

Resumen para Algoritmos y Estructuras de Datos III

Basado en [esta guía de estudio](#).

Tomás Spognardi

4 de agosto de 2022

Índice general

1. Técnicas de Diseño de Algoritmos	3
1.1. Complejidad	3
1.1.1. Repaso de complejidad computacional	3
1.1.2. Notación O	5
1.1.3. Problemas “bien resueltos” e intractabilidad	6
1.2. Backtracking	7
1.2.1. Fuerza bruta	7
1.2.2. Backtracking	7
1.3. Programación Dinámica	9
1.3.1. Definición	9
1.3.2. Principio de optimalidad de Bellman	11
1.4. Algoritmos Golosos	13
1.4.1. Heurísticas	13
1.4.2. Algoritmos golosos	14
1.5. Algoritmos Probabilísticos	15
1.5.1. Algoritmos numéricos	16
1.5.2. Algoritmos de Monte Carlo	16
1.5.3. Algoritmos de Las Vegas	16
1.5.4. Algoritmos de Sherwood	16
2. Introducción a Teoría de Grafos	17
2.1. Grafos	17
2.1.1. Definición	17
2.1.2. Vecinos	18
2.1.3. Generalizaciones	19
2.1.4. Recorridos	19
2.1.5. Distancia	20
2.1.6. Subgrafos	21
2.1.7. Conectividad	21
2.1.8. Representación de Grafos	21
2.1.9. Isomorfismo	22
2.1.10. Definiciones en digrafos	23
2.2. Grafos Bipartitos	23
2.3. Árboles	24
2.3.1. Definición	24
2.3.2. Árboles enraizados	25

2.3.3.	Representación de árboles	26
2.3.4.	Árbol generador	26
2.4.	Recorridos	26
2.4.1.	BFS	27
2.4.2.	DFS	28
2.5.	Orden Topológico	31
2.5.1.	Definición	31
2.5.2.	Algoritmo	31
2.6.	Algoritmo de Korasaju	31

Capítulo 1

Técnicas de Diseño de Algoritmos

1.1. Complejidad

1.1.1. Repaso de complejidad computacional

La complejidad computacional es una técnica de análisis de algoritmos, en particular, de su tiempo de ejecución. Es de carácter *teórico*: se basa en determinar matemáticamente la cantidad de operaciones que llevará a cabo el algoritmo para una instancia de tamaño dado, independientemente de la máquina sobre la cuál se implementa y el lenguaje utilizado.

Definición informal

La complejidad de un algoritmo es una función $T_A : \mathbb{N} \rightarrow \mathbb{R}_{>0}$ que representa el tiempo de ejecución en función del tamaño de la entrada. Para distinguir entre entradas de un mismo tamaño, se pueden considerar:

- Complejidad de peor caso:¹

$$T_{\text{peor}}(n) = \max \{t_A(i) \mid I \in I_A, |I| = n\}$$

- Complejidad de mejor caso:

$$T_{\text{mejor}}(n) = \min \{t_A(i) \mid I \in I_A, |I| = n\}$$

- Complejidad de caso promedio:²

$$T_{\text{prom}}(n) = \sum_{I \in I_A, |I|=n} P(I) \cdot t_A(I)$$

Para ciertos algoritmos, conviene hacer un análisis más profundo que distingue entre tipos de instancias particulares al problema.

¹ $t_A : I_A \rightarrow \mathbb{R}_{>0}$ devuelve el tiempo de ejecución para una instancia particular del problema A .

² $P(I)$ es la probabilidad de que la entrada sea la instancia I .

Estas definiciones no son rigurosas: no contienen ninguna indicación sobre cómo determinar T_A para un algoritmo A , y el “tiempo de ejecución” ni siquiera tiene unidades de medida. Para formalizarlas, es necesario definir un *modelo de cómputo*.

Modelo de cómputo: Máquina RAM

La *Máquina RAM* es una máquina abstracta que funciona como modelo de cómputo. Nos permite modelar computadoras en las que la memoria es suficiente y los enteros involucrados en los cálculos entran en una palabra³. Este modelo cuenta con:

- **Memoria Principal:** Una sucesión de celdas numeradas (tantas como se necesiten). Cada una puede guardar un entero de tamaño arbitrario.
- **Registro Acumulador:** un registro especial se usa como (generalmente primer) operando en las operaciones.
- **Acceso Aleatorio:** Acceso directo a cualquier celda en tiempo constante. También cuenta con direccionamiento indirecto: la dirección accedida puede ser el valor de una celda (un *puntero*).
- **Programa:** Se codifica en una serie de instrucciones, y se almacena en una memoria aparte de la principal. Hay un *contador de programa*, que identifica la próxima a ser ejecutada y puede ser manipulado a través de ciertas instrucciones (*jumps*).

Tanto la entrada como la salida son representadas como una sucesión de celdas numeradas, cada una con un entero de tamaño arbitrario. Para codificar un programa, es necesario definir un *set de instrucciones*. Un ejemplo posible sería el siguiente⁴:

- **LOAD valor** – Carga un valor en el acumulador.
- **STORE valor** – Carga el acumulador en un registro.
- **ADD valor** – Suma el operando al acumulador
- **SUB valor** – Resta el operando al acumulador
- **MULT valor** – Multiplica el operando por el acumulador
- **DIV valor** – Divide el acumulador por el operando
- **READ valor** – Lee un nuevo dato de entrada → operando
- **WRITE valor** – Escribe el operando a la salida
- **JUMP label** – Salto incondicional
- **JGTZ label** – Salta si el acumulador es positivo
- **JZERO label** – Salta si el acumulador es cero

³Una *palabra* es el tamaño de una celda de memoria

⁴Este ejemplo no es de ninguna forma minimal, pero es similar a un set de instrucciones RISC para una computadora real.

- **HALT** – Termina el programa

Para calcular la complejidad de un programa, se asume que cada instrucción tiene un tiempo de ejecución constante. En ese caso, se puede definir $t_A(I)$ = suma de los tiempos de ejecución de las instrucciones ejecutadas por el algoritmo A para la instancia I . Esto es casi suficiente para calcular $T_A(n)$: solo resta definir $|I|$, el tamaño de la instancia.

Modelo uniforme

En este modelo, cada **dato individual** ocupa una celda de memoria, y cada operación básica tiene tiempo de ejecución constante. Esto resulta razonable cuando la entrada es una estructura de datos y cada dato entra en una palabra de memoria. Bajo esta suposición, el tamaño de entrada se define como la cantidad de datos individuales de la instancia.

Sin embargo, para algoritmos que operan sobre un entero particular, esta definición no resulta adecuada. Por ejemplo, se puede tomar el siguiente algoritmo, que determina si un número es o no primo:

```
ES-PRIMO( $n$ )
1  for  $i = 2$  to  $\lceil \sqrt{n} \rceil$ 
2      if  $n \equiv 0 \pmod i$ 
3          return FALSE
4  return TRUE
```

Según la definición anterior, el tamaño de la entrada de ES-PRIMO es siempre 1, lo cual es anti-intuitivo: sería conveniente poder definir la complejidad de este algoritmo en función del tamaño de n .

Modelo logarítmico

En este caso, el tamaño de la instancia se define como la cantidad de símbolos de un **alfabeto** necesaria para representarla, y el tiempo de ejecución de cada operación elemental depende del tamaño de los operandos (definido de la misma manera). Esto es apropiado para algoritmos que toman como input un número fijo de datos individuales.

Para representar los datos, se suele tomar como alfabeto $\mathbb{B} = \{0, 1\}$, los dígitos binarios. En tal caso, el tamaño de un entero $n \in \mathbb{Z}$ es $L(n) = \lceil \log_2 n \rceil + 1$ bits, mientras que para almacenar una lista de m enteros se necesitan $L(m) + mL(N)$, donde N es el valor máximo posible en la lista.

1.1.2. Notación \mathcal{O}

Para comparar tiempos de ejecución entre distintos algoritmos, es conveniente obviar constantes de proporcionalidad y enfocarse en el comportamiento asintótico de las complejidades. Con eso en mente, se definen las clases:

$$\begin{aligned} f \in \mathcal{O}(g) &\iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N} \mid \forall n \geq n_0, f(n) \leq c \cdot g(n) \\ f \in \Omega(g) &\iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N} \mid \forall n \geq n_0, f(n) \geq c \cdot g(n) \\ f \in \Theta(g) &\iff f \in \mathcal{O}(g) \wedge f \in \Omega(g) \end{aligned}$$

Informalmente, $f \in \mathcal{O}(g)$ implica que f crece a lo sumo tan rápido como g .

Complejidades comunes

- Si un algoritmo es $\mathcal{O}(\log n)$, se dice **logarítmico**.
- Si un algoritmo es $\mathcal{O}(n)$, se dice **lineal**.
- Si un algoritmo es $\mathcal{O}(n^2)$, se dice **cuadrático**.
- Si un algoritmo es $\mathcal{O}(n^3)$, se dice **cúbico**.
- Si un algoritmo es $\mathcal{O}(n^k)$, se dice **polinomial**.
- Si un algoritmo es $\mathcal{O}(k^n)$ ($k > 1$), se dice **exponencial**.

Además, se tiene:

$$\mathcal{O}(n^k) \subsetneq \mathcal{O}(d^n) \quad \forall k, d \in \mathbb{N}$$

$$\mathcal{O}(\log n) \subsetneq \mathcal{O}(n^k) \quad \forall k \in \mathbb{R}_{>0}$$

1.1.3. Problemas “bien resueltos” e intractabilidad

Un problema se denomina *bien resuelto* si existe un algoritmo de tiempo polinomial que lo resuelve. Esto se debe a que el tiempo de ejecución de los algoritmos exponenciales crece demasiado rápido: su ejecución puede resultar infactible para valores de n pequeños.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$\mathcal{O}(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$\mathcal{O}(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$\mathcal{O}(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg
$\mathcal{O}(n^5)$	0.10 sg	3.20 sg	24.30 sg	1.70 min	5.20 min
$\mathcal{O}(2^n)$	1.00 ms	1.00 sg	17.90 min	12 días	35 años
$\mathcal{O}(3^n)$	0.59 sg	58 min	6 años	3855 siglos	2×10^8 siglos!

Tabla de comparaciones de los posibles tiempos de ejecución para distintas clases de complejidad.

Sin embargo, cabe destacar que:

- Si los tamaños de instancias no son muy grandes, un algoritmo exponencial puede ser apropiados.
- Un algoritmo puede ser polinomial, pero con un exponente o una constante demasiado grande para ser aplicado en la práctica.
- Existen ciertos algoritmos con complejidad de peor caso exponencial, pero que en la práctica son muy eficientes (como el método *simplex*).

1.2. Backtracking

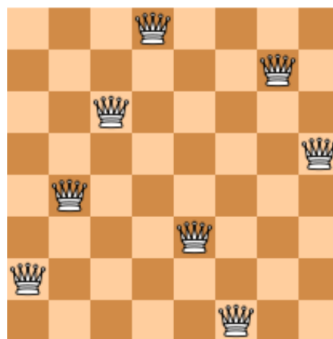
1.2.1. Fuerza bruta

Un algoritmo de *fuerza bruta* (también llamado de *búsqueda exhaustiva*) analiza todas las posibles configuraciones de la salida, hasta encontrar una que cumple con los requerimientos del problema.

Ejemplo: Problema de las n damas

Problema:

Ubicar n damas en un tablero de ajedrez de $n \times n$ casillas, de forma que ninguna dama amenace a otra.



Solución posible para el caso $n = 8$

Un posible algoritmo de fuerza bruta sería recorrer todos los posibles subconjuntos de n casillas, verificando si algún par de reinas se amenaza en caso de ser ubicarlas en las casillas del subconjunto. Sin embargo, esto no es muy eficiente: para $n = 8$, implicaría recorrer $\binom{64}{8} = 4,426,165,368$ combinaciones.

Se pueden lograr mejoras aprovechando la estructura del problema: como cada columna debe tener exactamente 1 dama, las configuraciones exploradas pueden representarse como un vector (a_1, \dots, a_n) , con $a_i \in \{1, \dots, n\}$ indicando la fila de la dama que está en la columna i . Además, cada fila tiene exactamente una reina, así que los elementos del vector no se repiten. Por ende, la cantidad de combinaciones se reduce a $n!$, que para el caso $n = 8$ es $8! = 40,320$. No obstante, esto puede mejorarse.

1.2.2. Backtracking

El backtracking es una técnica general de diseño de algoritmos que consiste de extender las soluciones parciales $a = (a_1, \dots, a_k)$, $k < n$, agregando un elemento a_{k+1} al final del mismo. Si se detecta que S_{k+1} , el conjunto de soluciones que tienen al vector como prefijo, es vacío, se retrocede a la solución anterior. Esto permite descartar configuraciones parciales apenas se determina que no pueden llevar a una solución. Los algoritmos de backtracking siguen el siguiente esquema general:


```

BT( $a$ )
1  if  $\neg$ ES-VÁLIDA( $a$ )
2      return
3  if ES-SOLUCIÓN( $a$ )
4      PROCESAR( $a$ )
5      return
6  for  $a' \in$  SUCESORES( $a$ )
7      BT( $a'$ )

```

Si solo se busca una solución, esto se puede volver más eficiente usando una variable global *encontró*:

```

BT( $a$ )
1  if  $\neg$ ES-VÁLIDA( $a$ )
2      return
3  if ES-SOLUCIÓN( $a$ )
4       $sol = a$ 
5       $encontró = \text{TRUE}$ 
6      return
7  for  $a' \in$  SUCESORES( $a$ )
8      BT( $a'$ )
9      if  $encontró$ 
10         return

```

Para que el backtracking sea eficiente, el procedimiento ES-VÁLIDA debe ser capaz de identificar algún conjunto de instancias inválidas, y no puede tener una complejidad demasiado grande.

En el problema de las n damas, se puede chequear en cada paso si alguna reina amenaza a la recién agregada. Esto se puede realizar en tiempo lineal, y por la construcción de las soluciones solo hace falta comprobar las amenazas diagonales. Utilizando este algoritmo, cualquier configuración de n elementos que no haya sido rechazada es una solución válida.

Ejemplo: Resolución de Sudokus

Problema:

Encontrar una asignación de números a casillas que resuelve un Sudoku particular.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Ejemplo de un Sudoku y su solución.

Los Sudokus pueden ser resueltos con un algoritmo de backtracking: las soluciones son extendidas agregando un número a algún casillero vacío. Cuando el nuevo número no cumple alguna de las restricciones del sudoku, la solución es rechazada. A pesar de ser exponencial, este algoritmo es muy eficiente en la práctica.

1.3. Programación Dinámica

1.3.1. Definición

La *programación dinámica* (PD/DP) es otra técnica de diseño de algoritmos. Es similar al *Divide & Conquer*, ya que se basa en dividir el problema en sub-problemas de menor tamaño, resolverlos recursivamente, y combinar las sub-soluciones para formar una solución. La diferencia con este método es que PD se utiliza en casos donde estos sub-problemas suelen superponerse, y aprovecha este hecho al resolverlos una única vez.

Para evitar repetir la resolución de sub-problemas equivalentes, los algoritmos de programación dinámica siguen alguno de estos dos esquemas:

- **Enfoque “top-down”:** Se implementa el algoritmo tradicionalmente, pero los resultados se guardan en una estructura de datos indexada por los parámetros de la llamada (*memoización*). Luego, antes de resolver ejecutar el algoritmo para una llamada, se chequea si sus parámetros están en esta estructura, y en tal caso se devuelve la solución previamente calculada.
- **Enfoque “bottom-up”:** Se resuelven los sub-problemas en un orden que asegura que las llamadas recursivas de cada uno son calculadas antes que este⁵, guardando los resultados de cada llamada en una tabla.

⁵Esto representa un ordenamiento topológico del árbol de llamadas de la función (en realidad, del árbol invertido, donde cada nodo tiene una arista apuntando hacia aquellos que lo tienen como sub-problema).

Ejemplo: Cálculo de coeficientes binomiales

Problema:

Calcular el valor del coeficiente $\binom{n}{k}$, definido como:

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

El problema se podría resolver calculando $\binom{n}{k}$ directamente, pero esto se dificulta para valores grandes. Por ejemplo, a pesar de que $\binom{100}{99} = 100$, el valor $100!$ es un número de 157 cifras, muy por encima del límite de 64 bits utilizados para representar enteros.

Una forma alternativa de realizar la operación sería haciendo uso del siguiente teorema:

Teorema. Si $n \geq 0$ y $0 \leq k \leq n$, entonces:

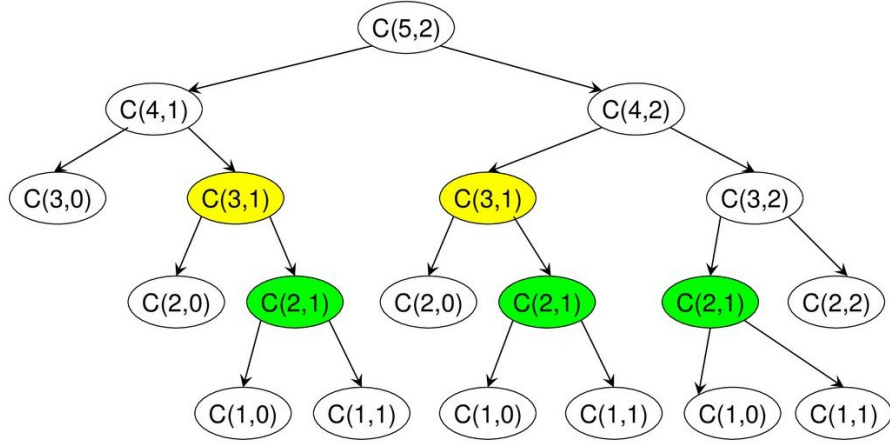
$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \end{cases}$$

Esta fórmula recursiva se puede implementar directamente:

COMBINATORIO(n, k)

```
1  if  $k == 0 \vee k == n$ 
2      return 1
3  else
4      return COMBINATORIO( $n - 1, k - 1$ ) + COMBINATORIO( $n - 1, k$ )
```

Este método tiene una complejidad de $\Omega(\binom{n}{k})$, y evita calcular factoriales, pero podría ser más eficiente, ya que al ejecutarlo se repiten llamadas con los mismos parámetros.



Árbol de llamadas de COMBINATORIO para la instancia
 $n = 5, k = 2$

Acá es donde entra en juego la programación dinámica. El siguiente algoritmo bottom-up calcula una única vez cada coeficiente necesario:

COMBINATORIO-PD(n, k)

```

1  Inicializar matriz  $A \in \mathbb{N}^{n \times k}$ 
2  for  $i = 1$  to  $n$ 
3       $A[i][0] = 1$ 
4  for  $j = 0$  to  $k$ 
5       $A[j][j] = 1$ 
6  for  $i = 2$  to  $n$ 
7      for  $j = 2$  to  $\min\{i-1, k\}$ 
8           $A[i][j] = A[i-1][j-1] + A[i-1][j]$ 

```

La complejidad de este método es $\mathcal{O}(nk)$, y $\mathcal{O}(nk) \subseteq \mathcal{O}(n^2)$, ya que $k \leq n$. Además, se puede implementar con una complejidad espacial de $\mathcal{O}(k)$ almacenando solo la fila actual y la anterior de la tabla en el ciclo.

1.3.2. Principio de optimalidad de Bellman

Un problema satisface el *principio de optimalidad de Bellman* cuando para cualquier *sucesión óptima* de decisiones, cada *subsucesión* es a su vez óptima para el subproblema asociado. Esta es una condición necesaria para que aplicar PD sea eficiente.

Ejemplo: Problema de la mochila

Problema:

Dados

- Capacidad $C \in \mathbb{Z}_+$ de la mochila (peso máximo).
- Cantidad $n \in \mathbb{Z}_+$ de objetos.
- Peso $p_i \in \mathbb{Z}_+$ del objeto i .
- Beneficio $b_i \in \mathbb{Z}_+$ del objeto i .

Determinar qué objetos se deben incluir en la mochila para **maximizar** el beneficio total, sin **excederse** del peso máximo C . Formalmente, encontrar:

$$\arg \max \left\{ \sum_{s \in S} b_s \mid S \subseteq \{1, \dots, n\}, \sum_{s \in S} p_s \leq C \right\}$$

Para resolver este problema utilizando PD, se puede definir la función $m(k, D)$ como el valor óptimo para el problema considerando solo los primeros k objetos y una mochila con capacidad D . Los valores de esta función pueden ser guardados en una tabla de $n \times C$ posiciones. Los valores se pueden calcular de manera recursiva:

$$m(k, D) = \begin{cases} 0 & \text{si } k = 0 \vee D \leq 0 \\ \max \{m(k-1, D), b_k + m(k-1, D - p_k)\} & \text{en caso contrario} \end{cases}$$

Esto contempla, para cada k dos posibilidades: o bien el objeto de índice k está en la solución óptima, y entonces $m(k, D) = b_k + m(k-1, D - p_k)$, o bien no, en cuyo caso $m(k, D) = m(k-1, D)$.

Si esta función se implementa directamente en un algoritmo de PD (ya sea top-down o bottom-up) utilizando una matriz como estructura de memoización, tanto la complejidad temporal como la espacial son $\mathcal{O}(nC)$. Esta complejidad es *pseudopolinomial*: está acotada por un polinomio, pero este incluye valores numéricos del input, no solo el tamaño del mismo.

Solución óptima

Calcular $m(k, D)$ nos da el *valor óptimo*, pero no la *solución óptima*. Para obtener el conjunto de objetos que resulta en ese valor se debe reconstruir a partir de la tabla calculada. El esquema general para la reconstrucción se basa en la observación anterior: recorriendo los índices de atrás para adelante, si $m(k, D) = b_k + m(k-1, D - p_k)$, entonces el valor k está en (alguna) solución. Si no, es porque $m(k-1, D) > b_k + m(k-1, D - p_k)$, es decir, ignorar el objeto k resulta en un mayor beneficio total. Este procedimiento permite obtener el conjunto solución en tiempo lineal (una vez que ya se ejecutó el algoritmo anterior).

Ejemplo: Multiplicación de matrices

Problema:

Dadas M_1, M_2, \dots, M_n , calcular:

$$M = M_1 \times M_2 \times \dots \times M_n$$

Realizando la menor cantidad de multiplicaciones entre números de punto flotante.

La dificultad de este problema radica en que la cantidad de operaciones realizadas depende de la forma en la que se asocie el producto. Para resolverlo, se puede observar que alguna de las multiplicaciones tiene que ser la última realizada, es decir, para algún i , se deben multiplicar primero las matrices de 1 a i por un lado y las de $i + 1$ a n por el otro, y finalmente multiplicar estos 2 resultados. Estos dos sub-problemas ($M_1 \times M_2 \times \dots \times M_i$ y $M_{i+1} \times M_{i+2} \times \dots \times M_n$) deben ser resueltos, a su vez, de forma óptima.

Luego, suponiendo que las dimensiones de las matrices están dadas por un vector $d \in \mathbb{N}^{n+1}$ tal que $M_i \in \mathbb{R}^{d[i-1] \times d[i]}$, se puede implementar el siguiente algoritmo bottom-up:

MIN-OPERACIONES(d)

```
1  Inicializar la matriz  $m \in \mathbb{N}^{n \times n}$ 
2  for  $i = 1$  to  $n$ 
3       $m[i][i] = 0$ 
4  for  $i = 1$  to  $n - 1$ 
5       $m[i][i + 1] = d[i - 1]d[i]d[i + 1]$ 
6  for  $s = 2$  to  $n - 2$ 
7      for  $i = 1$  to  $n - s$ 
8           $m[i][i + s] = \min \{m[i][k] + m[k + 1][i + s] + d[i - 1]d[k]d[i + s] \mid i \leq k < i + s\}$ 
```

En este caso, $m[i][j]$ representa la cantidad mínima de operaciones necesarias para calcular $M_i \times M_{i+1} \times \dots \times M_j$, y por ende el valor óptimo es $m[1][n]$. Para obtener la secuencia de multiplicaciones, se puede emplear un procedimiento similar [al del ejemplo anterior](#).

1.4. Algoritmos Golosos

1.4.1. Heurísticas

Una *heurística* para un problema dado es un procedimiento computacional que intenta obtener soluciones de “buena calidad” para el mismo. Por ejemplo, para un problema de optimización, una heurística obtendría una solución con un valor cercano al óptimo.

Un algoritmo A es ϵ -aproximado cuando:

$$\left| \frac{x_A - x^*}{x^*} \right| \leq \epsilon$$

Donde x^* es el valor óptimo, y x_A es el resultado del algoritmo.

Un ejemplo práctico es el algoritmo de Christofides y Serdyukov, un algoritmo $\frac{1}{2}$ -aproximado para instancias del problema del viajante de comercio que forman un espacio métrico (las distancias son simétricas y obedecen la desigualdad triangular). Lo notable de este algoritmo es que tiene complejidad polinómica, siendo el TSP un problema NP-Completo.

1.4.2. Algoritmos golosos

Los *algoritmos golosos* se basan en construir una solución para un problema seleccionando en cada la “mejor” alternativa, sin considerar (o haciéndolo débilmente) las implicancias posteriores de esa selección. Habitualmente, proporcionan heurísticas sencillas para los problemas de optimización, produciendo soluciones razonables (aunque subóptimas) en tiempos eficientes. Sin embargo, existen casos donde la solución que generan es óptima.

Ejemplo: Problema de la mochila

A pesar de haber resuelto el problema [anteriormente](#), un enfoque goloso puede proveer soluciones (subóptimas) con mayor eficiencia temporal. El esquema general es agregar a la mochila el objeto i que...

1. ...tenga el mayor beneficio b_i .
2. ...tenga el menor peso p_i .
3. ...maximice $\frac{b_i}{p_i}$ (la “densidad”).

Se puede demostrar que, si se corre el algoritmo goloso 2 veces, una con el primer criterio y otra con el segundo, alguno de los resultados tiene un valor de al menos la mitad de la solución óptima. Esto hace al procedimiento un algoritmo $\frac{1}{2}$ -aproximado, y se puede implementar en tiempo $\mathcal{O}(n \log n)$ si se ordenan los elementos previamente (es aún más eficiente usar una cola de prioridad implementadas con heap).

Por otro lado, si cambia el problema, permitiendo poner una fracción de cada elemento en la mochila, el algoritmo goloso que utiliza el tercer criterio devuelve soluciones óptimas.

Ejemplo: Problema del cambio

Problema:

Dado un monto m y un conjunto de denominaciones d_1, \dots, d_k , encontrar la mínima cantidad de monedas necesarias para obtener el valor m .

Para encontrar soluciones (no necesariamente óptimas) de este problema, se puede emplear un algoritmo goloso simple: en cada paso, seleccionar la moneda de mayor valor que no exceda el monto restante.

DAR-CAMBIO(D, m)

```
1 suma = 0
2 M = {}
3 while suma < m
4     próxima = máx {d | d ∈ D, d ≤ m}
5     M = M ∪ {próxima}
6     suma = suma + próxima
7 return M
```

Para ciertos conjuntos de denominaciones, como el tradicional $\{1, 5, 10, 25, 50\}$, este algoritmo siempre devuelve soluciones óptimas, mientras que para otros no (en $D = \{1, 5, 10, 12\}$, $m = 21$, el algoritmo devuelve un conjunto de 6 monedas cuando la solución óptima tiene 3).

El algoritmo es goloso porque en cada paso selecciona la moneda de mayor valor posible, sin preocuparse que esto puede llevar a una mala solución, y nunca modifica una decisión tomada.

Ejemplo: Tiempo de espera total en un sistema

Problema:

Un servidor tiene n clientes que puede atender en cualquier orden, y el tiempo necesario para atender al cliente i es $t_i \in \mathbb{R}_+$. Encontrar un orden de atención que minimice el tiempo de espera total de todos los clientes.

Si se denota $I = (i_1, \dots, i_n)$ al orden de atención, el tiempo de espera total T se puede calcular de la siguiente manera:

$$T = t_{i_1} + (t_{i_1} + t_{i_2}) + \dots = \sum_{k=1}^n (n - k + 1)t_{i_k}$$

Se puede plantear el siguiente algoritmo goloso: En cada paso, atender al cliente pendiente que tenga el menor tiempo de atención. La idea detrás de ese criterio es que el tiempo de los clientes que son atendidos primero tendrá que ser esperado por todos los demás, así que lo ideal es que sea el mínimo. Formalmente, la solución $I = (i_1, \dots, i_n)$ es una que cumple $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \dots, n - 1$.

En este caso, la solución que proporciona el algoritmo resulta ser óptima. Por otro lado, la complejidad temporal es $\mathcal{O}(n \log n)$, ya que el procedimiento es equivalente a ordenar a los clientes por tiempo de espera.

1.5. Algoritmos Probabilísticos

Un *algoritmo probabilístico* es uno que emplea un grado de aleatoriedad en su ejecución. Los efectos de esta aleatoriedad pueden variar: en **algunos casos** solo varía el tiempo de ejecución, mientras que en **otros** la salida tiene una probabilidad de ser incorrecta.

1.5.1. Algoritmos numéricos

Un *algoritmo numérico probabilístico* es uno que aproxima la solución a un problema matemático. Estos algoritmos suelen ser adaptaciones aleatorizadas de algoritmos clásicos, como el método de cuadratura bayesiana para la integración numérica, o el de optimización bayesiana para problemas de optimización.

1.5.2. Algoritmos de Monte Carlo

Los *algoritmos de Monte Carlo* son aquellos que proporcionan una respuesta que tiene cierta probabilidad (típicamente baja) de ser incorrecta. En general, si estos algoritmos se corren varias veces, la probabilidad de que la respuesta obtenida sea correcta aumenta (asumiendo independencia entre las distintas ejecuciones). Un ejemplo de estos algoritmos sería el test de primalidad de Solovay-Strassen, que siempre identifica a números primos correctamente, pero tiene una probabilidad menor a $\frac{1}{2}$ de devolver una respuesta falsa para los compuestos.

1.5.3. Algoritmos de Las Vegas

Los *algoritmos de Las Vegas* siempre devuelven una respuesta cuando terminan, pero su tiempo de ejecución es aleatorio (potencialmente infinito). Un ejemplo podría ser un algoritmo para el problema de n damas que chequea configuraciones aleatorias hasta encontrar una que satisface las restricciones.

1.5.4. Algoritmos de Sherwood

Los *algoritmos de Sherwood* son algoritmos que aleatorizan procesos determinísticos, habitualmente aquellos que tienen una gran diferencia entre el peor caso y el promedio. El ejemplo clásico es el algoritmo de quicksort con pivote seleccionado aleatoriamente.

Capítulo 2

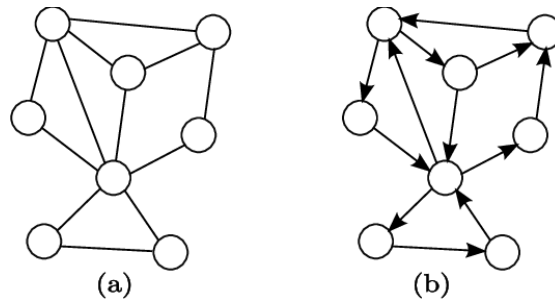
Introducción a Teoría de Grafos

2.1. Grafos

2.1.1. Definición

Un *grafo* es un par ordenado $G = (V, E)$: el conjunto V , de *vértices* o *nodos*, y el de aristas/arcos E , que relacionan a esos nodos.

En el caso de los grafos *no dirigidos* (o simplemente grafos), $E \subseteq \{\{v_1, v_2\} \mid v_1, v_2 \in V, v_1 \neq v_2\}$ es un conjunto de pares no ordenados de los elementos de V , conocidos como *aristas*. Por otro lado, para los *grafos dirigidos* (también llamados *digrafos*), $E \subseteq V \times V$ tiene pares ordenados de nodos, y sus elementos se denominan arcos. Se suele realizar un abuso de notación menor, utilizando (v, w) para referirse tanto a aristas como arcos.



Representaciones gráficas de un grafo (a) y un digrafo (b).
Los círculos son los vértices, y las líneas/flechas son las aristas/arcos.

En general, se denota $n_G = |V|$ y $m_G = |E|$ para referirse a las cantidades de vértices y aristas. Cuando el grafo referido es inambiguo, se omite el subíndice¹.

¹Lo mismo vale para el resto de las definiciones en esta sección.

2.1.2. Vecinos

Dados $v, w \in V$, se denominan *adyacentes* cuando $e = (v, w) \in E$, y que e es *incidente* a v y w . Similarmente, la *vecindad* de v , denotada por $N_G(v)$ es el conjunto de vértices adyacentes a v , es decir:

$$N_G(v) = \{w \in V \mid (v, w) \in E\}$$

Por otro lado, la cantidad de aristas incidentes a un vértice v se llama *grado*, definida como:

$$d_G(v) = |N_G(v)|$$

Teorema. Dado un grafo de $G = (V, E)$, la suma de los grados de sus vértices es el doble de la cantidad de aristas. Es decir,

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

Demostración. Se puede demostrar por inducción en m , la cantidad de aristas.

Caso base: Se puede tomar como caso base $m = 0$. En un grafo sin aristas, todos los vértices tienen grado 0, y por ende:

$$\sum_{v \in V} d(v) = 0 = 2m$$

Paso inductivo: Asumiendo que la propiedad se cumple para $m = k$, tomemos un grafo cualquiera $G = (V, E)$ con $|E| = k + 1$ aristas. Se puede elegir una arista cualquiera $e = (v, w) \in E$, y construir el grafo $G' = (V, E - e)$ que resulta de quitar una de sus aristas. Como $m_{G'} = k$, se cumple la hipótesis inductiva:

$$\sum_{u \in V} d_{G'}(u) = 2m_{G'} = 2k$$

Luego, para el grafo original G , la adición de la arista e solo incrementa el grado de los vértices v y w . Concretamente:

$$d_G(u) = \begin{cases} d_{G'}(u) + 1 & \text{si } u = v \vee u = w \\ d_{G'}(u) & \text{en caso contrario} \end{cases}$$

Por lo tanto, se tiene:

$$\begin{aligned} \sum_{u \in V} d_G(u) &= \sum_{v \in V - \{v, w\}} d_{G'}(u) + (d_{G'}(v) + 1) + (d_{G'}(w) + 1) \\ &= \sum_{u \in V} d_{G'}(u) + 2 \stackrel{HI}{=} 2k + 2 = 2(k + 1) = 2m_G \end{aligned}$$

Lo cual, por inducción, implica que la propiedad vale para todo $m \in \mathbb{N}$.

□

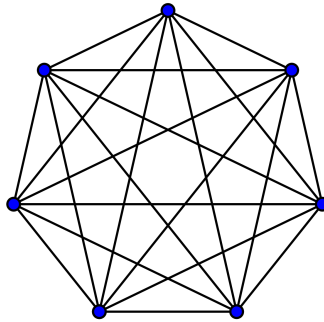
Complemento

Dado un grafo $G = (V, E)$, su *grafo complemento*, denotado como $\bar{G} = (V, \bar{E})$, tiene el mismo conjunto de vértices, pero cada par de vértices es adyacente en \bar{G} si y solo si no lo es en G . Es decir,

$$\bar{E} = (V \times V) - E$$

Grafos Completos

El grafo K_n es el *grafo completo* de n vértices, los cuales son todos adyacentes entre sí. Este grafo tiene $m_{K_n} = \frac{n(n-1)}{2}$.



Representación gráfica del grafo completo K_7 .

2.1.3. Generalizaciones

Algunas generalizaciones de grafos² son:

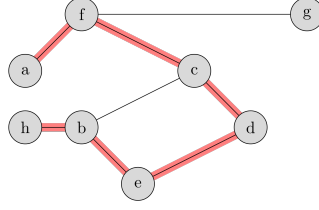
- **Multigrafos:** En un multigrafo, E pasa a ser un multiconjunto, es decir, pueden haber varias aristas entre un mismo par de vértices.
- **Pseudografo:** Los pseudografos pueden tener varias aristas entre un mismo par de vértices, y también puede haber aristas que unan a un mismo par de vértices (llamadas *loops*).

2.1.4. Recorridos

- Un *recorrido* en un grafo es una secuencia de vértices $P = v_0 v_1 \cdots v_k$ tal que todos los pares consecutivos son adyacentes, es decir, $(v_i, v_{i+1}) \in E \forall i = 0, \dots, k-1$. Para multi- y pseudo-grafos, se debe especificar entre qué aristas se pasa.
- Un *camino* es un recorrido que no pasa por el mismo vértice 2 veces.
- Una *sección* de un recorrido P es una subsecuencia $S = v_i v_{i+1} \cdots v_j$ de vértices consecutivos de P , y se denota $P_{v_i v_j}$.
- Un *circuito* es un recorrido que empieza y termina en el mismo vértice.

²No se estudian mucho en la materia.

- Un *ciclo* o *circuito simple* es un circuito de 3 o más vértices que no pasa 2 veces por el mismo vértice (salvo por el principio y fin).



Un ejemplo de un camino entre los vértices a y h .

2.1.5. Distancia

Dado un recorrido P , su *longitud*, $l(P)$, es la cantidad de aristas que tiene. Luego, la *distancia* entre v y w se define como la longitud del camino más corto entre v y w , y se llama $d(v, w)$. Si no hay recorrido entre v y w , se define que $d(v, w) = \infty$, mientras que $d(v, v) = 0$ para cualquier v .

Teorema. Si un recorrido P entre v y w cumple $l(P) = d(v, w)$, entonces es un camino.

Demostración. Se puede demostrar por el absurdo: si P no fuera un camino, tendría algún vértice u por el que se pasa 2 veces: $P = v \cdots u \cdots u \cdots w$. Si se forma un nuevo recorrido $P' = P_{vu} + P_{uw}$ (excluyendo el recorrido de u a sí mismo), este tendría una longitud estrictamente menor que P , y por ende $l(P') < d(v, w)$ (**Absurdo**).

□

Teorema. Para cualquier grafo $G = (V, E)$, la función de distancia $d : V \times V \rightarrow \mathbb{N}$ es una métrica, es decir, cumple las siguientes propiedades para todo $u, v, w \in V$:

- $d(u, v) = 0 \iff u = v$
- $d(u, v) = d(v, u)$
- $d(u, w) \leq d(u, v) + d(v, w)$ (desigualdad triangular)

Demostración. Se demuestra por separado:

- La ida vale por definición, y la vuelta vale porque cualquier camino entre un par de vértices tiene al menos 1 arista (y por ende $d(u, v) \geq 1$).
- En un grafo las aristas no tienen sentido, así que cualquier camino puede ser invertido para formar un camino válido. Por ende, la longitud del camino más corto entre u y v debe ser la misma que entre v y u .
- Si P_{uv} y P_{vw} son caminos tales que $l(P_{uv}) = d(u, v)$ y $l(P_{vw}) = d(v, w)$, se pueden concatenar para formar un recorrido $P_{uv} + P_{vw}$ entre u y w . Como la distancia es la longitud

mínima entre todos los recorridos, se tiene $d(u, w) \leq l(P_{uv} + P_{vw}) = d(u, v) + d(v, w)$.

□

2.1.6. Subgrafos

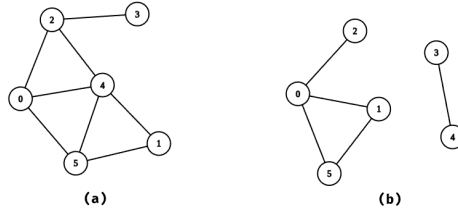
Dado un grafo $G = (V_G, E_G)$,

- Un *subgrafo* de G es un grafo $H = (V_H, E_H)$ tal que $V_H \subseteq V_G$ y $E_H \subseteq E_G \cap (V_H \times V_H)$. Los notamos como $H \subseteq G$.
- H es un *subgrafo propio* cuando $H \subseteq G$ y $H \neq G$.
- H es un *subgrafo generador* cuando $H \subseteq G$ y $V_H = V_G$.
- H es un *subgrafo inducido* cuando $(v, w) \in E_H \iff v, w \in V_H \wedge (v, w) \in E_G$. Estos subgrafos pueden definirse únicamente por su conjunto de vértices, y se denota como $G_{[V_H]}$.

2.1.7. Conectividad

Un grafo se denomina *conexo* cuando existe un camino entre todo par de vértices. Una *componente conexa* de un grafo es un subgrafo inducido conexo maximal (no se pueden agregar más vértices y mantenerlo conexo) de G .

Por otro lado, una arista de G es *punte* si $G - e$ tiene más componentes conexas que G .



Un grafo conexo (a) y uno desconexo (b).

2.1.8. Representación de Grafos

Existen distintas alternativas para representar grafos en un algoritmo, que proveen ventajas y desventajas a la hora de realizar diversas operaciones.

Lista de aristas

El grafo se almacena como una lista de pares de vértices, que representan sus aristas. Esta es la forma más simple de representarlo, y es el formato que se asume que tiene la entrada de cualquier algoritmo de grafos. Debido a su falta de estructura, realizar la mayoría de las operaciones resulta costoso, con la excepción de agregar nodos o aristas.

Esta estructura tiene ciertas variaciones. Por ejemplo, se pueden ordenar los vértices dentro de cada lista, lo cual permite usar búsqueda binaria para comprobar la pertenencia de un vértice a ellas, pero aumenta la complejidad de construir la estructura y la de agregar vértices (porque hay que mantener el orden).

Listas de adyacencia

Se mantienen n listas, donde cada lista L_i contiene todos los vértices de $N(v_i)$. Esto permite realizar algunas operaciones más rápidamente, y la estructura se puede construir a partir de la lista de aristas en tiempo lineal.

Matriz de adyacencia

En este caso, se tiene una matriz $M \in \{0, 1\}^{n \times n}$, donde cada posición está determinada por:

$$M_{ij} = \begin{cases} 1 & \text{si } (i, j) \in E \\ 0 & \text{en caso contrario} \end{cases}$$

La matriz es simétrica para grafos, pero no necesariamente para digrafos.

La estructura permite comprobar si dos vértices son adyacentes en tiempo constante. Sin embargo, construirla a partir de una lista de adyacencia es una operación de complejidad cuadrática, y la estructura es muy rígida (para agregar un vértice se debe armar una nueva matriz). Además, la complejidad espacial es también $\mathcal{O}(|V|^2)$, lo cual es problemático para guardar grafos malos³.

Matriz de incidencia

Esta estructura es una matriz $I \in \{0, 1\}^{m \times n}$ donde las filas representan los vértices y las columnas las aristas. Una posición i, j tiene uno cuando la arista de la columna j es incidente al vértice de la fila i .

Complejidades

2.1.9. Isomorfismo

Dos grafos $G = (V, E)$ y $G' = (V', E')$ son *isomorfos* cuando existe una función biyectiva $f : V \rightarrow V'$ tal que:

$$\forall v, w \in V, (v, w) \in E \iff (f(v), f(w)) \in E'$$

A la función f se la llama isomorfismo, y se denota $G \cong G'$ o (por abuso de notación) $G = G'$.

Teorema. Si dos grafos $G \cong G'$ son isomorfos.

- Tienen el mismo número de vértices.
- Tiene el mismo número de aristas.

³Un grafo *ralo* es uno con “pocas” aristas.

- $\forall 0 \leq k \leq n - 1$, tienen el mismo número de vértices de grado k .
- Tienen el mismo número de componentes conexas.
- $\forall 0 \leq k \leq n - 1$, tienen el mismo número de caminos simples de longitud k .

Demostración.

□

2.1.10. Definiciones en digrafos

Vecinos

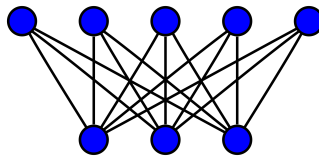
- Para un arco $e = (v, w) = v \rightarrow w$, se llama *cola* de e a v y *cabeza* de e a w .
- El *grado de entrada* $d_-(v)$ es la cantidad de arcos que tienen a v como cabeza.
- El *grado de salida* $d_+(v)$ es la cantidad de arcos que tienen a v como cola.
- El *grafo subyacente* de G es el grafo que resulta de ignorar las direcciones de sus arcos.

Recorridos

- Un *recorrido/camino orientado* en un digrafo es una sucesión de vértices que están conectados apropiadamente por arcos (sin repetidos en el caso del camino).
- Un *circuito/ciclo orientado* es un recorrido/camino orientado que empieza y termina en el mismo vértice.
- Un digrafo es *fuertemente conexo* si para todo par de vértices v, u existen caminos orientados de u a v y de v a u .

2.2. Grafos Bipartitos

Un grafo $G = (V, E)$ es *bipartito* cuando existe una *bipartición* de sus vértices (V_1, V_2) tal que todas las aristas de G tienen un extremo en V_1 y el otro en V_2 . Por otro lado, G es *bipartito completo* cuando todo vértice de V_1 es adyacente a todo vértice de V_2 , y se denota $G = K_{|V_1|, |V_2|}$.



El grafo bipartito completo $K_{3,5}$.

Teorema. Un grafo G es bipartito \iff no tiene ciclos de longitud impar.

Demostración. Como un grafo es bipartito si y solo si cada una de sus componentes conexas es

bipartita, y un grafo no tiene ciclos impares si y solo si ninguna de sus componentes conexas tiene ciclos impares, alcanza con demostrar el teorema para grafos conexos.

\Rightarrow) Sea (V_1, V_2) la bipartición de G .

Si G tiene algún ciclo $C = v_1v_2 \cdots v_kv_1$, se puede asumir sin pérdida de generalidad que $v_1 \in V_1$. Luego, como $v_1v_2 \in E$ (y G es bipartito), $v_2 \in V_2$. En general, $v_{2i+1} \in V_1$ y $v_{2i} \in V_2$. Como $v_1 \in V_1$ y $v_kv_1 \in E$, se debe cumplir $v_k \in V_2$. Por ende $k = 2i$ así que $l(C)$ es par.

\Leftarrow) Sea u cualquier vértice de V . Se definen los siguientes conjuntos:

$$V_1 = \{v \in V \mid 2 \mid d(u, v)\} \cup \{u\}$$

$$V_2 = \{v \in V \mid 2 \nmid d(u, v)\}$$

(V_1, V_2) definen una partición de V . Se puede demostrar que es una bipartición de G por el absurdo.

Supongamos que no es una bipartición, entonces existen $v, w \in V_1$ (s.p.g.) tales que $vw \in E$. Si $v = u$, entonces $d(v, u) = 1$, que es absurdo porque $d(v, u)$ es par. Lo mismo vale para w , así que $v \neq u$ y $v \neq w$.

Sea P un camino mínimo entre v y u y Q uno entre v y w . Como $u, w \in V_1$ P y Q tienen longitud par. Luego, sea z el vértice común a P y Q tal que P_{zv} y Q_{zw} son disjuntos (ignorando z).

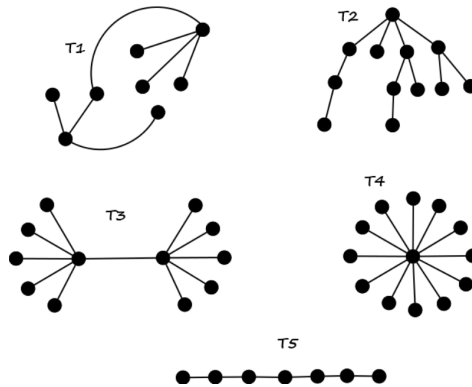
Se debe cumplir $d(u, z) = l(P_{uz}) = l(Q_{uz})$, porque del contrario P y Q no serían caminos mínimos. Esto implica que $l(P_{zv})$ y $l(Q_{zw})$ tienen la misma paridad, porque $l(P)$ y $l(Q)$ son ambos pares y la diferencia entre las longitudes totales y las de los subcaminos es la misma. Por ende, el ciclo $P_{zv}(v, w)Q_{wz}$ tiene longitud impar (**Absurdo**).

□

2.3. Árboles

2.3.1. Definición

Un *árbol* es un grafo conexo acíclico.



Ejemplos de grafos que son árboles.

Existen caracterizaciones alternativas:

Teorema. Dado un grafo $G = (V, E)$, son equivalentes:

1. G es un árbol (un grafo conexo acíclico).
2. G es un grafo acíclico y $\forall e \notin E, G + e = (V, E \cup \{e\})$ tiene exactamente un ciclo, y ese ciclo pasa por e .
3. Existe exactamente un camino simple entre todo par de vértices de G .
4. G es conexo, pero si se quita cualquier arista de G , queda un grafo desconexo (toda arista es puente).
5. G es un grafo conexo con $|E| = |V| - 1$
6. G es un grafo acíclico con $|E| = |V| - 1$.

Hojas

Una *hoja* en un árbol es un vértice de grado 1. Todo árbol *no trivial* (con al menos 2 vértices) tiene al menos 2 hojas.

Bosques

Un *bosque* es un grafo sin ciclos. Sus componentes conexas forman árboles, y se cumple $m = n - c$, donde c es la cantidad de componentes conexas del bosque.

2.3.2. Árboles enraizados

Un *árbol enraizado* es un árbol con un vértice especial r designado *raíz*. Luego, queda definido un árbol dirigido, donde los arcos van desde vértices más cercanos a la raíz hacia los más lejanos. En tal caso, está la siguiente terminología:

- Los vértices *internos* son aquellos que no son ni hojas ni raíz.

- El *nivel* de un vértice v es la distancia de la raíz a ese vértice ($d(r, v)$).
- Para cada arco $v \rightarrow w$, v es el *padre* de w , y w es el *hijo* de v .
- La *altura* h de un árbol enraizado es la distancia desde la raíz al vértice más lejano ($\max \{d(r, v) \mid v \in V\}$).
- Un árbol se dice *m-ario* si todos sus nodos internos tiene grado a lo sumo $m + 1$ y la raíz tiene grado a lo sumo m .
- Un árbol es *balanceado* cuando la diferencia entre el nivel de cada par de hojas es a lo sumo 1.

Teorema. Dado un árbol enraizado m -ario con altura h y l hojas, $l \leq m^h$ ($\iff h \geq \lceil \log_m l \rceil$).

2.3.3. Representación de árboles

Además de las representaciones de grafos [anteriormente mencionadas](#), los árboles enraizados tienen una alternativa particular: debido a que todos los nodos tienen un único padre, un árbol puede ser definido por la correspondencia entre cada nodo y su antecesor. Esto se puede lograr usando solamente un arreglo “**prev**”, en el que cada posición i contiene al padre del nodo v_i . La única excepción es la raíz r que, al no tener antecesor, se puede marcar utilizando un valor especial \perp , o como su propio padre ($d[r] = r$).

2.3.4. Árbol generador

Un *árbol generador* (AG) de un grafo G es un [subgrafo generador](#) que además es un árbol. En la práctica, los árboles generadores son utilizados cuando se busca conectar (con la cantidad mínima posible de conexiones) a n puntos (ciudades, centrales eléctricas, servidores).

Teorema. Dado un grafo conexo $G = (V, E)$.

- G tiene (al menos) un árbol generador
- G tiene un único árbol generador $\iff G$ es un árbol.
- Sea $T = (V, E_T)$ un AG de G y $e \in E - E_T$. Luego, para toda arista $f \neq e$ contenida en el único ciclo de $T + e$, $T + e - f = (V, E_T \cup \{e\} - \{f\})$ es un AG de G .

2.4. Recorridos

Es común querer pasar por todos los vértices de un grafo una única vez. Existen distintos métodos de hacerlo de forma sistemática y ordenada, y en este caso nos vamos a enfocar en los 2 más comunes: *BFS* y *DFS*. En ambos casos, se mantiene una *frontera* con los vértices que se están por explorar, y cada vez que se pasa por uno de ellos sus vecinos son agregados a la misma. Además, se mantiene un conjunto de los vértices explorados (generalmente implementado con un *bitset*) para evitar su repetición. El esquema general es el siguiente:

```

RECORRER( $G, s$ )
1  Inicializar la frontera  $F = \{s\}$ .
2  while la frontera no esté vacía
3      Extraer un  $v$  de la frontera.
4      PROCESAR( $v$ )
5      for each  $u \in N(v)$ 
6          if  $u$  no fue visitado
7              Marcar a  $u$  como visitado.
8              Agregar  $u$  a la frontera.

```

2.4.1. BFS

El *Breadth-First Search* (BFS) es un algoritmo que, dado un grafo G y un vértice inicial s , recorre todos los vértices nivel por nivel, es decir, los vértices más cercanos al inicial son visitados primero. Formalmente, si $\langle v_1, \dots, v_n \rangle$ es la secuencia de vértices en el orden en que son recorridos, entonces se cumple:

$$d(s, v_i) \leq d(s, v_j) \quad \forall 1 \leq i \leq j \leq n$$

Para lograr esto, BFS utiliza como frontera una cola, donde los elementos son procesados en orden de llegada (FIFO). El algoritmo se puede implementar iterativamente de la siguiente manera:

```

BFS( $G = (V, E), s$ )
1   $visitados = \emptyset$ 
2  Inicializar árbol  $T$  con raíz en  $s$ .
3  Inicializar arreglo de distancias  $d$ .
4  Inicializar cola  $Q$ .
5   $d[s] = 0$ 
6  ENCOLAR( $Q, s$ )
7  while  $\neg \text{VACÍO?}(Q)$ 
8       $v = \text{DESENCOLAR}(Q)$ 
9      PROCESAR( $v$ )
10     for each  $u \in N(v)$ 
11         if  $u \notin visitados$ 
12              $visitados = visitados \cup \{u\}$ 
13              $d[u] = d[v] + 1$ 
14             Agregar  $u$  a  $T$  como hijo de  $v$ .
15             ENCOLAR( $Q, v$ )
16 return ( $T, d$ )

```

El algoritmo devuelve 2 valores: un árbol generador T , que se denomina *árbol BFS* y contiene las aristas transitadas por el recorrido, junto con la función $d : V \rightarrow \mathbb{N}_0$, que indica las distancias de s a cada vértice del grafo.

Si el grafo se representa utilizando listas de adyacencia, el vecindario $N(v)$ se puede recorrer fácilmente. El algoritmo pasa una sola vez por cada vértice y, asumiendo que tanto $visitados$, T y d se representan a través arreglos, realiza una operación de tiempo constante en cada uno de

sus vecinos. Por ende, la complejidad de este algoritmo es:

$$\mathcal{O}(|V| + \sum_{v \in V} d(v)) = \mathcal{O}(|V| + 2|E|) = \mathcal{O}(|V| + |E|)$$

Árboles geodésicos

Un árbol generador T de un grafo G se llama v -geodésico cuando $d_G(v, w) = d_T(v, w) \forall w \in V$.

Teorema. Si se corre el algoritmo *BFS* en un grafo G empezando en un vértice s , el árbol generador resultante T es s -geodésico.

Demostración.

□

2.4.2. DFS

La estrategia que sigue el *Depth-First Search* (DFS) es buscar “en profundidad” siempre que sea posible. Esto significa que al llegar a v , se recorren todos los vértices no visitados alcanzables desde este. Este procedimiento se realiza hasta que todos los nodos hayan sido explorados.

El algoritmo de DFS se puede implementar recursivamente de la siguiente manera (*visitados*, T , *principio*, *fin* y *contador* son variables globales):

DFS(G, s)

- 1 $visitados = \emptyset$
- 2 $contador = 0$
- 3 Inicializar arreglos *principio* y *fin*.
- 4 Inicializar T como árbol vacío.
- 5 VISITAR-DFS(G, s)

VISITAR-DFS(G, v)

- 1 $contador = contador + 1$
- 2 $principio[v] = contador$
- 3 $visitados = visitados \cup v$
- 4 **for each** $u \in N(v)$
- 5 **if** $u \notin visitados$
- 6 Agregar u como hijos de v en el árbol T .
- 7 VISITAR-DFS(G, u)
- 8 $contador = contador + 1$
- 9 $fin[v] = contador$

El tiempo de ejecución del algoritmo, al igual que BFS, es lineal⁴: hay una llamada por cada nodo, y cada llamada tiene un tiempo de ejecución proporcional al grado del nodo, así que la complejidad es $\mathcal{O}(|V| + |E|)$.

⁴La linealidad de estas complejidades se refiere a que, como los grafos se pasan como listas de adyacencia, el tamaño de la entrada es $|E|$. Si se considerara la cantidad de vértices, una complejidad de $\mathcal{O}(|E|)$ sería cuadrática, ya que $|E| \in \mathcal{O}(|V|^2)$.

Al terminar, DFS no solo devuelve el árbol generado T , sino que también un par de arreglos *principio* y *fin*. El primero guarda el orden en el que se empieza a explorar el subárbol de cada nodo (también llamado *pre-order*), mientras que el segundo guarda el orden en el que se termina dicha exploración (también llamado *post-order*). Estos valores son muy útiles para analizar la estructura del árbol.

Teorema. Dado grafo G y un árbol DFS T_G y un par de vértices v, u , se cumple alguna de las siguientes:

1. $[principio[v], fin[v]] \cap [principio[u], fin[u]] = \emptyset$, y en tal caso v y u están en ramas distintas (ninguno es descendiente del otro).
2. $[principio[v], fin[v]] \subseteq [principio[u], fin[u]]$, y entonces el vértice v es descendiente de u .
3. $[principio[v], fin[v]] \supseteq [principio[u], fin[u]]$, y entonces el vértice u es descendiente de v .

Tipos de aristas

Dado un grafo G y un árbol DFS T , las aristas de G se pueden dividir en las siguientes categorías:

- *Tree edges*: son aquellas que están en E_T .
- *Back edges*: son aquellas que no están en E_T , y que conectan a un nodo con un antecesor en T .
- *Forward edges*: son aquellas que no están en E_T , y que conectan a un nodo con un descendiente en T .
- *Cross edges*: son aquellas que no están en E_T , y que conectan a nodos de distintas ramas del árbol.

La categoría de cualquier arista puede ser identificada en tiempo constante utilizando los resultados del DFS: la pertenencia a E_T se puede chequear revisando los padres de los vértices incidentes a la arista en el árbol, mientras que las otras condiciones se pueden verificar a través de los arreglos *principio* y *fin*.

Teorema. Dado un grafo no dirigido G , todas las aristas de cualquier árbol DFS son *Tree Edges* o *back edges*.

Detección de ciclos

Teorema. Un grafo G es acíclico \iff ningún árbol DFS de G tiene *back edges*.

Demostración.

\implies) Se demuestra por el contrarrecíproco: si T es un árbol DFS con una backedge $e = u \rightarrow v$, se puede tomar el ciclo $C = P + e$, donde P es el camino que une a v y u en T (debe existir, ya que v es antecesor de u por ser e back edge).

\impliedby) También se demuestra el contrarrecíproco: supongamos que C es un ciclo de G .

Dado un recorrido DFS cualquiera, sea v el primer vértice de C que se encuentra y u el vértice “anterior”⁵ en el ciclo. Luego, como u es alcanzable desde v , será uno de sus descendientes, así que la arista $u \rightarrow v$ es una back edge.

□

El algoritmo DFS puede ser utilizado para encontrar ciclos de un grafo, ya que todas las back edges forman parte de al menos un ciclo. Existen varias opciones:

- En el caso de los grafos no dirigidos, basta con adaptar DFS para devolver el ciclo cuando un vértice u adyacente al actual v ya fue visitado. En ese caso, el ciclo es $C = uT_{uv}vu$, donde T_{uv} es el camino entre u y v en el árbol. Esto funciona porque cuando se visita un vértice por segunda en un grafo no dirigido vez siempre se hace a través de una Back Edge.
- Otra opción que también sirve para grafos dirigidos es pasar por todas las aristas que no están en el árbol hasta identificar una Back Edge (usando las condiciones [establecidas anteriormente](#))

Versión iterativa

La versión iterativa de DFS sigue el esquema general [establecido previamente](#). En este caso, la frontera se implementa utilizando un stack (FILO), lo cual garantiza que un vértice se deja de explorar solo cuando todos los vértices alcanzables desde ese fueron visitados.

Bosques

Si el grafo recorrido G no es conexo, se puede formar un bosque donde cada componente conexa es un árbol DFS. Esto se logra corriendo DFS iterativamente, cada vez empezando en uno de los vértices que aún no fue recorrido por las iteraciones anteriores. El algoritmo es el siguiente, donde VISITAR-DFS es el procedimiento definido anteriormente:

DFS(G)

```

1  visitados =  $\emptyset$ 
2  contador = 0
3  Inicializar arreglos principio y fin.
4  Inicializar  $T$  como árbol vacío.
5  for each  $v \in V$ 
6      if  $v \notin \textit{visitados}$ 
7          VISITAR-DFS( $G, v$ )
```

⁵En grafos no dirigidos hay dos: se puede tomar sin pérdida de generalidad.

2.5. Orden Topológico

2.5.1. Definición

Problema:

Dado un digrafo acíclico (un “DAG”) $D = (V, E)$, encontrar un ordenamiento $\langle v_1, v_2, \dots, v_n \rangle$ de sus vértices de forma tal que, para todo $v \in V$ los vértices alcanzables desde v aparezcan después en el orden. Formalmente,

$$d(v_i, v_j) < \infty \implies \forall 1 \leq i \leq j \leq n$$

Esto se denomina un *orden/ordenamiento topológico*, y tiene múltiples aplicaciones prácticas, que surgen cada vez que se debe ordenar un conjunto de cosas con precedencias entre sí. Notar que solo es posible para digrafos acíclicos: en un ciclo, todos los vértices son alcanzables desde los demás, así que ninguno podría ir antes de los demás.

2.5.2. Algoritmo

Habiendo implementado y analizado DFS, el algoritmo para encontrar un ordenamiento topológico es simple, gracias al siguiente teorema:

Teorema. *Para un DAG D , el ordenamiento inverso al post-order de cualquier DFS es un orden topológico.*

Demostración. Como el valor de $\text{fin}[v]$ se asigna después de haber explorado por todos los vértices u alcanzables desde v , se cumple que $\text{fin}[u] < \text{fin}[v]$. Por ende, si se ordenan los vértices por valor fin decreciente, los vértices que se pueden alcanzar desde otros aparecerán después. \square

Para implementar este algoritmo, no es necesario ordenar directamente los vértices a través de fin (lo tendría complejidad $\mathcal{O}(|V| \log |V|)$), sino que basta con modificar DFS: después de terminar de explorar el subárbol de cada vértice, se lo agrega al principio de una secuencia. Esto produce un ordenamiento topológico de D .

2.6. Algoritmo de Korasaju

El algoritmo de Korasaku es un algoritmo lineal que encuentra las **componentes fuertemente conexas** de un digrafo G . Se basa en hacer 2 recorridos DFS: el primero se utiliza para obtener un post-order de los vértices del grafo, mientras que el segundo lo recorre de manera inversa para armar los componentes. Este segundo DFS se hace sobre G^T , el digrafo cuyas aristas tienen el sentido opuesto al de las de G . El algoritmo es el siguiente:

KOSARAJU(G)

- 1 Llamar a DFS(G) para obtener un post-order inverso.
- 2 Computar G^T
- 3 Llamar a DFS(G^T), pero en el loop principal explorar los vértices en el post-order inverso.
Cada vez que se visita un vértice nuevo, agregarlo a la CFC de la raíz actual.

El loop principal de DFS se refiere al de la versión que arma bosques iterando por cada vértice sin explorar.

Para ver por qué este algoritmo funciona, primero se debe observar el siguiente lema:

Lema. Dadas CFCs⁶ C y C' distintas en $G = (V, E)$, ningún vértice de C es alcanzable desde C' , o viceversa.

Demostración. Esto se puede demostrar por el absurdo: si existen caminos $u \rightsquigarrow v$ y $u' \rightsquigarrow v'$ tales que $u, v \in C$ y $u', v' \in C'$, entonces cualquier vértice de C puede llegar a cualquiera de C' a pasando por $u \rightsquigarrow v$, y cualquiera de C' puede a su vez llegar a uno de C a través de $u' \rightsquigarrow v'$. Esto implica que todos están en la misma CFC (**Absurdo**).

□

Esto permite visualizar las componentes conexas de un digrafo como un DAG: G^{CFC} se puede definir como el grafo donde cada vértice es una versión compactada de un CFC de G . No puede haber ciclos porque violarían el lema anterior.

Luego, el post-order se puede emplear gracias a que:

Lema. Si C y C' son CFCs distintas de G y existe un arco $u \rightarrow v$ tal que $u \in C$ y $v \in C'$, entonces $\text{fin}[C] > \text{fin}[C']$ ⁷.

Demostración. Se puede separar en dos casos dependiendo de qué CFC se visita antes:

- Si algún vértice w de C se explora primero, entonces esta exploración va a alcanzar el vértice u (porque están en la misma CFC), y por ende visitar todos los vértices de C' antes de $\text{fin}[w]$, y por ende $\text{fin}[C] > \text{fin}[C']$.
- Si C' se visita primero, la exploración va a concluir sin pasar por ningún vértice de C ya que, gracias al lema anterior, no existen caminos entre vértices de C' y los de C . Esto implica que $\text{fin}[C] > \text{fin}[C']$.

□

Esto tiene como corolario que, en el grafo G^T , un par de CFCs C y C' con un arco $u \rightarrow v$ tal que $u \in C$ y $v \in C'$ cumplen $\text{fin}[C] < \text{fin}[C']$. Eso se debe a que G^T tiene los mismos CFCs que G , y $u \rightarrow v \in E^T \implies v \rightarrow u \in E$.

Finalmente, se puede demostrar la correctitud del algoritmo:

Teorema. El algoritmo de Kosaraju devuelve las componentes conexas de un digrafo G .

Demostración. Como el segundo DFS empieza por un vértice v de la CFC con $\text{fin}[C]$ máximo, ninguno de sus vértices contiene una arista hacia otra CFC C' (porque en ese caso $\text{fin}[C] < \text{fin}[C']$). Además, como todos los vértices de C son alcanzables desde v , la CFC se construye

⁶Componentes Fuertemente Conexas

⁷El fin de una CFC se define como $\max \{\text{fin}[v] \mid v \in C\}$.

correctamente (se visitan todos sus vértices, y ningún otro). Luego, para las demás componentes, sucederá algo similar: las únicas aristas que las conectan con otras CFCs serán a aquellas con un mayor valor de fin , y por ende ya habrán sido exploradas. Esto implica que el algoritmo arma cada componente de forma correcta.

□