Resumen para Algoritmos y Estructuras de Datos III

Basado en esta guía de estudio.

Tomás Spognardi

6 de agosto de 2022

Índice general

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, indendientemente de la máquina sobre la cuál se implementa y el lenguaje utilizado.

Definición informal

La complejidad de un algoritmo es una función $T_A : \mathbb{N} \longrightarrow \mathbb{R}_{>0}$ 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]

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

- **Complejidad de mejor caso**:

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

- **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] $t_A : I_A \longrightarrow \mathbb{R}_{>0}$ 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 palabra[3]. 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 especiel 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 siguiente[4]:

- 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

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

[4] Este 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 instacia.

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 ⌈√n⌉
2       if n ≡ 0  mód 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 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:

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

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.

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$O(n)$	0.01 ms	0.02 ms	0.03 ms	0.04 ms	0.05 ms
$O(n^2)$	0.10 ms	0.40 ms	0.90 ms	0.16 ms	0.25 ms
$O(n^3)$	1.00 ms	8.00 ms	2.70 ms	6.40 ms	0.12 sg
$O(n^5)$	0.10 sg	3.20 sg	24.30 sg	1.70 min	5.20 min
$O(2^n)$	1.00 ms	1.00 sg	17.90 min	12 días	35 años
$O(3^n)$	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*).

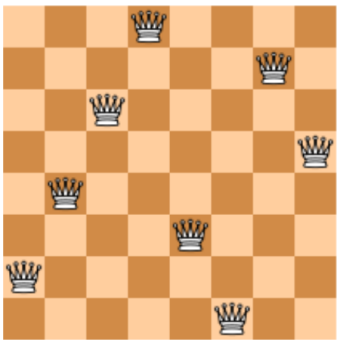
1.2. Backtracking

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 $(a_1, ..., a_n)$, con $a_i \in \{1, ..., n\}$ 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 $a = (a_1, ..., a_k), k < n$, 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 ¬Es-Válida(a)
2       return
3  if Es-Solución(a)
4       Procesar(a)
5       return
6  for a' ∈ 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 ¬Es-Válida(a)
 2       return
 3  if Es-Solución(a)
 4       sol = a
 5       encontró = true
 6       return
 7  for a' ∈ 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 este[5], guardando los resultados de cada llamada en una tabla.

[5]Esto 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:

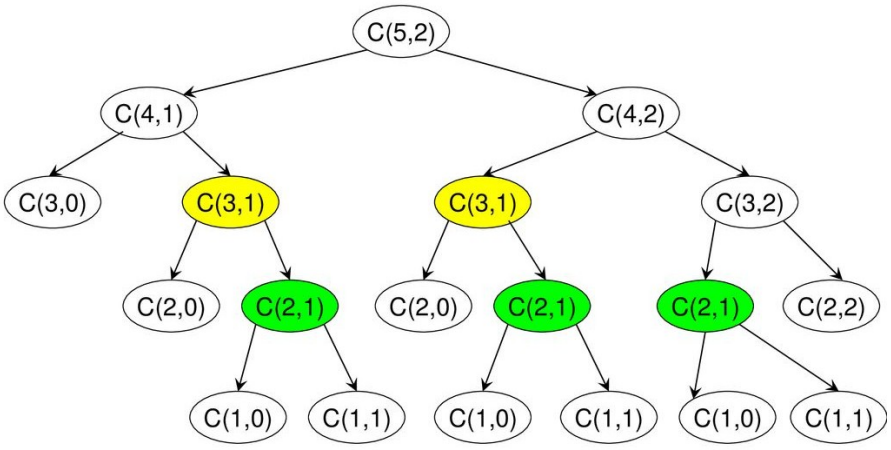
Teorema. *Si* $n \geq 0$ *y* $0 \leq k \leq n$*, entonces:*

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

Esta fórmula recursiva se puede implementar directamente:

```
COMBINATORIO(n, k)
1  if k == 0 ∨ 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 $\quad A[i][0] = 1$
4 **for** $j = 0$ **to** k
5 $\quad A[j][j] = 1$
6 **for** $i = 2$ **to** n
7 $\quad$ **for** $j = 2$ **to** mín $\{i - 1, k\}$
8 $\quad\quad A[i][j] = A[i - 1][j - 1] + A[i - 1][j]$

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:

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

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 númericos 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 $M_1, M_2, ..., M_n$, calcular:
>
> $$M = M_1 \times M_2 \times ... \times M_n$$
>
> 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 ($M_1 \times M_2 \times ... \times M_i$ y $M_{i+1} \times M_{i+2} \times ... \times M_n$) 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 ∈ ℕ^{n×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] = mín {m[i][k] + m[k + 1][i + s] + d[i − 1]d[k]d[i + s] | i ≤ k < i + s}
```

En este caso, $m[i][j]$ representa la cantidad mínima de operaciones necesarias para calcular $M_i \times M_{i+1} \times ... \times M_j$, 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 Serdykuov, 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 heurísitcas 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 colas 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 $d_1, ..., d_k$, 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 $I = (i_1, ..., i_n)$ al orden de atención, el tiempo de espera total T se puede calcular de la siguiente manera:

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

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 $I = (i_1, ..., i_n)$ es una que cumple $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, ..., n - 1$.

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

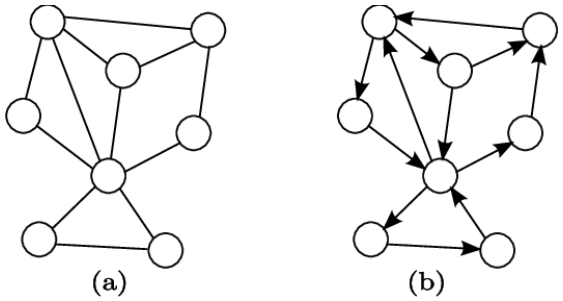
Introducción a Teoría de Grafos

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 inambigüo, se omite el subíndice[1].

[1] Lo 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 incrementea 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:

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

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

□

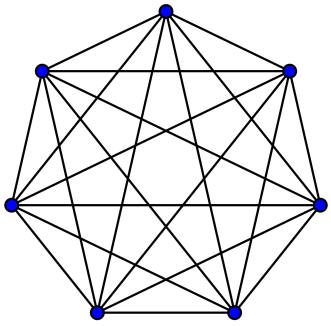
Complemento

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 grafos[2] son:

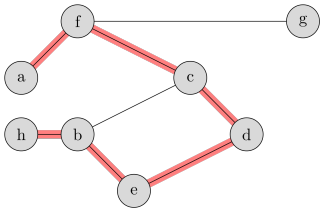
- **Multigrafos**: En un multigrafo, E pasa a ser un multiconjunto, es decir, pueden haber varias aristas entre un mismo par de vértices.
- **Pseudografo**: Los pseudografos pueden tiene varias aristas entre un mismo par de vértices, y también puede haber aristas que unan a un mismo par de vértices (llamadas *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, $(v_i, v_{i+1}) \in E \ \forall i = 0, ..., k-1$. 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.

[2] No 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 $d : V \times V \longrightarrow \mathbb{N}$ 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

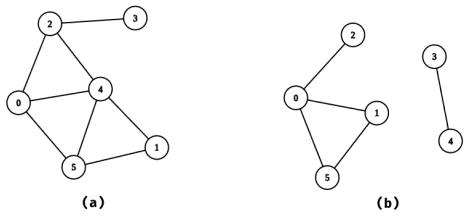
Dado un grafo $G = (V_G, E_G)$,

- 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 $(v,w) \in E_H \iff v,w \in V_H \wedge (v,w) \in E_H$. 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 *puente* si $G - e$ tiene más componentes conexas que G.



Un grafo conexo (a) y uno disconexo (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:

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

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 ralos[3].

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 $f : V \longrightarrow V'$ 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.*

[3] Un 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

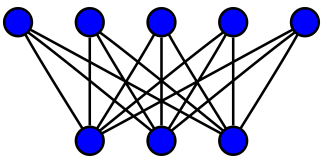
- Para un arco $e = (v, w) = v \to w$, se llama *cola* de e a v y *cabeza* de e a w.
- El *grado de entrada* $d_-(v)$ es la cantidad de arcos que tienen a v como cabeza.
- El *grado de salida* $d_+(v)$ es la cantidad de arcos que tienen a v como cola.
- El *grafo subyacente* de G es el grafo que resulta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

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

Si G tiene algún ciclo $C = v_1 v_2 \cdots v_k v_1$, se puede asumir sin pérdida de generalidad que $v_1 \in V_1$. Luego, como $v_1 v_2 \in E$ (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 $v_k v_1 \in E$, se debe cumplir $v_k \in V_2$. Por ende $k = 2i$ así que $l(C)$ es par.

$\Longleftarrow$) 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 $d(v, w)$ 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).

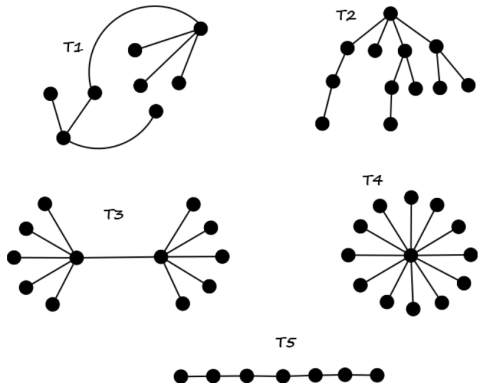
Se debe cumplir $d(u, z) = l(P_{uz}) = l(Q_{uz})$, porque del contrario P y Q no serían caminos mínimos. Esto implica que $l(P_{zv})$ y $l(Q_{zw})$ tienen la misma paridad, porque $l(P)$ y $l(Q)$ son ambos pares y la diferencia entre las longitudes totales y las de los subcaminos es la misma. Por ende, 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 $\forall e \notin X$, $G + e = (V, E \cup \{e\})$ tiene exáctamente 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 disconexo (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 $v \to w$, 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 ($d[r] = r$)[4].

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 vérices 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:

[4] Esto se puede extender para guardar bosques: basta con tener varias raíces.

```
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 ∈ 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 $\langle v_1, ..., v_n \rangle$ es la secuencia de vértices en el orden en que son recorridos, entonces se cumple:

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

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 = ∅
 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 ¬VACÍO?(Q)
 8      v = DESENCOLAR(Q)
 9      PROCESAR(v)
10      for each u ∈ N(v)
11          if u ∉ visitados
12              visitados = visitados ∪ {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 $d : V \longrightarrow \mathbb{N}_0$, 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, y las distancias que devuelve son las mínimas entre s y cada vértice del árbol.*

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 lineal[5]: hay una llamada por cada nodo, y cada llamada tiene un tiempo de ejecución proporcional al grado del nodo, así que la complejidad es $\mathcal{O}(|V| + |E|)$.

[5]La 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 $e = u \to v$, 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"[6] en el ciclo. Luego, como u es alcanzable desde v, será uno de sus descendientes, así que la arista $u \to v$ 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 = ∅
2  contador = 0
3  Inicializar arreglos principio y fin.
4  Inicializar T como árbol vacío.
5  for each v ∈ V
6      if v ∉ visitados
7          VISITAR-DFS(G, s)
```

[6] En 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 $\langle v_1, v_2, ..., v_n \rangle$ 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 $fin[v]$ se asigna después de haber explorado por todos los vértices u alcanzables desde v, se cumple que $fin[u] < fin[v]$. Por ende, si se ordenan los vértices por valor fin decreciente, los vértices que se pueden alcanzar desde otros aparecerán después. □

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 Kosaraju

El algoritmo de Kosaraju 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 CFCs*[7] C *y* C' *distintas en* $G = (V, E)$*, ningún vértice de* C *es alcanzable desde* C'*, o viceversa.*

Demostración. Esto se puede demostar 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* $u \to v$ *tal que* $u \in C$ *y* $v \in C'$*, entonces* $\mathit{fin}[C] > \mathit{fin}[C']$[8]*.*

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 $\mathit{fin}[w]$, y por ende $\mathit{fin}[C] > \mathit{fin}[C']$.
- 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 $\mathit{fin}[C] > \mathit{fin}[C']$.

□

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

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 $\mathit{fin}[C]$ máximo, ninguno de sus vértices contiene una arista hacia otra CFC C' (porque en ese caso $\mathit{fin}[C] < \mathit{fin}[C']$). Además, como todos los vértices de C son alcanzables desde v, la CFC se construye

[7] Componentes Fuertemente Conexas
[8] El *fin* de una CFC se define como $\max\{\mathit{fin}[v] \mid v \in C\}$.

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.

□

Capítulo 3

Árbol Generador Mínimo

3.1. Definición

Grafos pesados

Un *grafo pesado* es un grafo $G = (V, E)$ junto con una función $w : E \longrightarrow \mathbb{N}$, donde $w(e)$ es el peso de la arista e. El peso suele usarse para modelar diversas magnitudes, como el costo de una arista, o tiempo que se tarda en pasar por ella.

AGM

Un *árbol generador mínimo* (AGM) es un árbol generador cuyo peso total es menor al de todos los demás AGs, donde peso total se define como la suma de los pesos de sus aristas. Formalmente, el problema es el siguiente:

Problema:
Dado un grafo $G = (V, E)$ y una función de peso $w : E \longrightarrow \mathbb{N}$, encontrar un AG $T = (V, E_T)$ que cumpla:

$$w(T) = \sum_{e \in E_T} w(e) \leq \sum_{e \in E_{T'}} w(e) = w(T')$$

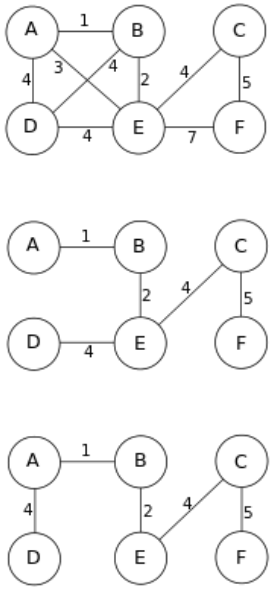
Para cualquier árbol generador $T' = (V, E_{T'})$ en G.

También se puede definir el árbol generador máximo, que maximiza el peso total de T, pero este problema se puede reducir al anterior: si se toman los pesos $w'(e) = -w(e)$, el peso de cualquier AG T será $w'(T) = \sum_{e \in E_T} w'(e) = \sum_{e \in E_T} -w(e) = -w(T)$, y por ende:

$$\begin{aligned} w'(AGM) \leq w'(T) &\iff -w'(AGM) \geq -w'(T) \\ &\iff w(AGM) \geq w(T) \end{aligned}$$

Propiedades

El AGM de un grafo no necesariamente es único: pueden varios grafos con el mismo peso, que es menor al de todos los demás.



Un grafo y 2 de sus AGMs, ambos con peso total 16.

Algoritmos

En las siguientes secciones analizaremos 2 algoritmos para encontrar un AGM: el algoritmo de Prim y el de Kruskal. Ambos son golosos: siguen estrategias que toman en cada paso la mejor decisión a corto plazo, y en este caso devuelven soluciones óptimas. Logran esto armando progresivamente un bosque en un ciclo que mantiene el invariante de que sus aristas forman un subconjunto de las de algún AGM.

ARBOL-GENERADOR-MINIMO(G, w)
1 Inicializar grafo $T = (V, \emptyset)$
2 **while** T no es un AG (es disconexo):
3 Encontrar una arista segura e.
4 Agregar e a E_T.
5 **return** T

En este contexto una arista es *segura* cuando se puede agregar al bosque T y mantener el invariante (es un subconjunto de algún AGM). Encontrar una es donde radica la dificultad del problema, y es en donde difieren los algoritmos.

3.2. Algoritmo de Prim

3.2.1. Introducción

El *algoritmo de Prim* mantiene un árbol $T = (V_T, E_T)$, y en cada iteración agrega un nuevo vértice de $V - V_T$ al mismo (junto con una arista a E_T), hasta que T pasa por todos los vértices de G (y se vuelve un AG). La arista seleccionada en cada paso es la de mínimo peso entre todas las $(u, v) \in V_T \times (V - V_T)$ (las que conectan vértices dentro del árbol con los de afuera).

Se puede implementar de la siguiente manera:

```
PRIM(G, w)
1  V_T = {s} (cualquier vértice)
2  E_T = ∅ for i = 1 to n − 1
3        e = arg mín {w(e) | e = uv ∈ E, u ∈ V_T, v ∈ V − V_T}
4        E_T = E_T ∪ e
5        V_T = V_T ∪ v
6  return T = (V_T, E_T)
```

El algoritmo asume que el grafo es conexo. Si esto no se cumple, se puede modificar el algoritmo levemente[1] para que devuelva el AGM de la componente conexa que contiene al vértice inicial.

3.2.2. Correctitud

Teorema. *El árbol que devuelve el algoritmo de Prim es un AGM.*

Demostración. Se demuestra por inducción. La hipótesis inductiva será la siguiente: $\forall\, 0 \leq k \leq n-1, T_k = (V_k, E_k)$, el grafo mantenido por Prim en la k-ésima iteración del ciclo, es un árbol y es subgrafo de algún AGM de G.

Caso Base: En el paso $k = 0$, todavía no se entró al ciclo, así que $T_0 = (\{s\}, \emptyset)$, que es un árbol (no tiene ciclos) y es subgrafo de cualquier AG (y por lo tanto de cualquier AGM).

Paso Inductivo: Sea $e = vw$ agregada en la iteración. Luego, $T_k = (V_k, E_k) = (V_{k-1} \cup \{w\}, E_{k-1} \cup \{e\})$. Por hipótesis inductiva, se sabe que T_{k-1} es subgrafo de algún árbol generador T. Existen dos posibilidades:

- $e \in E_T$, en cuyo caso T_k es también subgrafo de T.
- $e \notin E_T$. Aún así, sabemos que V_{k-1} forma una componente conexa en T, porque T_{k-1} es un subgrafo suyo y es conexo (es un árbol). Entonces, debe haber alguna arista $e' = v'w' \in E_T$ que conecta V_{k-1} y $V - V_{k-1}$. Si se saca esa arista de T, las componentes se vuelven disconexas, pero se pueden volver a conectar usando la arista e.

  Por ende, $T' = T + e - e' = (V, (E_T \cup \{e\}) - \{e'\})$ es un árbol (y es generador, ya que contiene a todos los vértices). Además, como e' era una de las aristas que se podían elegir al

[1] El único problema que tiene esa implementación es que asume que el conjunto del que se toma el arg mín no está vacío. Si esto se chequea en cada iteración, se puede terminar la ejecución cuando no quedan vértices por agregar.

principio de la iteración (conecta a V_{k-1} con $V - V_{k-1}$), se debe cumplir que $w(e) \leq w(e')$, así que:

$$w(T') = w(T) + \underbrace{w(e) - w(e')}_{\leq 0} \leq w(T)$$

Por lo tanto, el peso total de T' es menor o igual al de un AGM, así que este también es AGM. Como T_k es subgrafo de T' ($E_k = E_{k-1} \cup \{e\} \subseteq E_T \cup \{e\}, e' \notin E_k$), se cumple la propiedad.

Dado que la propiedad vale al terminar cualquier iteración, también se cumple en $k = n-1$, donde T_{k-1} tiene n vértices, y por lo tanto es un árbol generador (y mínimo).

□

3.2.3. Complejidad

Para implementar el algoritmo de Prim, es necesario determinar un método para encontrar la arista de menor peso entre V_T y $V - V_T$. Esto se puede lograr utilizando una cola de prioridad, pero la forma en la que esta se represente puede variar:

- Un arreglo de $|V|$ posiciones: En este caso, el peso de la arista que conecta a cada vértice con el árbol se guarda en una posición del arreglo (los vértices que ya se agregaron se marcan con un valor especial). Para determinar el siguiente vértice a agregar, se toma el mínimo del arreglo, y una vez que se agrega se actualizan las posiciones correspondientes (en caso de que uno de sus vecinos se pueda alcanzar con peso menor al anterior). Ambas operaciones son $\mathcal{O}(|V|)$, y como se realizan $|V| - 1$ operaciones, la complejidad total $\mathcal{O}(|V|^2)$.

- Un heap: En este caso, se mantiene un mín-heap de los nodos que no fueron explorados, donde la clave es el peso mínimo entre las aristas que lo conectan al árbol. En cada paso se desencola un vértice, y se actualizan las claves de sus vecinos[2]. Esto implica que se realizan $1 + d(v)$ operaciones logarítmicas en cada iteración, lo cual resulta en una complejidad total de $\mathcal{O}((|V| + |E|) \log |V|)$. Como se asume que G es conexo, $|E| \geq |V| - 1$, y por ende $\mathcal{O}((|V| + |E|) \log |V|) \subseteq \mathcal{O}(|E| \log |V|)$.

En el caso de los grafos *densos*, que se podrían definir como aquellos en los que $|E| \in \Omega(|V|^2)$, la segunda alternativa tendría complejidad $\mathcal{O}(|V|^2 \log |V|)$, mientras que la primera solo $\mathcal{O}(|V|^2)$. Por otro lado, si el grafo es *ralo*, que sería cuando $|E| \in \mathcal{O}(|V|)$, la segunda sería tan solo $\mathcal{O}(|V| \log |V|)$, mejor que $\mathcal{O}(|V^2|)$.

Hay una tercera opción para representar la cola de prioridad: a través de Fibonacci heaps. Estos permiten que el algoritmo corra en $\mathcal{O}(|E| + |V| \log |V|)$ (asintóticamente mejor que las dos anteriores), pero es más complejo de implementar que las otras opciones.

[2] Para lograr esto en tiempo logarítmico, es preferible usar un árbol balanceado (como los AVLs) antes que un heap.

3.3. Algoritmo de Kruskal

3.3.1. Disjoint-Set/Union-Find

Antes de ver el algoritmo en sí, es necesario una de las estructuras que usa: el *Disjoint-Set* o *Union-Find*. Esta mantiene una partición $P_1, ..., P_n$ de un conjunto C, asignándole a todos los elementos de cada P_i un mismo representante r_i. Permite realizar las siguientes operaciones de forma eficiente:

- MAKE-SET(x): crea un nuevo conjunto $\{x\}$ dentro de la partición. Como precondición, x no puede estar en ninguno de los conjuntos anteriores.
- UNION(x, y): Reemplaza los conjuntos que contenían a x e y, S_x, S_y por su unión $S_x \cup S_y$.
- FIND(x): Devuelve el representante r_x de x dentro de la partición. Esto puede ser utilizado para comprobar si dos elementos pertenecen al mismo conjunto.

Implementación

Internamente, la estructura se implementa como un bosque, donde cada árbol se corresponde con el conjunto de la partición que contiene a sus vértices, y la raíz es el representante. Implementar la operación UNION es simple: solo es necesario agregar a la raíz de uno de los conjuntos como hijo de la raíz del otro. FIND también resulta trivial: solo es necesario recorrer los antecesores de cada nodo hasta llegar a la raíz.

Esto alcanza para implementar a la estructura, pero la complejidad no es ideal: los árboles podrían ser degenerados (una línea donde cada nodo tiene 1 solo hijo), lo cual haría que FIND sea $\mathcal{O}(n)$. Para evitar que suceda esto, se emplean 2 heurísticas:

- **Union by rank**[3]: Esto implica mantener la altura de cada árbol guardado, lo cual en este caso no es costoso: la única forma en la que puede cambiar es cuando se agrega otro árbol T a la raíz como hijo, y en tal caso la altura pasa a ser el mínimo entre la anterior y $h(T)+1$. Esto se utiliza a la hora de decidir qué raíz utilizar para el nuevo árbol de UNION: se elije la que resulte en menor altura (queda como raíz el representante de mayor rango).
- **Path compression**: Esta optimización se basa en el hecho de que la estructura interna del árbol no es rígida: solo se debe cumplir que la raíz sea el representante de todos los nodos. Por ende, cada vez que se realiza un FIND, todos los vértices transitados se pueden colocar como hijos directos del árbol.

Los algoritmos se pueden implementar de la siguiente manera, donde *prev* es el arreglo que guarda los antecesores de cada nodo (representa al bosque) y *rank*[v] guarda la altura del árbol con raíz en v:

```
MAKE-SET(x)
1  prev[x] = x
2  rank[x] = 0
```

[3]Es equivalente usar la alternativa **union by size**, que usa la cantidad de nodos en el árbol en lugar de la altura.

```
UNION(x, y)
1  r_x = FIND(x), r_y = FIND(y)
2  if rank[r_x] > rank[r_y]
3        prev[y] = x
4  else
5        prev[x] = y
6        if rank[x] == rank[y]
7              rank[y] = rank[y] + 1
```

```
FIND(x)
1  if x ≠ prev[x]
2        prev[x] = FIND(x)
3  return prev[x]
```

Haciendo uso de ambas heurísticas, la complejidad de peor caso amortizada de la operación FIND (y por ende también la de UNION) baja a $\mathcal{O}(\alpha(n))$, donde α es la función de Ackermann inversa[4].

3.3.2. Algoritmo

Teniendo esta estructura, se puede definir el *algoritmo de Kruskal*: se construye un bosque, recorriendo todas las aristas en orden de peso creciente, y mientras tanto se mantiene un Union-Find con las componentes conexas del bosque. Cada vez que se visita una arista, se chequea si conecta dos vértices de componentes conexas distintas. Si es así, se corre UNION sobre los conjuntos de las componentes, y la arista es agregada al árbol. Esto se hace hasta que queda una sola componente, y queda formado un árbol generador (que además es mínimo).

```
KRUSKAL(G, w)
1  T = (V, E_T = ∅)
2  for each v ∈ V
3        MAKE-SET(v)
4  for each uv ∈ E_T, en orden de w creciente
5        if FIND(u) ≠ FIND(v)
6              E_T = E_T ∪ uv
7              UNION(u, v)
8        return T
```

3.3.3. Correctitud

La correctitud de este algoritmo se puede demostrar de forma similar al anterior:

Teorema. *El árbol que devuelve el algoritmo de Kruskal es un AGM.*

Demostración. Se demuestra por inducción con la siguiente hipótesis inductiva: $\forall\, 0 \leq k \leq n-1, T_k = (V_k, E_k)$, el grafo mantenido por Kruskal después de agregar la k-ésima arista, es un bosque y subgrafo de algún AGM de G.

[4] A efectos prácticos, $\alpha(n) \leq 5$

Caso Base: En $k = 0$, el grafo es $T_0 = (V, \emptyset)$, que es subgrafo de cualquier AG, y no tiene ciclos (así que es un bosque).

Paso Inductivo: Sea $uv \in E$ la k-ésima arista agregada al grafo. Por hipótesis inductiva, sabemos que T_{k-1} es un bosque, y como las aristas deben conectar componentes distintas, la nueva arista no forma parte de ningún ciclo, así que T_k sigue siendo un bosque.

Por otro lado, la HI también implica que T_{k-1} es subgrafo de algún AGM T. Como los árboles son conexos, debe haber algún camino de T entre los vértices u y v. Si este camino es solo la arista uv, T_k también es un subgrafo del árbol. Por otro lado, si el camino es distinto, seguro contiene una arista $u'v'$ que conectaría el componente de u en T_{k-1} con algún otro (porque v no es alcanzable desde u en T_{k-1}). Eso implica que $w(uv) \leq w(u'v')$, ya que uv es la arista de menor peso que cumple esa propiedad.

Si se toma $T' = (V, (E \cup \{uv\} - \{u'v'\}))$, T' es un árbol porque agregar uv genera un único ciclo que contiene a $u'v'$, y sacar esta arista lo rompe. Por otro lado, su peso es:

$$w(T') = w(T) + \underbrace{w(uv) - w(u'v')}_{\leq 0} \leq w(T)$$

Por ende, T' es un AGM y, como T_k es un subgrafo de T', se cumple la propiedad.

□

3.3.4. Complejidad

El algoritmo de Kruskal tiene dos pasos generales: ordenar las aristas, y recorrerlas hasta armar el árbol. El ordenamiento es una operación $\mathcal{O}(|E| \log |E|)$, mientras que el ciclo realiza a lo sumo $|E|$ iteraciones, y en cada se hace una llamada a UNION, que tiene complejidad $\mathcal{O}(\alpha|V|)$. Aprovechando que $|E| \in \mathcal{O}(|V|^2)$, se tiene $\mathcal{O}(|E| \log |E|) \subseteq \mathcal{O}(|E| \log (|V|^2)) = \mathcal{O}(|E| \log |V|)$. La complejidad final entonces es:

$$\mathcal{O}(|E| \log |V| + |E|\alpha(|V|)) = \mathcal{O}(|E| \log |V|)$$

3.4. Camino Minimáx

3.4.1. Definición

Dado un grafo pesado $G = (V, E)$ con $w : E \longrightarrow \mathbb{N}$, el *camino minimáx* entre un par de vértices $u, v \in V$ es aquel que minimiza el peso de la arista más pesada del camino. Formalmente, es el P_m que cumple:

$$P_m = \arg\min \{\max \{w(e) \mid e \in P\} \mid P \text{ camino entre } u \text{ y } v\}$$

Análogamente, el camino maximín es aquél que maximiza el peso de la arista menos pesada del camino, es decir, el P_M tal que:

$$P_M = \arg\max \{\min \{w(e) \mid e \in P\} \mid P \text{ camino entre } u \text{ y } v\}$$

Esta noción tiene varias aplicaciones: en general se la conoce como "ancho de banda". Podría utilizarse para modelar la cantidad de datos que se pueden transmitir entre dos terminales pasando por una red con ciertas capacidades[5], o la cantidad de peso que puede pasar por una red de puentes con límites dados.

3.4.2. Solución

El método para encontrar un camino minimax entre dos vértices es simple:

Teorema. *Dado un grafo pesado $G = (V, E)$ con $w : E \longrightarrow \mathbb{N}$ y un AGM T, para cualquier par de vértices $u, v \in V$, el camino que conecta a u con v en T es minimax.*

Demostración. Supongamos que existe un camino minimax P que pasa por aristas que no están en el árbol T. Sea $uv \in E \cap P - E_T$, y T_{uv} el camino que conecta sus vértices en el árbol. Luego, si se toma $e' = \arg\max\{w(e) \mid e \in T_{uv}\}$, se puede ver que $w(uv) \geq w(e')$, ya que si no e' podría ser reemplazada por uv, formando un AG $T \cup \{uv\} - \{e'\}$ con un peso estrictamente menor, lo cual es absurdo porque T es AGM.

Por ende, cualquier arista fuera del árbol puede ser reemplazada por un camino que pasa por el árbol y tiene aristas de menor o igual peso (así que el peso máximo del camino se mantiene igual). Entonces, queda demostrado que siempre existe un camino minimáx que pasa por el árbol.

□

Esto significa que, para encontrar el camino minimáx entre cualquier par de vértices, se puede calcular el AGM del grafo, y tomar el camino que los une dentro de este. Para el camino maximín, se puede tomar el árbol generador máximo (la demostración es análoga).

[5] Con la restricción de que todos los datos pasan por un mismo camino; si se pueden "dividir", es un problema de flujo.

Capítulo 4

Camino Mínimo

4.1. Definición

Dado un digrafo[1] pesado $G = (V, E, w)$, el peso de un camino P entre sus vértices se define como la suma de los pesos de sus aristas:

$$w(P) = \sum_{e \in P} w(e)$$

Entonces, dados dos vértices u y v, existe un camino de peso mínimo, es decir uno con el peso:

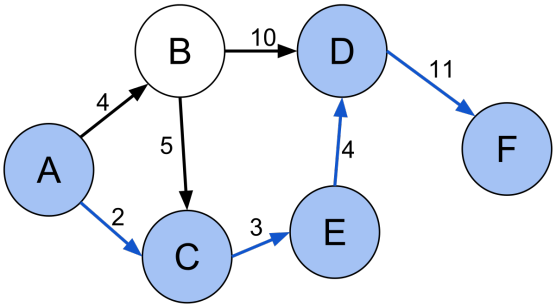
$$\delta(v, w) = \text{mín}\,\{w(P) \mid P \text{ es un camino entre } v \text{ y } w\}$$

La función $\delta(v, w)$ sigue reglas similares a $d(v, w)$, es decir:

- $\delta(v, v) = 0 \ \forall v \in V$
- $\delta(u, v) = \infty \iff u$ no es alcanzable desde v

P es un *camino mínimo* cuando $w(P) = \delta(v, w)$, y no necesariamente es único.

[1] Todos los algoritmos de camino mínimo que analizamos pueden ser fácilmente adaptados para operar en grafos no dirigidos.



El camino mínimo entre el par de vértices A y F.

Este mínimo existe porque el conjunto de caminos es finito: a lo sumo hay $n!$, uno por cada permutación de los vértices de G. Esto no necesariamente vale en el caso de recorridos, que pueden tener vértices repetidos.

Un problema análogo es el de *camino máximo*, que es el camino de peso máximo entre un par de vértices.

4.1.1. Existencia

A pesar de que un **camino** mínimo entre u y v siempre existen (asumiendo que están conectados), no es el caso para los recorridos: si un grafo tiene algún *ciclo de peso negativo* (o simplemente *ciclo negativo*) alcanzable desde u y v, este puede ser transitado múltiples veces para obtener un peso tan bajo como se desee.

4.1.2. Camino mínimo elemental

El problema de esta índole más simple de definir (y más general) es el de *camino mínimo elemental*:

> **Problema:**
> Dado un (di)grafo $G = (V, E, w)$ y un par de vértices $v, u \in V$, encontrar el camino (sin vértices repetidos) de peso mínimo.

Como establecimos antes, este problema esta bien definido para cualquier grafo y cualquier par de vértices. El problema de *camino máximo elemental*, donde se busca el camino de peso máximo, se puede reducir fácilmente a este: basta con tomar una función de peso $w'(e) = -w(e)$[2].

Para el caso general, este problema es NP-hard, así que se suelen estudiar casos restringidos. La restricción más importante para nuestros algoritmos es que el grafo no tenga ciclos negativos. En ese caso, encontrar un camino mínimo es equivalente a encontrar un recorrido mínimo.

[2] A pesar de que los problemas son equivalentes, las aplicaciones de camino mínimo suelen ser en casos donde los pesos son no negativos (y por ende el grafo no tiene ciclos negativos). En tal caso, encontrar un camino máximo se vuelve más difícil, ya que negar los pesos hace que casi cualquier ciclo sea negativo. Es por esto que camino máximo se considera "más difícil".

Teorema. *Si un (di)grafo $G = (V, E, w)$ no tiene ningún ciclo negativo, existe un camino entre cualquier par de vértices $v, u \in V$ con peso total menor al de cualquier recorrido.*

Demostración. Supongamos que existe algún recorrido $R = v \cdots u$ que no es un camino y cuyo peso $w(R)$ es menor a la longitud de todos los caminos entre v y u. Debido a que no es camino, tiene que pasar por dos veces por algún vértice z, definiendo un ciclo $R_{z_1 z_2}$[3]. Como el grafo no tiene ciclos negativos, $w(P_{z_1 z_2}) \geq 0$, y entonces se puede definir un nuevo recorrido $R' = R_{vz_1} + R_{z_2 u}$ que "recorta" el ciclo, y cumple $w(R') = w(R) - w(R_{z_1 z_2}) \leq w(R)$. Este proceso puede repetirse hasta conseguir un camino de peso menor o igual al recorrido original, lo cual contradice la suposición inicial. □

Subestructura óptima

El problema de encontrar un camino mínimo P_{uv} entre un par de vértices u y v cumple el princpio de optimalidad de Bellman: Cualquier subcamino $P_{u'v'} \subseteq P_{uv}$ es a su vez un camino mínimo entre u' y v'.

Teorema. *Dado un digrafo pesado $G = (V, E, w)$ sin ciclos negativos y un camino mínimo $P = v_1 \cdots v_k$, el subcamino $P_{v_i v_j}$ es un camino mínimo entre v_i y v_j.*

Demostración. Supongamos que $P_{v_i v_j}$ no es un camino mínimo. Eso implica que existe un camino alternativo $P' = v_i \cdots v_j$ tal que $w(P') < w(P_{v_i v_j})$. En tal caso, se podría construir el recorrido $R = P_{v_1 v_i} + P' + P_{v_j v_k}$, para el cual existen dos posibilidades:

- Si el recorrido R es un camino, entonces $w(R) = w(P_{v_1 v_i}) + w(P') + w(P_{v_j v_k}) < w(P_{v_1 v_i}) + w(P_{v_i v_j}) + w(P_{v_j v_k}) = w(P)$, lo cual es absurdo porque P es un camino mínimo.
- Si el recorrido R no es un camino, hay algún vértice v_a por el que pasa 2 veces. Esto forma un ciclo $C = R_{v_{a1} v_{a2}}$ dentro del recorrido, que por hipótesis debe tener peso no negativo. Por lo tanto, si se "recorta" el ciclo, se obtiene un recorrido $R' = R_{v_1 v_{a1}} R_{v_{a2} v_k}$ de peso menor o igual. Estos recortes se pueden repetir hasta obtener un camino que, al igual que en el caso anterior, tiene peso estrictamente menor a P (**Absurdo**).

Como ambas alternativas llevan a una contradicción, la suposición inicial ($P_{v_i v_j}$ no es camino mínimo) es falsa. □

4.1.3. Árbol de caminos mínimos

Ahora que definimos las restricciones necesarias (no hay ciclos negativos), se pueden empezar a analizar los métodos para encontrar caminos mínimos. Sorprendentemente, el problema de encontrar un camino mínimo entre v y u parece ser igual de difícil que el de encontrar el camino mínimo entre v y todos los vértices del grafo, ya que no se conoce ningún algoritmo que resuelva el primero con una complejidad de peor caso estrictamente menor a los métodos para resolver el otro.

Este nuevo problema se llama *camino mínimo con un origen y múltiples destinos* (SSSP[4]), y los caminos definen un árbol enraizado en v, llamado el *árbol de caminos mínimos* (ACM). Un v-

[3] z_1 y z_2 no son vértices distintos, sino formas de distinguir las apariciones de z en R.
[4] *Single Source Shortest Path.*

ACM es análogo a un árbol v-geodésico, solo que en este caso se cumple $\delta_T(v,u) = \delta_G(v,u)\ \forall u \in V$.

4.2. Algoritmo de Dijkstra

El *algoritmo de Dijkstra* es uno de los más famosos en todo el área de computación. Este resuelve el problema de camino mínimo uno a todos, bajo la restricción de que los pesos de las aristas son no negativos.

4.2.1. Definición

El procedimiento general es similar al de Prim: se mantiene un árbol enraizado en el vértice inicial v junto con una "frontera" de vértices adyacentes a los del árbol. En cada paso, se agrega el vértice de la frontera más cercano a la raíz. Esto puede ser clasificado como goloso, ya que en cada iteración se toma la decisión más ventajosa a corto plazo. Extendiendo la comparación, mientras que Prim agrega la arista segura de costo mínimo, Dijkstra agrega la que conecta al vértice con menor distancia a v.

DIJKSTRA(G, w, s)
1 $T = (V_T = \{s\}, E_T = \emptyset)$
2 Inicializar arreglo d con $+\infty$ en cada posición.
3 $d[s] = 0$
4 **for** $i = 1$ **to** $n - 1$
5 $\quad u \to v = \arg\min \{d[x] + w(x \to y) \mid x \in V_T, y \in V - V_T\}$
6 $\quad d[v] = d[u] + w(u \to v)$
7 $\quad V_T = V_T \cup \{v\}$
8 $\quad E_T = E_T \cup \{u \to v\}$
9 **return** (T, d)

El diccionario d contiene la distancia mínima entre s y cada vértice. Esto se puede utilizar, entre otras cosas, para obtener el camino mínimo a cualquier vértice u en tiempo lineal: se puede revisar el vecindario de entrada $N^-(u)$ hasta encontrar un vértice $v \in N^-(u)$ tal que $d[v] + w(v \to u) = d[u]$. Esto significa que hay un camino mínimo que pasa por la arista $v \to u$, y el proceso se puede repetir hasta llegar a s.

El algoritmo puede modificarse para grafos disconexos: debe mantener un conjunto de vértices conectados, y frenar cuando no quedan nuevos vértices por explorar. El comportamiento en ese caso es devolver los caminos mínimos a los vértices alcanzables (los demás están a distancia ∞).

Generalización

El procedimiento se puede generalizar para cualquier función $\bullet : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$ no decreciente de forma que el costo $w_\bullet(P)$ de un camino $P = v_1 \cdots v_k$ sea:

$$w_\bullet(v_1 \cdots v_k) = \begin{cases} 0 & \text{si } k = 1 \\ w_\bullet(v_1 \cdots v_{k-1}) \bullet c(v_{k-1}v_k) & \text{en caso contrario} \end{cases}$$

La restricción de ser *no decreciente* significa que $w_\bullet(P + y) \geq w_\bullet(P)$ para todo camino P y vértice y. La suma cumple esto solo cuando los pesos son no negativos.

Si se define $\bullet$-peso de un camino P al valor $w_\bullet(P)$, se puede llamar camino $\bullet$-mínimo al camino entre v y u con menor $\bullet$-peso (y $\delta_\bullet(v, u)$ al peso de ese camino). Más aún, el árbol de caminos $\bullet$-mínimos de v es uno en el cual $\delta_{G,\bullet}(v, u) = \delta_{T,\bullet}(v, u)$ para todo u.

Finalmente, si se tiene un $\bullet$-ACM T enraizado en v, se puede definir $w_\bullet(x \to v) = w_\bullet(P) \bullet w(x \to y)$, donde P es el camino de T entre v y x. Esto permite diseñar el siguiente algoritmo general:

DIJKSTRA-$\bullet(G, w, s)$
1 $T = (V_T = \{s\}, E_T = \emptyset)$
2 **for** $i = 1$ **to** $n - 1$
3 $\quad u \to v = \arg\min \{w_\bullet(x \to y) \mid x \in V_T, y \in V - V_T\}$
4 $\quad V_T = V_T \cup \{v\}$
5 $\quad E_T = E_T \cup \{u \to v\}$
6 **return** (T, d)

Bajo esta definición, el algoritmo de Prim es solo una versión de DIJKSTRA-$\bullet$, donde la función $\bullet(x, y)$ es $\max\{x, y\}$.

4.2.2. Correctitud

Se podría demostrar la correctitud de la función DIJKSTRA-$\bullet$, pero para el final es más útil el caso particular de camino mínimo.

Teorema. *Dado un (di)grafo pesado sin ciclos negativos $G = (V, E, w)$ y un origen $s \in V$, el algoritmo de Dijkstra devuelve un s-ACM T, y $d[u] = \delta(v, u)$ $\forall u \in V$.*

Demostración. Se procede demostrando la siguiente hipótesis inductiva: para cada $0 \leq k \leq n-1$, el grafo $T_k = (V_k, E_k)$ mantenido por Dijkstra en la k-ésima iteración es un árbol que cumple $\delta_{T_k}(s, u) = \delta_G(s, u)$ $\forall u \in V_k$, y además $d[u] = \delta_G(s, u)$ $\forall u \in V_k$.

Caso Base: Antes de la primera iteración, $T_0 = (\{s\}, \emptyset)$, y se cumple que $d[s] = 0 = \delta(s, s)$ (por definición).

Paso Inductivo: Llamemos $u \to v$ el arco que se agrega al árbol en la k-ésima iteración, es decir, $T_k = (V_k, E_k) = (V_{k-1} \cup \{u\}, E_{k-1} \cup \{u \to v\})$. Es claro que T_k sigue siendo un árbol, ya que la nueva arista conecta a un vértice que no estaba en el grafo (y por ende no puede formar un ciclo).

Sea $P = s \cdots y$ un camino mínimo entre s e y, es decir, $w(P) = \delta(s, y)$. Tomamos el último vértice x del camino que pertenece a V_{k-1}, y sea y el que lo sigue (estos deben existir, porque x está en el conjunto, e y no). Como se demostró anteriormente, $w(P_{su}) = \delta(s, u)$ (es subcamino de un camino mínimo). Como la arista $x \to y$ conecta un vértice de V_{k-1} con uno de $V - V_{k-1}$, se debe cumplir que:

$$d[u] + w(u \to v) \leq d[x] + w(x \to y) \qquad (x \to y \text{ fue elegida por Dijkstra})$$
$$\delta(s, u) + w(u \to v) \leq \delta(s, x) + w(x \to y) \qquad (\text{Por HI, ya que } u, x \in T_{k-1})$$

Entonces, se tiene que el camino une a s con v en T_k, P' tiene un peso

$$w(P') = \delta(s,u) + w(u \to v) \leq \delta(s,x) + w(x \to y) \underset{w \geq 0}{\leq} \underbrace{\delta(s,x) + w(x \to y) + w(P_{yv})}_{= w(P), \text{ que es un CM}} = \delta(s,v)$$

Así que el camino P' es mínimo, y T_k cumple que las distancias de sus vértices son las mínimas en el grafo. Por otro lado, el nuevo valor de $d[v] = d[u] + w(u,v)$ es la distancia mínima de s a v.

□

4.2.3. Complejidad

La complejidad de Dijkstra es análoga a la de Prim: solo es necesario mantener un arreglo adicional con las distancias a cada nodo, lo cual no impone ningún costo en términos de tiempo. Por ende, el tiempo de ejecución dependerá de como se implemente la cola de prioridad de vértices a agregar:

- Arreglo: $\mathcal{O}(|V|^2)$.
- Binary Heap o AVL: $\mathcal{O}(|E| \log |V|)$.
- Fibonacci Heap: $\mathcal{O}(|E| + |V| \log |V|)$.

4.3. Bellman-Ford

El *algoritmo de Bellman-Ford*, al igual que el de Dijkstra, resuelve el problema de camino mínimo de uno a todos. Sin embargo, a diferencia de este, Bellman-Ford puede ser utilizado en (di)grafos con pesos negativos (pero sin ciclos negativos). Si el grafo tiene algún ciclo alcanzable desde el vértice de origen, Bellman-Ford es capaz de detectarlo.

4.3.1. Definición

Este algoritmo se basa en "*relajar*" las aristas del grafo. La relajación de $u \to v$ implica comprobar si el camino mínimo que va del origen a v puede ser mejorado pasando por la arista. Si las distancias mínimas $\delta(s,v)$ se guardan en $d[v]$, se peude expresar de la siguiente manera:

RELAJAR$(u \to v, w)$
1 $\quad d[v] = \text{mín}\{d[u] + w(u \to v), d[v]\}$

Bellman-Ford repite esta operación para cada arista, $|V| - 1$ veces. En cada iteración i ciclo, los vértices se conectan a s por caminos de a lo sumo i aristas.

BELLMAN-FORD(G, w, s)

```
1  Inicializar arreglo de distancias d con +∞ en cada posición
2  d[s] = 0
3  for i = 0 to n − 1
4       for each e ∈ E
5            RELAJAR(e, w)
6  for each u → v ∈ E
7       if d[u] + w(u → v) < d[v]
8            return Hay un ciclo negativo
9  return d
```

El último ciclo chequea si existe algún ciclo negativo. Esto se debe a que, si alguna arista se puede relajar después de i iteraciones, el recorrido más corto entre s y algún vértice tiene $|V|$ aristas, lo cual implica que contiene un ciclo negativo (se demuestra más adelante).

El algoritmo puede adaptarse para devolver un s-ACM: basta con actualizar el padre de v en RELAJAR, es decir, asignar $prev[v] = u$.

4.3.2. Correctitud

Primero analizamos el caso de grafos sin ciclos negativos:

Teorema. *Dado un (di)grafo pesado $G = (V, E, w)$ sin ciclos negativos, el algoritmo de Bellman-Ford devuelve en d las distancias $d[v] = \delta(s, v)$.*

Demostración. Para demostrar esto, se demuestra la siguiente propiedad hipótesis inductiva: para cada iteración k del ciclo exterior de Bellman-Ford, si un vértice v está conectado a s por un camino mínimo P tal que $|P| \leq k$, se tiene que $d_k[v] = \delta(s, v)$, y $d_k[u] \geq \delta(s, u)\ \forall v \in V$.

Caso Base: Para $k = 0$, el único vértice conectado por un camino de longitud 0 es s, y se cumple que $d[s] = \delta(s, s) = 0$

Paso Inductivo: Primero, demuestro la parte auxiliar de la hipótesis inductiva: sabiendo que $d_{k-1}[u] \geq \delta(s, u)\ \forall v \in V$, se puede analizar la asignación de RELAJAR (es el único procedimiento que cambia d en el algoritmo):

$$d_k[u] = \text{mín}\left\{d_{k-1}[u], d_{k-1}[u'] + w(u' \to u)\right\}$$

Podemos ver que ambas opciones cumplen la propiedad: $d_{k-1}[u] \geq \delta(s, u)$ por HI, mientras que: $d_{k-1}[u'] + w(u' \to u) \geq \delta(s, u') + w(u' \to u) \geq \delta(s, u)$ (desigualdad triangular).

Luego, por hipótesis Sabemos que en el paso $k-1$-ésimo se cumplía que $d_{k-1}[v] = \delta(s, v)$ para cualquier vértice v con camino mínimo de longitud menor o igual a $k - 1$. Esto se sigue cumpliendo en d_k para esos mismos vértices gracias a que, como RELAJAR asigna:

$$d_k[v] = \text{mín}\left\{d_{k-1}[v], d_{k-1}[u] + w(u \to v)\right\}$$

Se tiene que $d_k[v] \leq d_{k-1}[v] = \delta(v, s)$. Como demostramos previamente, $d_k[v] \geq \delta(v, s)$, así que $d_k[v] = \delta(v, s)$.

Por otro lado, para aquellos vértices z tales que el camino mínimo de s a z es $P = s \cdots z$ de longitud $|P| = k$, sea z' el anteúltimo vértice. Debido a que $P_{sz'} = s \cdots z'$ es también camino mínimo (es subcamino de un camino mínimo) y tiene longitud $k-1$, sabemos por HI que $d_{k-1}[z'] = \delta(s, z')$. Esto, implica que $d_{k-1}[z'] + w(z' \to z) = \delta(s, z') + w(z' \to z) = w(P) = \delta(s, z)$. Por lo tanto, en la asignación:

$$d_k[z] = \text{mín}\{d_{k-1}[z], d_{k-1}[z'] + w(z' \to z)\}$$

Como $d_{k-1}[z] \geq \delta(s, z)$, se tiene $d_k = \delta(s, z)$.

Finalmente, habiendo demostrado la inducción, nos sirve un caso particular de la misma: después de la última iteración, todos los vértices conectados al origen por un camino mínimo de longitud máxima $|V| - 1$ tendrán los pesos asignados de forma correcta. Estos son pesos son las distancias $\delta(s, v)$, porque en caso contrario el recorrido mínimo tendría que tener longitud mayor a $|V| - 1$, y por ende repetir algún vértice (esto no sucede en grafos sin ciclos negativos).

□

El comportamiento cuando hay ciclos negativos también es correcto:

Teorema. *Si un (di)grafo pesado $G = (V, E, w)$ tiene algún ciclo de peso negativo C alcanzable desde un origen s, el algoritmo de Bellman-Ford lo detecta.*

Demostración. Primero, demuestro la siguiente hipótesis inductiva: en la k-ésima iteración de relajaciones, cualquier vértice u con distancia[5] a s menor o igual a k cumple $d_k[v] < \infty$

Caso Base: El único vértice alcanzable desde s a través de 0 aristas es s en sí, y $d[s] = 0 < \infty$.

Paso inductivo: Si la propiedad vale para $k-1$, entonces $d_{k-1}[u] < \infty$ para los u a distancia $\leq k-1$ de s. Después de la iteración k, eso se sigue cumpliendo para esos vértices, ya que Relajar asigna un mínimo entre dos valores, uno de los cuales es $d_{k-1}[u]$, así que $d_k[u] \leq d_{k-1}[u] < \infty$.

Por otro lado, si un vértice u está a distancia k de s, existe un camino $P = s \cdots u$ tal que $|P|$. Si llamamos u' al anteúltimo vértice de P, se puede ver que $d_{k-1}[u] < \infty$, ya que el camino $P_{su'}$ tiene longitud $k-1$ (hipótesis inductiva). Luego, el valor $d_k[u]$ está dado por un mínimo entre dos valores, uno de los cuales es $d_{k-1}[u'] + w(u' \to u) < \infty$ (es una suma de valores finitos), así que $d_k[u] < \infty$. Entonces, queda demostrado el caso k.

En el caso particular de $k = |V| - 1$, cualquier vértice v alcanzable desde s tiene $d[v] < \infty$, ya que si existe un recorrido entre ambos, se pueden recortar los ciclos del mismo para obtener un camino (que tiene una longitud de a lo sumo $|V| - 1$).

Sea $C = v_0 \cdots v_l$ el ciclo de peso negativo alcanzable desde v. Se tiene que:

$$w(C) = \sum_{i=1}^{l} w(v_{i-1} \to v_i) < 0$$

[5] Distancia es la longitud del camino más corto, no el peso del camino mínimo.

Supongamos que el algoritmo no detecta ningún ciclo. Esto implica que $d[u] + w(u \to v) < d[v]$ para todas las aristas del grafo. En particular, $d[v_i] \leq d[v_{i-1}] + w(v_{i+1} \to v_i)\ \forall i = 1...l$. Sumando estas desigualdades:

$$\sum_{i=1}^{l} d[v_i] \leq \sum_{i=1}^{l} d[v_{i-1}] + w(v_{i-1} \to v_i)$$
$$\sum_{i=1}^{l} d[v_i] \leq \sum_{i=1}^{l} d[v_{i-1}] + \sum_{i=1}^{l} w(v_{i-1} \to v_i)$$

Por otro lado, como C es un ciclo, $v_0 = v_l$, y por ende:

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=0}^{k-1} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

Como los vértices son alcanzables desde el origen, sus valores $d[v_i]$ son finitos, así que:

$$\sum_{i=1}^{l} d[v_i] \leq \sum_{i=1}^{l} d[v_{i-1}] + \sum_{i=1}^{l} w(v_{i-1} \to v_i)$$
$$\sum_{i=1}^{l} d[v_i] - \sum_{i=1}^{l} d[v_{i-1}] \leq \sum_{i=1}^{l} w(v_{i-1} \to v_i)$$
$$0 \leq \sum_{i=1}^{l} w(v_{i-1} \to v_i) = w(C)$$

Esto es absurdo, así que el algoritmo tiene que haber detectado el ciclo.

□

4.3.3. Complejidad

La complejidad de Bellman-Ford es fácil de calcular: RELAJAR es una operación de tiempo constante, y corre $|E|$ veces en cada iteración (una por cada arista). Como hay $|V|-1$ iteraciones, la complejidad es $\mathcal{O}(|V||E|)$ (el último ciclo para comprobar la existencia de ciclos negativos es $\mathcal{O}(|E|)$, lo cual no cambia la complejidad).

4.3.4. Mejoras

4.3.5. Aplicación: Sistema de Restricciones de Diferencias

Un problema que se puede resolver a través de Bellman-Ford es el de *sistemas de restricciones de diferencias* (SRD). El enunciado es el siguiente:

Problema:
Encontrar un conjunto de valores $x_1, ..., x_n$, que respeten un sistema de m inecuaciones de la forma:

$$x_i - x_j \leq c_{ij}$$

El problema se puede reducir a camino mínimo el digrafo D, que tiene las siguiente características:

- Para cada $i = 1, ..., n$, hay un vértice $v_i \in V_D$ que se corresponde con la incógnita x_i.
- Para cada inecuación $x_i - x_j \leq c_{ij}$, hay un arco $v_j \to v_i \in E_D$ de peso $w(v_j \to v_i) = c_{ij}$.
- Hay un vértice adicional v_0, que cuenta con arcos $v_0 \to v_i$ hacia todos los vértices v_i, todos de peso $w(v_0 \to v_i) = 0$.

El sistema tiene solución solo cuando el digrafo no tiene ciclos negativos

Teorema. *Un sistema SRD tiene solución cuando el digrafo correspondiente D no tiene ciclos negativos, y en tal caso la solución es $\{x_i = \delta(v_0, v_i) \mid 1 \leq i \leq n\}$.*

Demostración. Primero, veamos que pasa si D tiene algún ciclo negativo $C = v_{i_0} \cdots v_{i_l}$. Supongamos que existe una solución $\{x_1, ..., x_n\}$. En tal caso, se tiene:

$$\sum_{j=1}^{l} w(v_{i_{j-1}} \to v_{i_j}) = \sum_{j=1}^{l} c_{i_j i_{j-1}} < 0$$

Si se suman las inecuaciones correspondientes del sistema, se llega a:

$$\underbrace{\sum_{j=1}^{l} v_{i_j} - v_{i_{j-1}}}_{\text{Suma telescópica}} \leq \sum_{j=1}^{l} c_{i_j i_{j-1}} < 0$$

$$v_{i_l} - v_{i_0} < 0$$

Como C es un ciclo, $v_{i_l} = v_{i_0}$, así que $0 < 0$ (**Absurdo**). Esto implica que no puede existir una solución para el sistema.

Por otro lado, si no existe ningún ciclo de peso negativo, tomemos la asignación de valores $\{x_i = \delta(v_0, v_i) \mid 1 \leq i \leq n\}$. Entonces, para cada arco del grafo $v_j \to v_i$, se cumple la desigualdad triangular:

$$\delta(v_0, v_i) \leq \delta(v_0, v_j) + w(v_j \to v_i)$$

Reemplazando por los valores representados en el sistema,

$$x_i \leq x_j + c_{ij}$$
$$x_i - x_j \leq c_{ij}$$

Es decir, se cumplen todas las desigualdades, así que $\{x_i = \delta(v_0, v_i) \mid 1 \leq i \leq n\}$ es una solución al sistema.

□

Entonces, el sistema de ecuaciones se puede resolver encontrando el camino mínimo de un origen a todos los vértices o detectando algún ciclo de peso negativo alcanzable, que es precisamente lo que logra el algoritmo Bellman-Ford.