

Algoritmos III - Apuntes para final

Gianfranco Zamboni

16 de abril de 2022

Índice

1. Introducción	2
1.1. Análisis de algoritmos	2
1.1.1. Modelo Random Access Machine (RAM)	3
1.2. Cálculo de complejidad	3
1.2.1. Tamaño de una instancia	3
1.2.2. Funciones de complejidad	4
1.2.3. Problemas bien resueltos	4
1.3. Técnicas de diseño de algoritmos	5
1.3.1. Algoritmos golosos	5
1.3.2. Recursividad	6
1.3.3. Divide and Conquer	6
1.3.4. Backtracking	7
1.3.5. Programación dinámica	8
1.3.6. Algoritmos probabilísticos	8
1.3.7. Heurísticas	9
2. Grafos	10
2.1. Definiciones básicas	10
2.1.1. Caminos y ciclos	12
2.1.2. Subgrafos y Componentes Conexas	14
2.2. Grafos bipartitos	15
2.3. Representación de grafos	16
2.3.1. Matriz de adyacencia de un grafo	16
2.3.2. Matriz de incidencia de un grafo	17
2.4. Digrafos	17
A. Hoja de complejidades	19

1. Introducción

Un **algoritmo** es una secuencia **finita** de pasos **precisos** (no deben requerir tomar ninguna decisión subjetiva, ni hacer uso de la intuición o la creatividad) necesarias para llevar a cabo un cálculo de manera correcta.

Se dice que un algoritmo está **bien definido** cuando toda ejecución del mismo bajo los mismos parámetros devuelve el mismo resultado. Hay que tener en cuenta que si bien esto es lo deseable en la mayoría de los casos, muchas veces es necesario usar variables aleatorias o heurísticas para conseguir un resultado medianamente decente para problemas que no tienen solución o que son muy difíciles de calcular de manera exacta.

Pseudocódigo: Es una descripción informal de alto nivel de un algoritmo que transmite el procedimiento a realizar de forma clara y precisa.

1.1. Análisis de algoritmos

Cuando tenemos más de un algoritmo para resolver un problema y queremos elegir el *mejor* entre ellos, hay diferentes criterios que podemos analizar para medir la eficiencia de los mismos, según el contexto de aplicación. Los recursos de mayor interés suelen ser el tiempo de cómputo y el espacio de almacenamiento requerido. Podemos considerar dos enfoques:

Análisis empírico: Implementar el algoritmo en una máquina determinada utilizando un lenguaje determinado, correrlo para un conjunto de instancias y comparar sus tiempos de ejecución. Esto implicaría perder tiempo y esfuerzo programándolo, ejecutándolo y además probarlo con un conjunto de instancias acotados por lo que realmente no podríamos concluir nada sobre su comportamiento con instancias que queden fuera de ese conjunto.

Análisis teórico: Determinar matemáticamente la cantidad de tiempo que llevará su ejecución como una función de la medida de la instancia considerada, independizándolo de la máquina sobre la cuál es implementado el algoritmo y el lenguaje para hacerlo. Para esto es necesario definir un modelo de cómputo, un lenguaje sobre este modelo, cuales son las instancias del problema relevantes al mismo y sus tamaños.

Dicho de otra forma, desde el punto de vista teórico, se busca determinar la velocidad de crecimiento del tiempo o el espacio requerido por el algoritmo en función del tamaño de las instancias del problema.

Dado un problema, se le asigna un entero n (llamado el **tamaño** del problema) que servirá como medida de la cantidad de datos de entrada. Y se define como **complejidad temporal** del algoritmo al tiempo ejecución de un algoritmo expresado como una función que depende de n .

Análogamente, se puede definir la **complejidad espacial** del algoritmo como el espacio de memoria necesario para su ejecución en función del tamaño del problema.

1.1.1. Modelo Random Access Machine (RAM)

Una Random Access Machine (RAM) modela una computadora con un único registro acumulador en la cuál las instrucciones no pueden modificarse. La misma está formada por:

- Una **unidad de entrada** que contiene los datos necesarios para poder ejecutar una corrida del algoritmo. La misma es de solo lectura y, en cada una de sus celdas, puede contener un entero de tamaño arbitrario. Cuando un valor es leído, la cinta se avanza una celda.
- Una **unidad de salida** que es de solo escritura y se encuentra inicialmente vacía. Cuando una instrucción es ejecutada, se imprime un entero en la celda que se encuentra bajo la lectora y luego se la avanza una posición.
- Una **memoria** que consiste en una secuencia infinita de registros, cada uno de los cuales puede contener un entero de tamaño arbitrario.
- Un **programa** es una secuencia de instrucciones etiquetadas que no está almacenado en memoria. Asumimos que las instrucciones que contiene un programa, son las instrucciones aritméticas básicas, de entradas/salidas y direccionamiento indirecto y de branching encontradas en la mayoría de las computadoras.

Este modelo se usa cuando los tamaños de los problemas a resolver es lo suficientemente pequeño como para que entren en la memoria principal de una computadora o cuando los enteros usados en la computación son lo suficientemente chicos como para que entren en una palabra de la misma.

1.2. Cálculo de complejidad

Una operación es elemental si su tiempo de ejecución puede ser acotado por una constante dependiente sólo de la implementación particular utilizada. Esta constantes no depende de la medidad de los parámetros de la instancia considerada.

En una máquina RAM, se asume que toda instrucción es una operación elemental con un **tiempo de ejecución** asociado y definimos el tiempo de ejecución de un programa como:

$t_A(I)$ = suma de los tiempos de ejecución de las instrucciones realizadas por el programa A con la instancia I

1.2.1. Tamaño de una instancia

Para especificar la complejidad en peor caso (tanto espacial como temporal), debemos especificar el tiempo de ejecución requerido por cada instrucción y el espacio de cada registro.

La forma más simple para esto es usar el **criterio de costo uniforme** en el que cada instrucción requiere una unidad de tiempo y cada registro requiere una unidad de espacio. En general, usaremos este criterio para analizar problemas de ordenamientos o sobre grafos y matrices.

Una segunda opción (un poco más realista), es tomar en cuenta el tamaño real (en bits) de los operandos. Este el **criterio de costo logarítmico** que toma en cuenta que son necesarios para representar los datos.

Alfabeto: Es el conjunto de símbolos que puede interpretar en una máquina. Por ejemplo, las computadoras actuales reconocen el alfabeto binario $\{0, 1\}$.

Dada una instancia I , se define $|I|$ como el número de símbolos de un alfabeto finito necesarios para codificar I . Este tamaño depende del alfabeto elegido. En el caso de las computadoras actuales, que se manejan con el alfabeto binario, para almacenar un número natural n , se necesitan $L(n) = \lfloor \log_2 n \rfloor + 1$ bits.

Usaremos este criterio cuando analicemos problemas sobre números (por ejemplo, el calculo de un factorial).

1.2.2. Funciones de complejidad

Complejidad peor caso: Se toma como complejidad del algoritmo, la función que dada una instancia de tamaño de n , devuelve la máxima complejidad posible sobre todas las instancias de ese tamaño.

Complejidad caso promedio: Se toma como complejidad del algoritmo la función que dada una instancia de tamaño de n , devuelve el promedio de todas las complejidades posible sobre todas las instancias de ese tamaño.

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

- $f(n) = O(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

$$f(n) \leq cg(n) \text{ para todo } n \geq n_0$$

- $f(n) = \Omega(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

$$f(n) \geq cg(n) \text{ para todo } n \geq n_0$$

- $f(n) = \Theta(g(n))$ si

$$f(n) = O(g(n)) \text{ y } f(n) = \Omega(g(n))$$

1.2.3. Problemas bien resueltos

Decimos que un problema está **bien resuelto** si existe un algoritmo de complejidad polinomial que lo resuelva. Es decir, que no consideraremos soluciones satisfactorios a los algoritmos supra-polinomiales.

1.3. Técnicas de diseño de algoritmos

1.3.1. Algoritmos golosos

Los algoritmos golosos toman decisiones basandose en la información disponible actualmente, sin considerar los efectos que esas decisiones podrían tener en el futuro. Son fáciles de inventar, implementar y son eficientes. Sin embargo, aunque muchas veces proporcionan heurísticas sencillas para resolver problemas de optimización, no siempre funcionan.

Comunmente, los algoritmos golosos y los problemas que pueden resolver tienen las siguientes características:

- El problema se debe resolver de alguna manera óptima. Para construir su solución se mantiene **un conjunto de candidatos** entre los cuales el algoritmo puede elegir para agregar a la solución en el próximo paso. Por ejemplo, un conjunto de monedas o de nodos de un grafos.
- A medida que el algoritmo progresa, se acumulan dos conjuntos: Uno contiene candidatos que ya fueron considerados y usados; el otro contiene los candidatos que fueron considerados y rechazados.
- Hay una **función de verificación** que dado un conjunto de candidatos verifica si puede ser una solución válida al problema propuesto aunque esta no sea la solución óptima.
- Hay una **función de factibilidad** que dado un conjunto de candidatos verifica si se puede extender con nuevos candidatos para conseguir una solución al problema.
- Hay una **función de selección** que decide cual de los candidatos disponibles (que no fueron elegidos ni rechazados) es el más prometedor para acercarnos a la solución deseada.
- Hay una **función objetivo** que nos da el valor de la solución encontrada. Por ejemplo, el número de monedas utilizadas para el cambio, o la cantidad de nodos usados en un camino, o cualquier otro valor que estemos intentando optimizar. A diferencia de las otras tres funciones mencionadas anteriormente, esta no aparece explícitamente en el algoritmo goloso.

Estructura general de un algoritmo goloso:

```

function GREEDY( $C$ : Set)                                ▷  $C$  es el conjunto de candidatos
   $S \leftarrow \emptyset$                                     ▷ Conjunto en el que se construye la solución
  while  $C \neq \emptyset \wedge \neg \text{ESOLUCION}(S)$  do
     $x \leftarrow \text{SELECCIONAR}(C)$ 
     $C \leftarrow C \setminus \{x\}$ 
    if  $\text{ESVALIDO}(S \cup \{x\})$  then
       $S \leftarrow S \cup \{x\}$ 
    end if
  end while

```

```

if ESSOLUCION( $S$ ) then return  $S$ 
else
    return Error: No hay solución
end if
end function

```

1.3.2. Recursividad

Un algoritmo recursivo se define en términos de si mismo, esto es, en su cuerpo aparece una aplicación suya. En general, las llamadas recursivas se aplican sobre parámetros más pequeños que los iniciales. Cuando el parámetro sobre el que se aplica es lo suficientemente chico se ejecuta lo que se llama el **caso base** que, para su resolución, no necesita llamar a la misma función, terminando así el proceso recursivo.

En general, para calcular la complejidad de un algoritmo recursivo, se debe plantear primero las ecuaciones de recurrencia y luego, utilizando herramientas matemáticas, encontrar la fórmula cerrada (que no dependa de la complejidad de instancias más chicas) de esta ecuación.

1.3.3. Divide and Conquer

Es una técnica que consiste en dividir la instancia de un problema a resolver en varias instancias más pequeñas del mismo problema y resolverlas de manera independiente para luego combinar sus soluciones en la solución de la instancia original.

Para que valga la pena realizar un algoritmo con esta técnica se deben cumplir tres condiciones:

1. Debe ser posible descomponer las instancias en subinstancias y combinar las subsoluciones de manera eficiente.
2. Las subinstancias deben ser todas más o menos del mismo tamaño. La mayoría de los algoritmos de divide and conquer son tales que el tamaño de una subinstancia I , es aproximadamente $\frac{n}{b}$ donde n es el tamaño de la instancia original y b alguna constante.

La estructura general de un algoritmo de divide and conquer es la siguiente:

```

function DIVIDEANDCONQUER( $I$ : Instancia del problema)
    if  $I$  es suficientemente pequeño o simple then return resolver( $I$ )
    else
        Descomponer  $I$  en subinstancias  $I_1, I_2, \dots, I_k$ 
        for  $i \leftarrow 1$  to  $k$  do
             $y_i \leftarrow$  DIVIDEANDCONQUER( $I_i$ )
        end for
        Combinar las soluciones  $y_1, y_2, \dots, y_k$  obtenidas para obtener la solución  $y$  de  $I$ 
        return  $y$ 
    end if

```

end function

En este tipo de algoritmos, generalmente las instancias que son caso base toman tiempo constante c y para las casos recursivos se pueden identificar tres puntos críticos:

- Cantidad r de llamadas recursivas que haremos.
- Medida de cada subproblema: $\frac{n}{b}$ para alguna constante b .
- Tiempo requerido por el algoritmo para ejecutar, descomponer y combinar para una instancia de tamaño n , $g(n)$.

Entonces, el tiempo total $T(n)$ consumido por el algoritmo está definido por la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} c & \text{si } n \text{ es caso base} \\ rT\left(\frac{n}{b}\right) + g(n) & \text{si } n \text{ es caso recursivo} \end{cases}$$

1.3.4. Backtracking

Es una técnica que permite recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional. Cuando el algoritmo comienza, no se sabe nada sobre la solución del problema. A medida que va avanzando, se agrega un elemento a la solución para ir consiguiendo soluciones parciales. Si en algún momento, la solución parcial deja de poder ser extendida entonces esa solución se descarta.

La idea básica es tratar de extender una solución parcial del problema hasta, eventualmente, llegar a obtener una solución completa, que podría ser válida o no. Habitualmente, se utiliza un vector $a = (a_1, a_2, \dots, a_n)$ para representar una solución candidata donde cada a_i pertenece a un dominio finito A_i . El espacio de soluciones es el producto cartesiano $A_1 \times \dots \times A_n$.

En síntesis, en cada paso se extienden las soluciones parciales $a = (a_1, a_2, \dots, a_k)$ con $k < n$, agregando un elemento más al final. Las nuevas soluciones parciales son sucesoras de la anterior.

Se puede pensar este espacio de búsqueda como un árbol dirigido donde cada vértice representa una solución parcial y un vértice x es hijo de y si la solución parcial x se puede extender desde la solución parcial y . La raíz del árbol se corresponde con el vector vacío.

El proceso de backtracking recorre este árbol en profundidad. Cuando se puede deducir que una solución parcial no llevará a una solución válida, no es necesario seguir explorando esa rama del árbol y se retrocede hasta encontrar un vértice con un hijo válido por donde seguir la exploración.

El template general para algoritmos de este tipo es:

```
sol: Vector  $\leftarrow []$ 
encontre: Booleano  $\leftarrow false$ 
function BACKTRACKING( $v[1 \dots k]$ ,  $s$ : Instancia del problema)
  if  $k == 0 \wedge esSolucion(s)$  then
    sol  $\leftarrow s$ 
```

```

    encontro  $\leftarrow true$ 
else
    for  $i \leftarrow 1$  to  $k$  do
        if encontro then return
        end if
    end for
end if
end function

```

1.3.5. Programación dinámica

Esta técnica es aplicada a problemas de optimización combinatoria, donde puede haber muchas soluciones factibles, cada una con un valor (o costo asociados) y pretende obtener la solución con mejor valor (o menor costo).

Al igual que dividir y conquistar, se divide al problema en problemas que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original. Se diferencian, en que esta técnica es iterativa (no recursiva) y es adecuada cuando es necesario resolver varias subsinstancias del problema que son iguales ya que se van almacenando los resultados parciales obtenidos para su posterior utilización.

Principio de Optimalidad de Bellman: Un problema de optimización satisface este principio si en una solución óptima cada subsolución es a su vez óptima del subproblema correspondiente. Es decir, dada una secuencia óptima de decisiones, toda subsecuencia de ella es, a su vez, óptima.

Este principio es condición necesaria del problema para poder usar la técnica.

1.3.6. Algoritmos probabilísticos

Cuando un algoritmo tiene que hacer una elección, a veces es preferible elegir al azar en vez de gastar mucho tiempo tratando de ver cual es la mejor opción.

Una de las características de los algoritmos probabilísticos es que puede comportarse de maneras distintas si se lo usa para computar una misma instancia dos o más veces. Su tiempo de ejecución y resultados pueden variar considerablemente de un uso a otro pudiendo incluso nunca llegar a terminar en ciertas corridas para una instancia específica. Sin embargo, dada la posibilidad de reiniciar el algoritmo y obtener un resultado válido éste es un comportamiento que no molesta.

Por otro lado, una consecuencia de este comportamiento es que, si hay más de una solución al problema, las mismas se pueden obtener corriendo el algoritmo más de una vez.

Los algoritmos probabilísticos se pueden clasificar dependiendo de la probabilidad de que devuelvan una respuesta correcta:

- **Algoritmos numéricos:** Estos algoritmos devuelven un intervalo de confianza (del estilo “La solución es $x \pm y$ con probabilidad z ”). Por lo general, mientras más tiempo de proceso

les demos, mas preciso es el intervalo que devuelven.

- **Algoritmos de Montecarlo:** Son algoritmo que dan la respuesta exacta con alta probabilidad. Por lo general, no es posible verificar que la respuesta dada sea correcta sin embargo la probabilidad de error disminuye mientras más tiempo se este ejecutando el algoritmo.
- **Algoritmos Las Vegas:** Son algoritmos que siempre dan una respuesta correcta pero puede llegar a no dar ninguna respuesta.
- **Algoritmos Sherwood:** Son algoritmos que agregan una componente aleatoria a algoritmos determinísticos para tratar de evitar tiempos de ejecución del peor caso. Un ejemplo de esto es el Quicksort con pivote seleccionado aleatoriamente.

1.3.7. Heurísticas

Dado un problema de optimización \square difícil de resolver (y para el cual probablemente no exista un algoritmo eficiente), los **algoritmos heurísticos** definen un procedimiento que intenta conseguir soluciones para el mismo pero puede devolver resultados erróneos o no devolver nada directamente.

En otras palabras: Sea I es una instancia del problema y $x^*(I)$ el valor óptimo de la función a optimizar en dicha instancia. Buscamos crear una heurística H tal que la solución $x^H(I)$ devuelta por la misma sea lo más cercano a $x^*(I)$ posible.

Algoritmos aproximados: Decimos que H es un algoritmo ϵ -aproximado para el problema \square si para algún $\epsilon > 0$ vale que $|x^H(I) - x^*(I)| \leq \epsilon|x^*(I)|$

Algoritmos con certificados: Por lo general, si bien los problemas difíciles no pueden ser resueltos con algoritmos polinomiales si se puede verificar que las soluciones provistas por las heurísticas sean correctas con algoritmos polinomiales.

Se dice que los algoritmos diseñados para realizar esta verificación proveen un certificado que afirma la validez de la soluciones propuestas por una heurística.

2. Grafos

2.1. Definiciones básicas

Los grafos proporcionan una forma conveniente y flexible de representar problemas de la vida real que consideran una red como estructura subyacente. Esta red puede ser física (como instalaciones eléctricas) o abstractas (que modelan relaciones menos tangibles, como relaciones sociales y bases de datos).

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

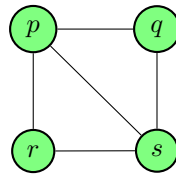


Figura 1: $G = ([p, q, r, s], [(p, q), (p, s), (q, s), (r, s)])$

Dados v y $w \in V$, si $e = (v, w) \in X$ se dice que v y w son **adyacentes** y que e es **incidente** a v y a w .

La definición de grafo no alcanza para modelar todas las situaciones posibles de una red. Por ejemplo, si se quisiese modelar la ruta de los aviones entre varias ciudades, deberíamos poder modelar varios vuelos entre dos ciudades:

Multigrafo: Es un grafo en el que puede haber varias aristas entre el mismo par de nodos distintos.



Figura 2: Multigrafo

Seudografo: Es un grafo en el que puede haber varias aristas entre cada par de nodos y también puede haber aristas (*loops*) que unan a un nodo con sí mismo.



Figura 3: Multigrafo

Notación: $n = |V|$ y $m = |X|$

Grado: El grado $d_G(v)$ de un nodo v es la cantidad de aristas incidentes a v en el grafo G .

Notaremos $\Delta(G)$ al máximo grado de los vértices de G y $\delta(G)$ al mínimo.

Nota: En un pseudografo, un loop aporta 2 al grado del vértice.

Teorema 1. *La suma de los grados de los nodos de un grafo es igual a dos veces el número de aristas, es decir:*

$$\sum_{i=1}^n v_i = 2m$$

DEMOSTRACIÓN

Sea $G = (V, X)$ un grafo de n nodos y m aristas. Haremos inducción en la cantidad de aristas:

Caso base: $m = 1$.

En este caso, el grafo G tiene sólo una arista que notamos $e = (u, w)$. Entonces $d(u) = d(w) = 1$ y $d(v) = 0$ para todo $v \in V$ tal que $v \neq u, w$. Por lo tanto,

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

y $2m = 2$, cumpliéndose la propiedad.

Paso inductivo: Consideremos un grafo $G = (V, X)$ con m aristas ($m > 1$). Nuestra hipótesis inductiva es: *En todo grafo $G' = (V', X')$ con m' aristas ($m' < m$), se cumple que*

$$\sum_{v \in V'} d_{G'}(v) = 2m'$$

Sea $e = (u, w) \in X$, llamemos $G'' = (V, X - \{e\})$ al grafo que resulta si le quitamos e a G . Como la cantidad de aristas de G'' es $m - 1$, G'' cumple con la hipótesis inductiva. Entonces vale que

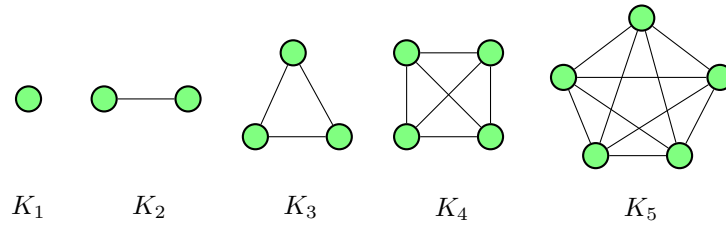
$$\sum_{v \in V} d_{G''}(v) = 2(m - 1)$$

Como $d_G(u) = d_{G''}(u) + 1$, $d_G(w) = d_{G''}(w) + 1$ y $d_G(x) = d_{G''}(x) \forall x \in V, x \neq u, w$, obtenemos que:

$$\sum_{v \in V} d_G(v) = \sum_{v \in V} d_{G''}(v) + 2 = 2(m - 1) + 2 = 2m \blacksquare$$

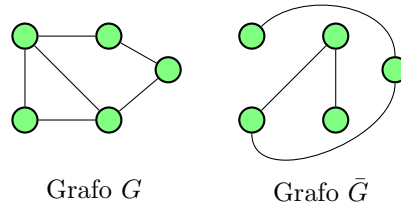
Corolario 1. *La cantidad de vértices de grado impar de un grafo es par.*

Grafo completo: Son grafos en los que todos sus vértices son adyacentes entre sí. Notaremos como K_n al grafo completo de n vértices.



Propiedad: Un grafo completo de n nodos tiene $\frac{n(n-1)}{2}$ aristas

Grafo complemento: Dado un grafo $G = (V, X)$, su grafo complemento $\bar{G} = (V, \bar{X})$ es el grafo con el mismo conjunto de vértices pero un par de vértices son adyacentes en \bar{G} si y solo si no son adyacentes en G .



Propiedad: Si G tiene n vértices y m aristas, entonces:

$$m_{\bar{G}} = \frac{n(n-1)}{2} - m$$

2.1.1. Caminos y ciclos

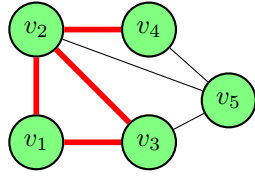
Camino: Un camino en un grafo, es una secuencia alternada de vértices y aristas $P = v_0 e_1 v_1 e_2 \dots v_{k-1} e_k v_k$ tal que un extremo de la arista e_i es v_{i-1} y el otro es v_i para $i = 1 \dots k$.

Camino simple: Es un camino que no pasa dos veces por el mismo vértice.

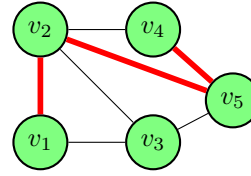
Sección: La sección de un camino $P = v_0 e_1 v_1 e_2 \dots v_{k-1} e_k v_k$ es una subsecuencia $v_i e_{i+1} v_{i+1} e_{i+2} \dots v_{j-1} e_j v_j$ de términos consecutivos de P , y lo notamos como $P_{v_i v_j}$.

Circuito: Es un camino que empieza y termina en el mismo vértice.

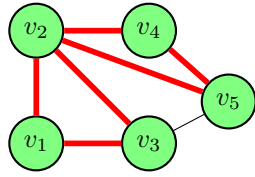
Circuito Simple: Es un circuito de tres o más vértices que no pasa dos veces por el mismo vértice.



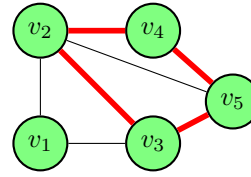
Camino no simple
 $P = v_2v_3v_1v_2v_4$



Camino simple
 $P = v_1v_2v_5v_4$



Circuito no simple
 $P = v_1v_3v_2v_4v_5v_2v_1$



Circuito simple
 $P = v_2v_3v_5v_4v_2$

Longitud de un camino: Dado un camino P , su longitud $l(P)$ es la cantidad de aristas que tiene.

Distancia: La distancia $d(v, w)$ entre dos vértices v y w se define como la longitud del camino más corto entre v y w .

- Si no existe camino entre v y w decimos que $d(v, w) = \infty$.
- $\forall v \in V, d(v, v) = 0$

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

DEMOSTRACIÓN

Demostración por el absurdo. Sea $P = v \dots w$ un camino entre v y w con $l(P) = d(v, w)$. Supongamos que P no es simple, es decir existe un vértice u que se repite en P (u podría llegar a ser v o w) entonces $P = v \dots u \dots u \dots w$.

Formemos ahora un camino $P' = P_{vu}P_{uw}$, como P' no tiene todos los nodos que están en P_{uu} nos queda que $l(P') < l(P) = d(v, w)$. Esto genera un absurdo porque por definición $d(v, w)$ es la longitud del camino más corto entre v y w . ■

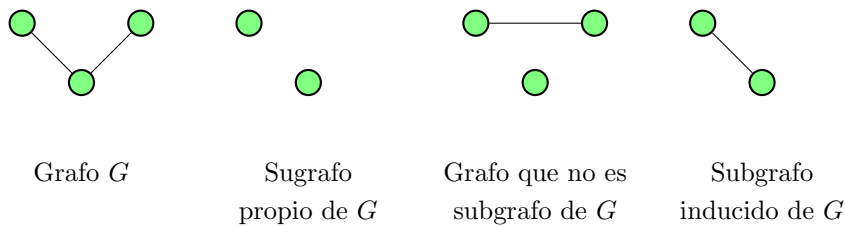
Proposición 2. La función de distancia cumple las siguientes propiedades para todo u, v, w pertenecientes a V :

1. $d(u, v) \geq 0$
2. $d(u, v) = 0 \iff u = v$
3. $d(u, v) = d(v, u)$
4. $d(u, w) \leq d(u, v) + d(v, w)$

2.1.2. Subgrafos y Componentes Conexas

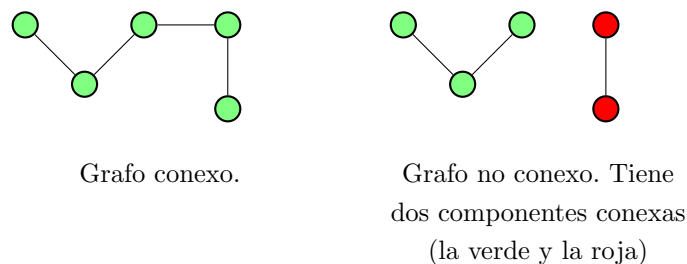
Subgrafo: Dado un grafo $G = (V_G, X_G)$, un subgrafo de G es un grafo $H = (V_H, X_H)$ tal que $V_H \subseteq V_G$ y $X_H \subseteq X_G \cap (V_H \times V_H)$. Y notamos $H \subseteq G$.

- Si $H \subseteq G$ y $H \neq G$, entonces H es un **subgrafo propio** de G y notamos $H \subset G$.
- H es un 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 **subgrafo inducido** si $\forall u, v \in V_H$ tal que $(u, v) \in X_G \Rightarrow (u, v) \in X_H$.
- Un subgrafo inducido de $G = (V_G, X_G)$ por un conjunto de vértices $V' \subseteq V_G$, se denota como $G_{[V']}$.



Grafo conexo: Un grafo se dice **conexo** si existe camino entre todo par de vértices.

Componente conexa: Una componente conexa de un grafo $G = (V_G, X_G)$ es un subgrafo conexo maximal de G . Esto es un subgrafo $H = (V_H, X_H)$ inducido de G tal que H es conexo y si tratamos de agregar cualquier vértice $v \in V_G \setminus V_H$ entonces nos queda un grafo no conexo.

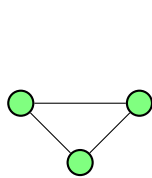


2.2. Grafos bipartitos

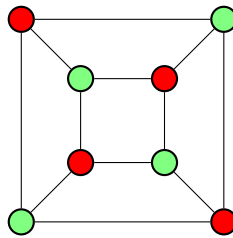
Un grafo $G = (V, X)$ se dice **bipartito** si existe una partición V_1, V_2 del conjunto de vértices V tal que:

- $V = V_1 \cup V_2$
- $V_1 \cap V_2 \neq \emptyset$
- $V_1 \neq \emptyset$
- $V_2 \neq \emptyset$
- Todas las aristas de G tiene un extremo en V_1 y otro en V_2

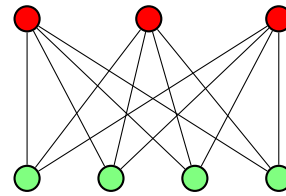
Un grafo bipartito con partición V_1, V_2 es **bipartito completo** si todo vértice en V_1 es adyacente a todo vértice en V_2 .



Grafo no bipartito.



Grafo bipartito.



Grafo bipartito completo.

Teorema 2. *Un grafo G con dos o más vértices es bipartito si y solo si no tiene circuitos de longitud impar.*

DEMOSTRACIÓN

Como un grafo es bipartito si y solo si cada una de sus componentes conexas es bipartita alcanza con demostrar el teorema para grafos conexos.

\Rightarrow) Sea G un grafo conexo bipartito y $V = (V_1, V_2)$ su bipartición. Si G no tiene circuitos entonces el teorema se cumple de manera trivial. Supongamos que G tiene circuitos y sea $C = v_1 v_2 \dots v_k v_1$ un circuito de G . Sin pérdida de generalidad, supongamos que $v_1 \in V_1$. Como $(v_1, v_2) \in X$ entonces $v_2 \in V_2$. En general, $v_{2i+1} \in V_1$ y $v_{2i} \in V_2$. Como $v_1 \in V_1$ y $(v_k, v_1) \in X$, debe pasar $v_k \in V_2$. Luego $k = 2i$ para algún i , lo que implica que $l(C)$ es par. ■

\Leftarrow) Sea G un grafo conexo sin circuitos impares. Sea u cualquier vértice de V . Definimos:

$$V_1 = \{v \in V / d(u, v) \text{ es par}\}$$

$$V_2 = \{v \in V / d(u, v) \text{ es impar}\}$$

V_1 y V_2 definen una partición de V (ya que como G es conexo no hay vértices a distancia ∞ de v). Tenemos que ver que definen una bipartición, es decir que no existe arista entre dos vértice de V_1 y dos de V_2 . Hagamos esto por el absurdo:

Supongamos que no es bipartición, es decir existen $v, w \in V_1$ tales que $(v, w) \in E$. Si $v = u$, entonces $d(v, w) = 1$, pero esto no puede pasar por que $d(u, w)$ es par. Lo mismo para w , luego $v \neq u, w$.

Sea P un camino mínimo entre v y u y Q un camino mínimo entre v y w . Como $u, w \in V_1$, P y Q tienen longitud par.

Sea $z \in V$ el último nodo en el que P y Q se cruzan (podría pasar que $z = u$). Como P y Q definen las distancias a v y w respectivamente desde u , entonces P_{uz} y Q_{uz} tienen que ser caminos mínimos. Osea que $l(P_{uz}) = l(Q_{uz}) = d(u, z)$.

Entonces $l(P_{zv})$ y $l(Q_{zw})$ tienen igual paridad. Definamos $C = P_{zv}(v, w)Q_{wz}$, entonces $l(C) = l(P_{zv}) + l(Q_{wz}) + 1$ que es una longitud impar. Absurdo, partiamos de la supocisión de que G no tenia circuitos de longitud impar ■

2.3. Representación de grafos

2.3.1. Matriz de adyacencia de un grafo

Dado un grafo G , se define su **matriz de adyacencia** $A \in \mathbb{R}^{n \times n}$, $A = [a_{ij}]$ como:

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

Proposición 3. Si A es la matriz de adyacencia del grafo G , entonces:

- La suma de los elementos de la columna (o fila) i de A es igual a $d(v_i)$.
- Los elementos de la diagonal de A^2 indican los grados de los vértices: $a_{ii}^2 = d(v_i)$.

Para los pseudografos, se generaliza la definición dada de la siguiente manera:

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

2.3.2. Matriz de incidencia de un grafo

Dado un grafo G , se define su **matriz de incidencia** $B \in \mathbb{R}^{m \times n}$ con $B = [b_{ij}]$ como:

$$b_{ij} = \begin{cases} 1 & \text{si la arista } i \text{ es incidente al v\u00e9rtice } v_j \\ 0 & \text{sino} \end{cases}$$

Proposici\u00f3n 4. Si B es la matriz de incidencia del grafo G , entonces:

- La suma de los elementos de cada fila es igual a 2.
- La suma de los elementos de la j -\u00e9sima columna es igual a $d(v_j)$.

Para los pseudografos, se generaliza la definici\u00f3n de la siguiente forma:

$$b_{ij} = \begin{cases} 2 & \text{si la arista } i \text{ es loop sobre el v\u00e9rtice } v_j \\ 1 & \text{si la arista } i \text{ no es loop e incide sobre el v\u00e9rtice } v_j \\ 0 & \text{sino} \end{cases}$$

2.4. Digrafos

Grafo dirigido o digrafo: Es un par de conjuntos $G = (V, X)$ donde V es el conjunto de nodos y X es un subconjunto del conjunto de 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 (w), **cabeza** de e .

Grado de entrada: $d_{in}(v)$ de un v\u00e9rtice v de un digrafo es la cantidad de arcos que llegan a v . Es decir, la cantidad de arcos que tienen como cabeza a v .

Grado de salida: $d_{out}(v)$ de un nodo v de un digrafo es la cantidad de arcos que salen de v . Es decir, la cantidad de arcos que tiene a v como cola.

Grafo subyacente: El grafo subyacente de un digrafo G es el grafo G^S que resulta de remover las direcciones de sus arcos (si para un par de v\u00e9rtices hay arcos en ambas direcciones, s\u00f3lo se coloca una arista entre ellos).

Matriz de adyacencia: de un digrafo G , $A \in \mathbb{R}^{n \times n}$, $A = [a_{ij}]$ se define como:

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

Proposici\u00f3n 5. 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 $d_{out}(v_i)$.
- La suma de los elementos de la columna i de A es igual a $d_{in}(v_i)$.

Matriz de incidencia: de un digrafo G , $B \in \mathbb{R}^{m \times n}$, $B = [b_{ij}]$ se define como:

$$b_{ij} = \begin{cases} 1 & \text{si } v_j \text{ es cabeza del arco } i \\ -1 & \text{si } v_j \text{ es cola del arco } i \\ 0 & \text{si no} \end{cases}$$

Proposición 6. *Si B es la matriz de incidencia del digrafo G , entonces la suma de los elementos de cada fila es igual a cero.*

Camino orientado: Es una sucesión de arcos $e_1 e_2 \dots e_k$ tal que el primer elemento del arco e_i coincide con el segundo de e_{i-1} y el segundo elemento de e_i con el primero de e_{i+1} con $i = 2, \dots, k-1$

Digrafo fuertemente conexo: Es un digrafo G tal que para todo par de vértices $u, v \in V_G$ existe un camino orientado de u a v .

A. Hoja de complejidades

Algoritmo	Complejidad
De Búsqueda	
Secuencial	$O(n)$
Binaria	$O(\log n)$
De Ordenamiento	
Bubblesort	$O(n^2)$
Quicksort	$O(n^2)$
Heapsort	$O(n \log n)^*$

* $O(n \log n)$ es la complejidad óptima para algoritmos de ordenamiento basados en comparaciones.

Grafos Definiciones básicas: isomorfismos. Árboles: caracterización, árboles orientados, árbol generador. Enumeración. Grafos eulerianos y hamiltonianos. Planaridad. Coloreo. Número cromático. Matching, conjunto independiente, recubrimiento. Recubrimiento de aristas y vértices.

Algoritmos en grafos y aplicaciones Representación de un grafo en la computadora: matrices de incidencia y adyacencia, listas. Algoritmos de búsqueda en grafos: BFS, DFS, A*. Mínimo árbol generador, algoritmos de Prim y Kruskal. Árboles ordenados: códigos unívocamente descifrables. Algoritmos para detección de circuitos. Algoritmos para encontrar el camino mínimo en un grafo: Dijkstra, Ford, Dantzig. Planificación de procesos: PERT/CPM. Algoritmos heurísticos: ejemplos. Nociones de evaluación de heurísticas y de técnicas metaheurísticas. Algoritmos aproximados. Heurísticas para el problema del viajante de comercio. Algoritmos para detectar planaridad. Algoritmos para coloreo de grafos. Algoritmos para encontrar el flujo máximo en una red: Ford y Fulkerson. Matching: algoritmos para correspondencias máximas en grafos bipartitos. Otras aplicaciones.

Problemas NP-completos Problemas tratables e intratables. Problemas de decisión. P y NP. Maquinas de Turing no determinísticas. Problemas NP-completos. Relación entre P y NP. Problemas de grafos NP-completos: coloreo de grafos, grafos hamiltonianos, recubrimiento mínimo de las aristas, corte máximo, etc.