$$Si\ f(n) = \Theta(n^{\log_c^a}), \text{ entonces } T(n) = \Theta(n^{\log_c^a} \log n)$$

$$Si\ f(n) = \Theta(n^{\log_c^a} \log^k n) \text{ para algún } k \geq 0, \text{ entonces } T(n) = \Theta(n^{\log_c^a} \log^{k+1} n)$$

$$Si\ f(n) = \Omega(n^{\log_c^{a+\epsilon}}) \text{ para } \epsilon > 0 \text{ y } af(n/c) < kf(n) \text{ para}$$
$$k < 1 \text{ y } n \text{ suficientemente grandes, entonces } T(n) = \Theta(f(n))$$

Correctitud de algoritmos recursivos :

Para demostrar la correctitud de un algoritmo tenemos que demostrar que termina (o sea, que es un algoritmo) y que cumple la especificación.

Para demostrar que termina, tenemos que demostrar que los procesos repetitivos terminan cuando se aplican a instancias que cumplen la precondición. Para demostrar la correctitud, debemos asegurar que, si el estado inicial satisface la precondición, entonces el estado final cumplir´a la postcondición.

Grafos

Un grafo G= (V,X) es un par de conjuntos, donde V es un conjunto de puntos o nodos (o vértices) y X es un subconjunto del conjunto de pares no ordenados de elementos distintos de V. Los elementos de X se llaman aristas o ejes.

Si no aclaramos lo contrario, por lo general, en el curso llamaremos nG= ǀ V ǀ y mG= ǀ X ǀ . Cuando esté claro a qué grafo nos referimos, evitaremos el subíndice.

Dados v y w∈V, si e=(v,w)∈X se dice que v y w son adyacentes y que e es incidente a v y w. La vecindad de un vértice v,N(v), es el conjunto de los vértices adyacentes a v. Es decir:

$$N(v) = \{w \in V : (v, w) \in X\}.$$

Un multigrafo es un grafo en el que puede haber varias aristas entre el mismo par de vértices.

Un pseudografo es un grafo en el que puede haber varias aristas entre cada par de vértices y también puede haber aristas (loops) que unan a un vértice con sí mismo

El grado de un vértice v en el grafo G, dG(v) es la cantidad de aristas incidentes a v en G. Llamaremos Δ(G) al máximo grado de los vértices de G, δ(G) al mínimo.

Teorema 1 :

Dado un grafo G=(V,X), la suma de los grados de sus vértices es igual a 2 veces el número de sus aristas. Es decir,

$$\sum_{v \in V} d(v) = 2m$$

donde m= ∣ X ∣ es el número de aristas del grafo.

Corolario 1. Para todo grafo, la cantidad de vértices que tienen grado impar es par.

Definición 5. Un grafo se dice completo si todos sus vértices son adyacentes entre sí. Notaremos como Kn al grafo completo de n vértices.Tiene n(n-1)/2 aristas.

Definición 6: Dado un grafo G=(V,X), su grafo complemento, que notaremos

$$\overline{G} = (V, \overline{X})$$

tiene el mismo conjunto de vértices. Un par de vértices son adyacentes en G¯ si, y solo si, no son adyacentes en G. También puede ser notado como G^c.

¿cuántas aristas tiene G¯?
mG¯ =n(n − 1)/2− m.

Un recorrido en un grafo es una secuencia alternada de vértices y aristas

P= v0 ,e1,v1,e2...vk−1,ek,vk tal que un extremo de la arista ei es vi−1 y el otro es vi para i=1,...,k. Decimos que **P es un recorrido** entre v0 y vk.

En los grafos (no multi ni pseudo), un recorrido queda definido por la secuencia de vértices:

P= v0,v1...vk−1,vk .

Un **camino** es un recorrido que no pasa dos veces por el mismo vértice.

Una **sección** de un camino P=v0,e1,v1,e2...vk−1,ek,vk, es una subsecuencia vi ei+1,vi+1 ei+2,vj-1 ej,vj de términos consecutivos de P, y lo notamos como P vi vj.

Un **circuito** es un recorrido que empieza y termina en el mismo vértice.

Un **ciclo o circuito simple** es un circuito de 3 o más vértices que no pasa dos veces por el mismo vértice salvo el vértice donde empieza y termina.

Definición 8 :

Dado un recorrido P, su longitud, |P|, es la **cantidad de aristas** que tiene.

La **distancia entre dos vértices** v y w, d(v,w), se define como la longitud del recorrido más corto entre v y w.

Si no existe un recorrido entre v y w, se dice que d(v,w)=∞ .

Para todo vértice v, d(v,v)=0.

Proposición 1 : Si un recorrido P entre v y w tiene longitud d(v,w), entonces P es un camino.

Proposición 2 :

La función de distancia cumple las siguientes propiedades para todo u,v,w ∈V:

$$d(u,v) = 0 \; si \; y \; solo \; si \; u = v.$$
$$d(u,v) = d(v,u).$$
$$d(u,w) \leq d(u,v) + d(v,w) (desigualdad \; triangular).$$

Definición 9 :

$$Dado\ un\ grafo\ G = (V_G, X_G), un\ \textbf{subgrafo}\ de\ G\ es\ un\ grafo$$
$$H = (V_H, X_H)\ tal\ que\ V_H \subseteq V_G$$
$$y\ X_H \subseteq X_G \cap (V_H \times V_H). Lo\ notamos\ como\ H \subseteq G.$$

$$Si\ H \subseteq G\ y\ H \neq G, entonces\ H\ es\ un\ \textbf{subgrafo propio}$$
$$de\ G\ es\ decir\ H \subset G.$$

$$H\ es\ un\ \textbf{subgrafo generador}\ de\ G\ si\ H\ \subseteq G\ y\ V_G = V_H.$$

$$Un\ subgrafo\ H = (V_H, X_H))\ de\ G = (V_G, X_G)\ es\ un\ \textbf{subgrafo inducido}$$
$$si\ para\ todo\ par\ u, v \in V_H, si (u, v) \in X_G, entonces\ también\ (u, v) \in X_H.$$

Definición 10 :
Un grafo se dice
conexo si existe camino entre todo par de vértices.
Una
componente conexa de un grafo G es un subgrafo conexo maximal (no está incluido estrictamente en otro subgrafo conexo) de G.

$$Un\ subgrafo\ inducido\ de\ G = (V_G, X_G) por\ un\ conjunto\ de\ vértices\ V' \subseteq V_G se\ denota\ como\ G[V\prime].$$

La suma de grados de una componente conexa es par.

Definición 11:
Un grafo G = (V, X) se dice bipartito si existen dos subconjuntos V1, V2 del conjunto de vértices V tal que:

$$V = V1 \cup V2,\ V1 \cap V2 = \emptyset$$

y tal que todas las aristas de G tienen un extremo en V1 y otro en V2.

Un grafo bipartito con subconjuntos V1, V2, es bipartito completo si todo vértice en V1 es adyacente a todo vértice en V2

Teorema 2. Un grafo G es bipartito ⇐⇒ no tiene ciclos de longitud impar.

Definición 12. Dados dos grafos G = (V, X) y G0 = (V0, X0) se dicen isomorfos si existe una función biyectiva f : V → V0 tal que para todo v, w en V : (v, w) ∈ X si y solo si (f(v), f(w)) ∈ X0.

A la función f se la llama función de isomorfismo. Cuando G y G0 son isomorfos lo notaremos como G u G0 o, simplemente (por abuso de notación) G = G0.

Osea que si en G existe una arista u,w, en G0 existe la arista pero aplicando f a v y a w.
Y viceversa.

Proposición 3. Si dos grafos G = (V, X) y G0 = (V0, X0) son isomorfos, entonces

1. tienen el mismo número de vértices,
2. tienen el mismo número de aristas,
3. para todo k, 0 ≤ k ≤ n − 1, tienen el mismo número de vértices de grado k,
4. tienen el mismo número de componentes conexas,
5. para todo k, 1 ≤ k ≤ n − 1, tienen el mismo número de caminos simples de longitud k.

Matriz de adyacencia de un grafo

Dado un grafo G, se define su matriz de adyacencia A∈{0,1}n×n, A=[aij]

$$a_{ij} = \begin{cases} 1 & \text{si } G \text{ tiene una arista entre } v_i \text{ y } v_j, \\ 0 & \text{si no.} \end{cases}$$

Proposición 4 :Si A es la matriz de adyacencia del grafo G, entonces:

1. La suma de los elementos de la columna i de A (o fila i, dado que A es simétrica) es igual a d(vi), el grado del vértice vi.

2. Los elementos de la diagonal de A*A indican los grados de los vértices:

$$Los\ elementos\ de\ la\ diagonal\ de\ A^2\ indican\ los\ grados\ de\ los\ vértices: a_{ii}^2 = d(v_i).$$

Para los pseudógrafos, se generaliza la matriz de adyacencia A como:

$$a_{ij} = \begin{cases} \text{cantidad de aristas } (v_i, v_j) & \text{si } i \neq j, \\ \text{cantidad de loops sobre } v_i & \text{si } i = j. \end{cases}$$

Digrafos

▶ Un digrafo G = (V, X) es un par de conjuntos V y X donde V es el **conjunto de puntos, nodos o vértices** y X es un subconjunto del conjunto de los pares ordenados de elementos distintos de V. A los elementos de X los llamaremos **arcos**.

▶ Dado un arco e = (u,w) llamaremos al primer elemento, u, cola de e y al segundo elemento, w, cabeza de e.
▶
El grado de entrada din(v) de un vértice v de un digrafo es la cantidad de arcos que llegan a v. Es decir, la **cantidad de arcos** que tienen a v como cabeza.
▶
El grado de salida dout(v) de un vértice v de un digrafo es la cantidad de arcos que salen de v. Es decir, la **cantidad de arcos** que tienen a v como cola.

▶ **El grafo subyacente** de un digrafo G es el grafo Gs que resulta de remover las direcciones de sus arcos (si para un par de vértices hay arcos en ambas direcciones, sólo se coloca una arista entre ellos).

Matriz de adyacencia

La matriz de adyacencia de un digrafo G, A ∈ {0, 1} n×n, A = [aij] se define como:

$$a_{ij} = \begin{cases} 1 & \text{si } G \text{ tiene un arco entre } v_i \text{ y } v_j, \\ 0 & \text{si no.} \end{cases}$$

Proposición:
Si A es la matriz de adyacencia del digrafo G, entonces:
▶ La suma de los elementos de la fila i de A es igual a dOUT (vi).
▶ La suma de los elementos de la columna i de A es igual a dIN(vi).

Definiciones :

▶ Un recorrido/camino orientado en un grafo dirigido es una sucesión de arcos e1e2 . . . ek tal que el primer elemento del arco ei coincide con el segundo de ei−1 y el segundo elemento de ei con el primero de ei+1 i = 2, . . . , k − 1.Ej : (a.b) (b,c)(c,d)
▶ Un circuito/ciclo orientado en un grafo dirigido es un recorrido/camino orientado que comienza y termina en el mismo vértice.(a,b)(b,c)(c,a)
▶ Un digrafo se dice fuertemente conexo si para todo par de vértices u, v existen caminos orientados de u a v y de v a u.

Árboles

▶ Un **árbol es un grafo** conexo sin circuitos simples.
▶ Una arista e de G es puente si G − e tiene más componentes conexas que G.
▶ Un vértice v de G es punto de corte o punto de articulación si G − v tiene más componentes conexas que G.

Teorema: Dado un grafo G = (V, X) son equivalentes:

1. G es un árbol.
2. G es un grafo sin ciclos, pero si se agrega una arista e a G resulta un grafo con exactamente un ciclo, y ese ciclo contiene a e.
3. Existe exactamente un camino entre todo par de nodos.
4. G es conexo, pero si se quita cualquier arista a G queda un grafo no conexo (toda arista es puente).

Lema 0: Sea G = (V, X) un grafo, v,w ∈ V dos vértices diferentes y existen 2 caminos distintos entre estos dos vértices entonces hay un ciclo en G.

Lema 1: Sea G = (V, X) un grafo conexo y e ∈ X. G − e es conexo si y solo si e pertenece a un ciclo de G.

Definición :

Dado un digrafo D, un orden topológico de D es un ordenamiento v1 . . . vn de sus nodos que cumple que toda arista queda de la forma vi vj con i < j (en el ordenamiento). Es decir, damos un orden a los nodos de tal forma que las aristas apuntan de izquierda a derecha (“no hay aristas para atrás”).

Definición:

▶ Una hoja es un vértice de grado 1.

Lema 2: Todo árbol no trivial tiene al menos dos hojas.

Lema 3: Sea G = (V, X) un árbol. Entonces m = n − 1.

Corolario 1: Sea G = (V, X) sin ciclos y c componentes conexas. Entonces m = n − c.

Corolario 2: Sea G = (V, X) con c componentes conexas. Entonces m ≥ n − c.

Teorema: Dado un grafo G son equivalentes:

1. G es un árbol.
2. G es un grafo sin ciclos y m = n − 1.
3. G es conexo y m = n − 1.

Árboles enraizados:

▶ Un **árbol enraizado** es un árbol que tiene un vértice distinguido que llamamos raíz.

▶ Explícitamente queda definido un árbol dirigido.
▶ El
nivel de un vértice es la distancia de la raíz a ese vértice.
▶ La altura h de un árbol enraizado es el máximo nivel de sus vértices.
▶ Los vértices internos de un árbol enraizado son aquellos que no son ni hojas ni la raíz.
▶ Un árbol enraizado se dice m-ario si todos sus vértices internos tienen grado a lo sumo m + 1 y su raíz a lo sumo m (cada vertice tiene a lo sumo m hijos + 1 padre = grado m+1 y la raiz tiene a lo sumo m hijos y no tiene padre).

▶ Un árbol se dice **balanceado** si todas sus hojas están a nivel h o h − 1.
▶ Un árbol se dice
balanceado completo si todas sus hojas están a nivel h.
▶ Decimos que
dos vértices adyacentes tienen relación padre-hijo, siendo el padre el vértice de menor nivel.

▶Un **bosque** es un conjunto de árboles que no contienen ciclos, y cada componente dentro del bosque es un árbol que es conexo y acíclico.

Teorema:
▶ Un árbol m-ario de altura h tiene a lo sumo m*h hojas. Alcanza esta cota si es un árbol exactamente m-ario balanceado completo con h ≥ 1.
▶ Un árbol m-ario con l hojas tiene h ≥ ⌈log m l⌉.
▶ Si T es un árbol exactamente m-ario balanceado no trivial entonces h = ⌈log m l⌉.

Definición:
▶ Un árbol generador (AG) de un grafo G es un subgrafo generador (que tiene el mismo conjunto de vértices) de G que es árbol.

Teorema:
▶ Todo grafo conexo tiene (al menos) un árbol generador.
▶ G conexo. G tiene un único árbol generador ⇐⇒ G es árbol.
▶ Sea T = (V, XT) un AG de G = (V, X) y e ∈ X \ XT . Entonces T′ = T + e − f = (V, XT ∪ {e} \ {f }), con f una arista del único circuito de T + e, T′ es árbol generador de G.

Recorrido de árboles :

▶ En muchas situaciones y algoritmos, dado un árbol (grafo o digrafo), queremos recorrer sus vértices exactamente una vez.
▶ Para hacerlo de una forma ordenada y sistemática, podemos seguir dos órdenes:

▶ a lo ancho (
Breadth-First Search - BFS): se comienza por el nivel 0 (la raíz) y se visita cada vértice en un nivel antes de pasar al siguiente nivel.

Con BFS vamos a recorrer todos los nodos pero en lugar de recorrer en profundidad va recorriendo a lo ancho. Se suele implementar iterativo. La idea del algoritmo es arrancar en algún nodo y recorrer todos sus vecinos, luego los vecinos de sus vecinos...

La complejidad de BFS es O(m+n).

Nos devuelve un árbol v-geodésico, siendo v el nodo desde el cual corremos el algoritmo. Puede devolver también las distancias de v a todos en el grafo.

En el **árbol BFS**, no es necesario agregar todas las aristas del grafo original. El árbol BFS solo incluye las **aristas que forman el camino más corto** desde el nodo fuente hasta los demás nodos.

▶ en profundidad (
Depth-First Search - DFS): se comienza por la raíz y se explora cada rama lo más profundo posible antes de retroceder.

DFS es un algoritmo recursivo que sigue la idea de backtracking para poder recorrer todos los nodos. Lo podemos usar tanto para grafos dirigidos como para no dirigidos.
Vamos a recorrer en profundidad (Depth): siempre vamos hasta el final de la rama y de ahí subimos.

La complejidad de DFS es O(m + n) ya que recorro todos los nodos una sola vez y reviso las aristas también una sola vez (aunque puede que no recorra todas).

Si aplicamos **DFS** para enumerar todos los vértices de un digrafo,se pueden clasificar sus arcos en 4 tipos:
▶

tree edges: arcos que forman el bosque DFS.
▶
backward edges: van hacia un ancestro.
▶
forward edges: van hacia un descendiente.
▶
cross-edges: van hacia a otro árbol (anterior) del bosque o al otra rama (anterior) del árbol.

Para
grafos, solamente existen aristas **tree edges** y **back edges**.

Una arista es puente si al sacarla aumenta la cantidad de componentes conexas
Una back-edge nunca puede ser puente
Las aristas que son puentes son aquellas tree-edges que no tienen una back-edge que las “cubra”

Detección de ciclos:

Teorema: Dado un grafo (digrafo) G.
G tiene un ciclo (ciclo orientado) ⇔ existe un backward edge en G.

Algoritmos útiles

BFS :

```
BFS ( s , Adj ) :
| level = { s : 0 }
| parent = { s : None }
| i = 1
| frontier = [ s ] # level i-1
| while frontier :
| | next = [ ] # level i
| | for u in frontier :
| | | for v in Adj[ u ] :
| | | | if v not in level :
| | | | | level [ v ] = i
| | | | | parent [ v ] = u
| | | | | next.append( v )
| | frontier = next
| | i += 1
```

DFS:

```
//Lista de adyacencia
const int N = 10;
vector<int> ady[N]; //array de vectores (lista de ady)

bool visited[N];

void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    // process node s
    for (auto u: ady[s]) {
        dfs(u);
```

```
    }
}
```

Orden topológico :

```
Topological_sort ( G ) :
| DFS ( G )
| return invertir finish
```

Componentes conexas :

```
function contarComponentesConexas(ady):
    visitados = conjunto vacío
    contador = 0

    for cada vértice v en ady:
        if v no está en visitados:
            contador = contador + 1  // Nueva componente conexa encontrada
            DFS(v,ady)  // Ejecuta DFS para marcar todos los vértices conectados

    return contador
```

Chequeo de bipartito :

```
vector <int > color (n , -1) ;
int NOT_COLOR = -1, RED = 0, BLUE = 1;
bool IsBipartite ( int start ){
queue <int > q;
for ( node = 0; i < n; node ++) {
if ( color [ node ] == NOT_COLOR ){
color [ node ] = 0; enqueue (&q , node );
while (! isEmpty (& q)) {
int v = q . dequeue ()
for ( int u : aristas [v ]) {
if ( color [u ] == NOT_COLOR ) {
( color [v ] == RED ) ? color [ u] = BLUE : color [u ] = RED ;
q. enqueue (u);
} else {
if ( color [u ] == color [v ]) return false ;
}
}
}
}
}
return true ;
}
```

Puentes (digrafo) :

Sabemos que e es un puente si y solo si no esta cubierta por ninguna arista backedge.

```
vector <vector<int>> treeEdges ;
```

```
void dfs(int v,int padre = -1) {
    estado[v] = EMPECE_A_VER ;
    for(int u : aristas[v]){
        if(estado[u]== NO_LO_VI ){ //si no lo vi,es una tree edge porque es una arista que
        //conecta un nodo con otro que es visitado por primera vez
            treeEdges[v].pushback(u) ;
          dfs(u,v) ; //v es el padre
          }
         else if (u!=padre) { //es una backedge
             backConExtremoInferiorEn[v] ++ ;
             backConExtremoSuperiorEn[u] ++ ;
             }

    }
    estado[v] = TERMINE_DE_VER ;
    }
    // Este algoritmo guarda las backedges

    vector<int> memo ;
    int cubren(int v,int p = -1) {
        if(memo[v] != -1) {
           return memo[v]
          }
         int res = 0 ;
         res += backConExtremoInferiorEn[v] ;
         res -= backConExtremoSuperiorEn[v] ; //esto te dicen cuantas backedges "pasan" por v
         for(int hijo : treeEdges[v]) {
             if(hijo != p) {
                 res += cubren(hijo,v)
             }
             }
             memo[v] = res ;
             return res ;
             }
             int puentes = 0
             for (int i = 0,i<n,i++) {
                 if(cubren(i) == 0 ) {
                     puentes ++ ;
                     }
                     }


         Complejidad de cubren = O(n)
         complejidad de puentes = O(n)
```

Cantidad de caminos hasta un vertice :

```
cantidadDECaminosHasta(v) {
    if(distancia[v]==0) {
        return 1 ;
        }
        if(memo[v] != 1)  {
            return memo[v] ;
            }
```

```
        i intn res = 0 ;
        for (int vecino : aristas[v])   {
             if(distancia[vecino] +1 == distancia[v]) {
                 res += cantidadDeCaminosHasta(vecino) ;
                 }
            }
            memo[v] = res ;
            return res ;
            }
            Complejidad = O(n+m)
```

El **algoritmo de Dijkstra** es un algoritmo de búsqueda de caminos más cortos en un grafo ponderado dirigido o no dirigido. El objetivo del algoritmo es encontrar el **camino más corto** desde un **vértice fuente** a todos los demás vértices en el grafo, minimizando la suma de los pesos de las aristas.

- Funciona solo con **pesos no negativos** en las aristas.
- Es un algoritmo **voraz** (greedy), lo que significa que toma la mejor decisión en cada paso basándose en la información disponible en ese momento.

```
Dijkstra(Grafo G, Nodo fuente):
    1. Inicializar un arreglo `distancia[]` con infinito para todos los nodos excepto el nodo
         distancia[fuente] = 0
         para cada otro nodo v en G:
             distancia[v] = ∞

    2. Inicializar una cola de prioridad (min-heap) que contiene pares (distancia, nodo):
         insertar en la cola (0, fuente)  # (distancia desde el origen, nodo)

    3. Mientras la cola no esté vacía:
         (dist_actual, u) = extraer el nodo con la menor distancia de la cola

         para cada vecino v de u:
             w = peso de la arista (u, v)

             si distancia[u] + w < distancia[v]:
                 distancia[v] = distancia[u] + w
                 insertar (distancia[v], v) en la cola de prioridad

    4. Al finalizar, el arreglo `distancia[]` contiene la menor distancia desde la fuente a t
```

Tiene complejidad O((V+E)log V)

BFS para obtener las distancias más cortas de un vertice al resto del grafo.

```
unordered_map<long long int,long long int> bfs(long long int s, unordered_map<long long int,u

  unordered_map<long long int,long long int> distancias ;
  for(const pair<long long int,unordered_map<long long int,long long int>>& vertices : camino
        distancias[vertices.first] = INT_MAX ;
  }

  distancias[s] = 0;
  queue<long long int> enEspera ;
  enEspera.push(s) ;
  while(!enEspera.empty()) {
    long long  u = enEspera.front() ;
```

```
    enEspera.pop();
    for(const pair<long long int,long long int>& ady : caminos[u]) {
         if(distancias[ady.first]==INT_MAX){
            distancias[ady.first] = distancias[u]+1 ;

            enEspera.push(ady.first) ;
        }

    }
  }
 return distancias ;
} //O(|caminos|+|vertices|)
```