Unidad 2: Problemas de camino mínimo

Representación

- Matriz de adyacencia
- Matriz de incidencia
- Listas de vecinos

Recorrido de grafos

- Estructuras de datos
- BFS
- DFS

Camino mínimo

- Dijkstra & Moore
- Bellman & Ford
- Dantzig & Floyd

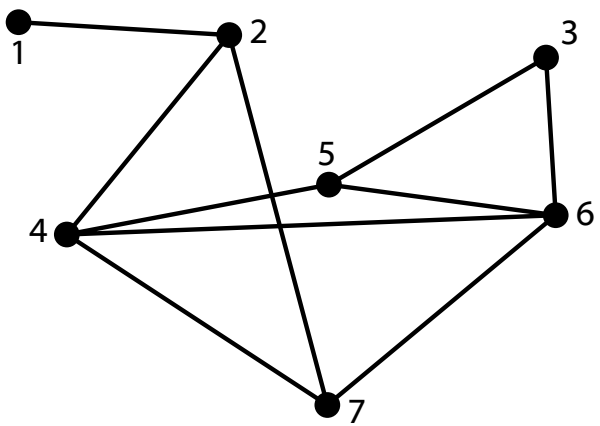
Representación de grafos

- Matriz de adyacencia
- Matriz de incidencia
- Listas de vecinos (para cada vértice, se indica la lista de sus vecinos).
- Cantidad de vértices y lista de aristas (se asumen los vértices numerados de 1 a n).

Matriz de adyacencia

Dado un grafo G cuyos vértices están numerados de 1 a n, definimos la matriz de adyacencia de G como $M \in \{0,1\}^{n \times n}$ donde $M(i,j) = 1$ si los vértices i y j son adyacentes y 0 en otro caso.

Ej:



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

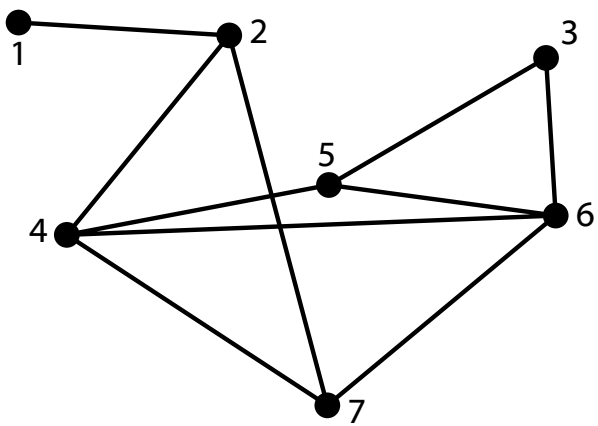
Propiedades de la matriz de adyacencia

- Si A_G es la matriz de adyacencia de G, entonces $A_G^k[i,j]$ es la cantidad de caminos (no necesariamente simples) distintos de longitud k entre i y j.
- En particular, $A_G^2[i,i] = d_G(i)$.
- Y además, un grafo G es bipartito si y sólo si $A_G^k[i,i] = 0$ para todo k impar, o sea, si la diagonal de la matriz $\sum_{j=0}^{n/2} A_G^{2j+1}$ tiene la diagonal nula.
- Un grafo G es conexo si y sólo si la matriz $\sum_{j=1}^{n} A_G^j$ no tiene ceros.

Matriz de incidencia

Numerando las aristas de G de 1 a m, definimos la matriz de incidencia de G como $M \in \{0,1\}^{m \times n}$ donde $M(i,j) = 1$ si el vértice j es uno de los extremos de la arista i y 0 en otro caso.

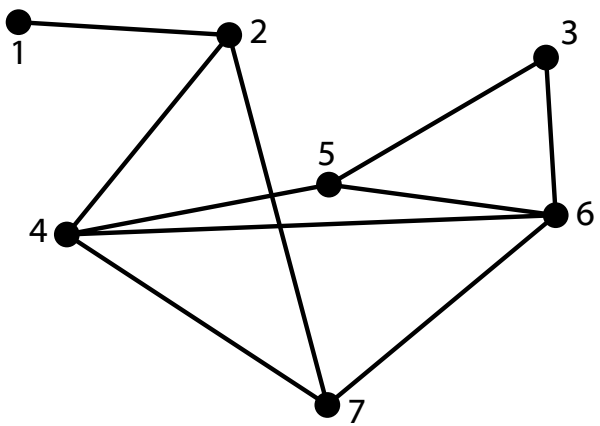
Ej:



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Listas de vecinos

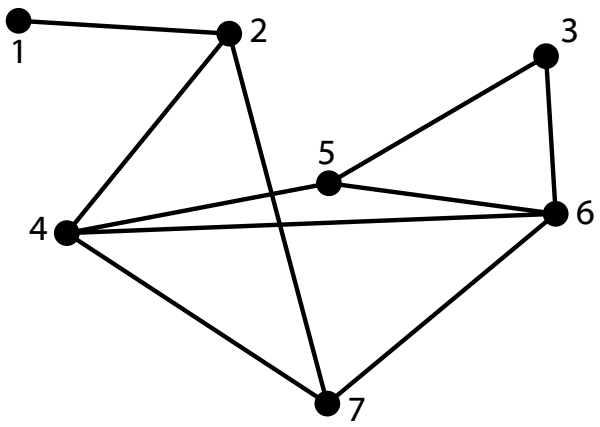
Ej:



$L_1 : 2$
$L_2 : 1 \longrightarrow 4 \longrightarrow 7$
$L_3 : 5 \longrightarrow 6$
$L_4 : 2 \longrightarrow 5 \longrightarrow 6 \longrightarrow 7$
$L_5 : 3 \longrightarrow 4 \longrightarrow 6$
$L_6 : 3 \longrightarrow 4 \longrightarrow 5 \longrightarrow 7$
$L_7 : 2 \longrightarrow 4 \longrightarrow 6$

Lista de aristas

Ej:



Cant. vértices: 7

Aristas: (1,2),(2,4),(2,7),(3,5),
(3,6),(4,5),(4,6),(4,7),(5,6),(6,7)

Es una estructura útil para el algoritmo de Kruskal, por ejemplo. También es la que se usa en general para almacenar un grafo en un archivo de texto.

Algoritmos para recorrido de grafos

Descripción del problema a grandes rasgos:

Tengo un grafo descripto de cierta forma y quiero recorrer sus vértices para:

- encontrar un vértice que cumpla determinada propiedad.
- calcular una propiedad de los vértices, por ejemplo la distancia a un vértice dado.
- calcular una propiedad del grafo, por ejemplo sus componentes conexas.
- obtener un orden de los vértices que cumpla determinada propiedad.

Estructuras de datos

Una pila (*stack*) es una estructura de datos donde el último en entrar es el primero en salir. Las tres operaciones básicas de una pila son:

- apilar (*push*), que coloca un objeto en la parte superior de la pila,
- tope (*top*) que permite ver el objeto superior de la pila y
- desapilar (*pop*), que elimina de la pila el tope.

Estructuras de datos

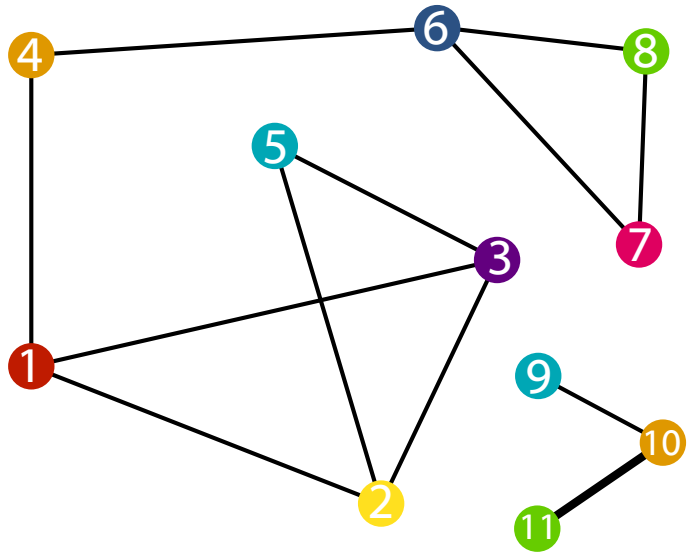
Una cola (*queue*) es una estructura de datos donde el primero en llegar es el primero en salir.
Las tres operaciones básicas de una cola son:

- encolar (*push*), que coloca un objeto al final de la cola,
- primero (*first*) que permite ver el primer objeto de la cola y
- desencolar (*pop*), que elimina de la cola el primer objeto.

Recorrido BFS

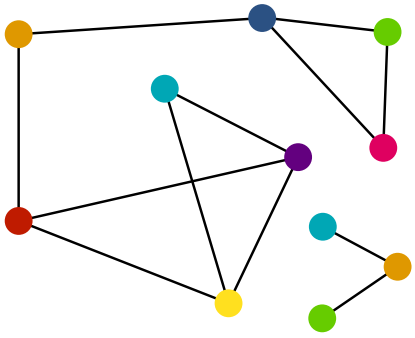
BFS: *Breadth-First Search*, o sea, recorrido a lo ancho. A partir de un vértice, recorre sus vecinos, luego los vecinos de sus vecinos, y así sucesivamente.... si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

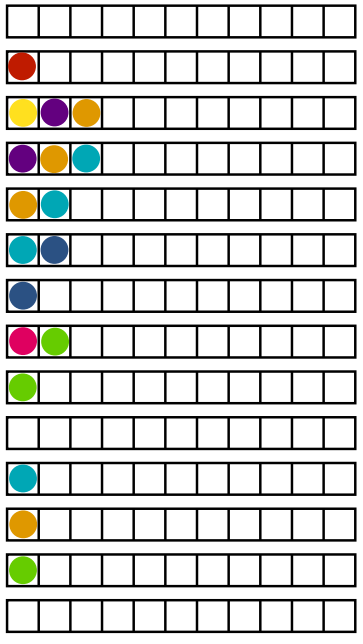
Ejemplo:



Recorrido BFS: pseudocódigo

```
cola C; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    C.vaciar();
    C.encolar(v);
    Mientras C no sea vacía
        w = C.primero();
        ord++;
        orden(w)=ord;
        C.desencolar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                C.encolar(z);
```





Recorrido BFS: propiedades

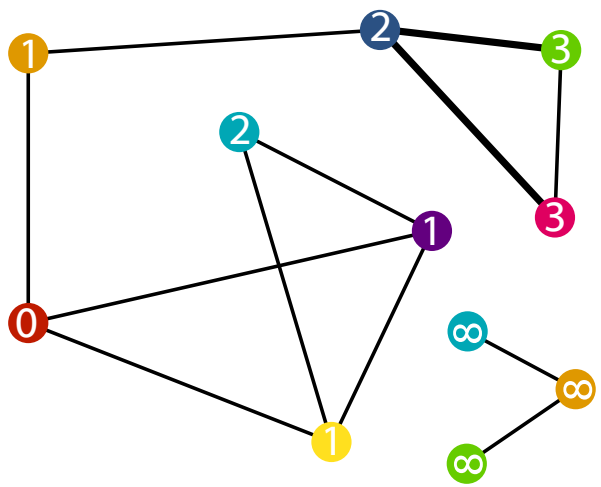
Notemos que cada vez que comenzamos desde un vértice nuevo, es porque los vértices que quedan sin visitar no son alcanzables desde los vértices visitados. Entonces lo que estamos encontrando son las diferentes componentes conexas del grafo.

Si ponemos un contador que se incremente luego de la operación *elegir un vértice no visitado*, lo que obtenemos es la cantidad de componentes conexas del grafo (y con algunas leves modificaciones podemos obtener las componentes en si).

Recorrido BFS: propiedades

Si partimos desde un vértice v, modificando levemente el código podemos calcular para cada vértice su distancia a v. Inicialmente todos los vértices tienen rótulo ∞. Rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún tienen rótulo ∞, los rotulamos con el rótulo de w más 1. (Notemos que los que no sean alcanzados desde v, o sea, no pertenecen a la componente conexa de v, tienen distancia infinita a v.)

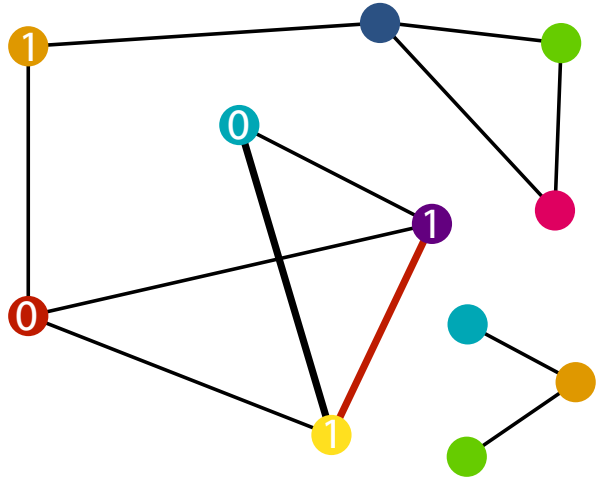
Ejemplo:



Recorrido BFS: reconocimiento de grafos bipartitos

Utilizamos una idea parecida a la del algoritmo anterior en cada componente conexa: rotulamos el vértice v con 0 y luego, al visitar los vecinos de cada vértice w que aún no tienen rótulo, los rotulamos con 1 menos el rótulo de w. De esta manera queda determinada la partición, de ser bipartito el grafo. Queda entonces verificar que ningún par de vértices con el mismo rótulo sean adyacentes (ese chequeo se puede ir haciendo en cada paso).

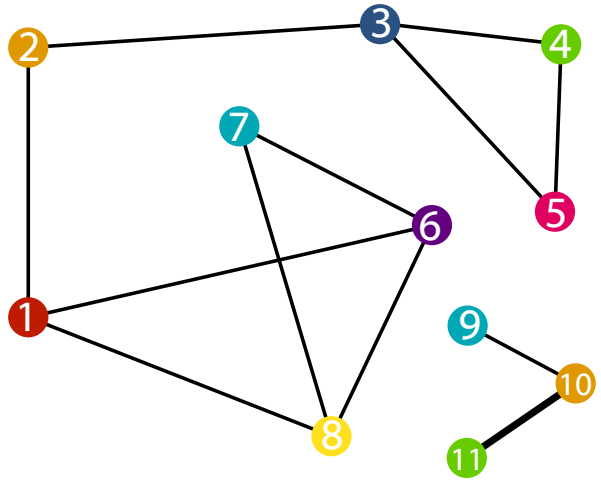
Ejemplo:



Recorrido DFS

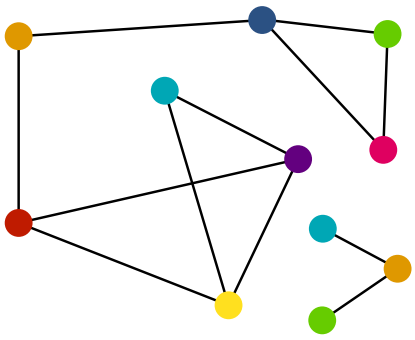
DFS: *Depth-First Search*, o sea, recorrido en profundidad. A partir de un vértice, comienza por alguno de sus vecinos, luego un vecino de éste, y así sucesivamente hasta llegar a un vértice que ya no tiene vecinos sin visitar. Sigue entonces por algún vecino del último vértice visitado que aún tiene vecinos sin visitar.
Igual que en BFS, si al finalizar quedan vértices sin visitar, se repite a partir de un vértice no visitado.

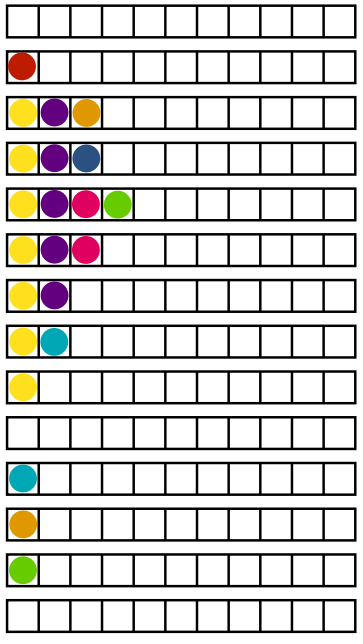
Ejemplo:



Recorrido DFS: pseudocódigo

```
pila P; vértices v, w; ord = 0;
Mientras queden vértices sin visitar
    v = elegir un vértice no visitado;
    marcar v;
    P.vaciar();
    P.apilar(v);
    Mientras P no sea vacía
        w = P.tope();
        ord++;
        orden(w)=ord;
        P.desapilar();
        Para cada vecino z de w
            Si z no está visitado
                marcar z;
                P.apilar(z);
```





Propiedades

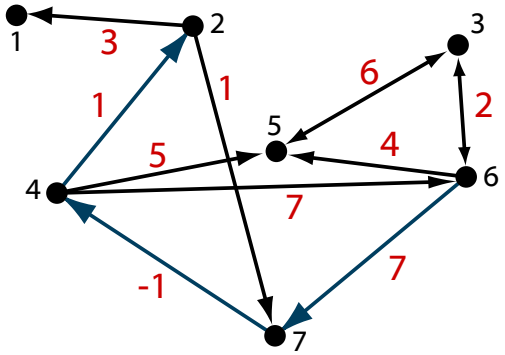
- Al igual que BFS, este algoritmo puede modificarse para calcular las componentes conexas de un grafo y/o su cantidad.
- El grafo en si puede no estar totalmente almacenado, pero tener una descripción implícita que permita generar bajo demanda los vecinos de un vértice. Un ejemplo típico de este caso es cuando el grafo es el árbol de posibilidades de un problema de búsqueda exhaustiva. El recorrido DFS es el que se utiliza habitualmente para hacer *backtracking*, ya que permite encontrar algunas soluciones rápido y usarlas para "podar" (es decir, no recorrer) ciertas ramas del árbol.
- BFS puede ser útil en ese contexto para encontrar el camino más corto hacia la salida de un laberinto, por ejemplo, aún si el laberinto viene descripto de manera implícita.

Estructuras de datos y complejidad

- Para estos algoritmos conviene en general tener el grafo dado por listas de adyacencia.
- De esta manera, el *Para cada vecino z de w* puede ser realizado en orden $O(d(w))$.
- Con lo cual, si el marcar, asignar orden y las operaciones de cola y pila se realizan en orden constante, la complejidad total será de $O(n + m)$, para un grafo de n vértices y m aristas. El n viene del recorrido secuencial que va identificando si quedan vértices no marcados.
- Si en cambio el grafo está dado por matriz de adyacencia, la complejidad será $O(n^2)$, porque buscar los vecinos cuesta $O(n)$.

Problemas de camino mínimo

- Dado un grafo orientado $G = (V, E)$ con longitudes asociadas a sus aristas ($\ell : E \to \mathbb{R}$), la longitud (o peso o costo) de un camino es la suma de las longitudes de sus aristas.



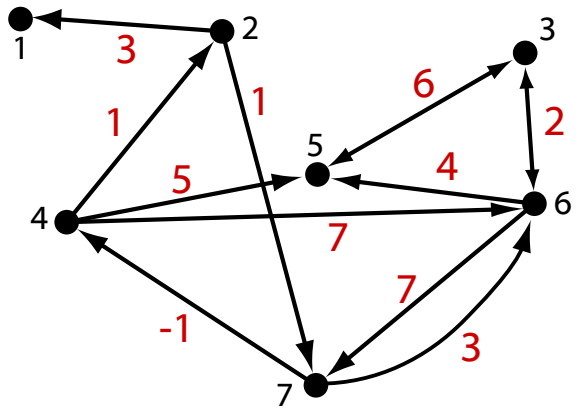
En el digrafo de la figura, la longitud del camino es 7.

- El problema del camino mínimo consiste en encontrar el camino de menor longitud...
  - de un vértice a otro
  - de un vértice a todos los demás
  - entre todos los pares de vértices.

Problemas de camino mínimo

- La distancia de v a w, $d(v, w)$, es la longitud de un camino mínimo entre v y w, $+\infty$ si no existe ningún camino de v a w, y $-\infty$ si existen caminos de v a w pero no uno mínimo.
- Observemos que en un grafo orientado no siempre vale $d(v, w) = d(w, v)$.

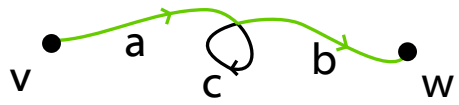
  Ej:



En el digrafo de la figura, $d(7, 4) = -1$ y $d(4, 7) = 2$.

Problemas de camino mínimo

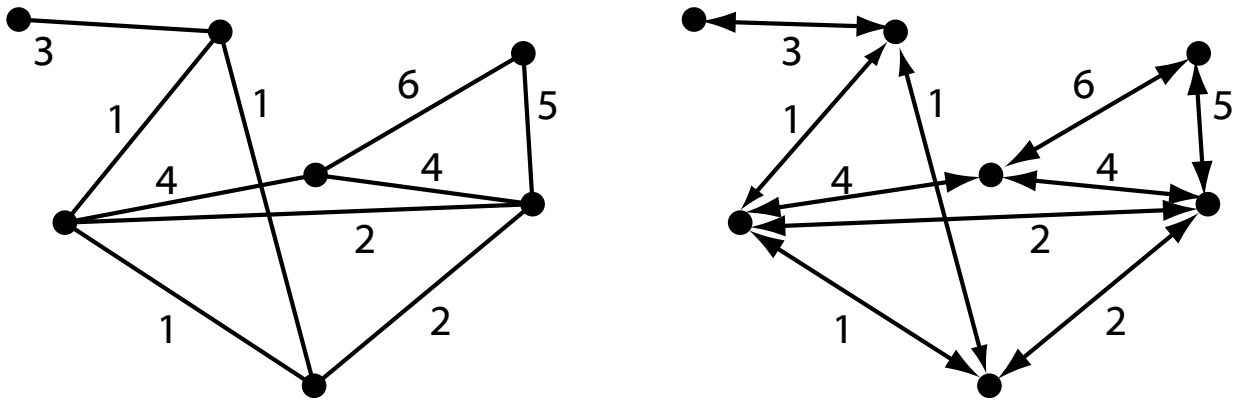
- Si G tiene un ciclo orientado de longitud negativa, no va a existir camino mínimo entre algunos pares de vértices, es decir, va a haber vértices a distancia $-\infty$.



- Si G no tiene ciclos orientados de longitud negativa (aunque pueda tener aristas de longitud negativa), para todo camino P existe P' simple con $\ell(P') \leq \ell(P)$. Como la cantidad de caminos simples en un grafo finito es finita, si existe un camino entre v y w, existe uno mínimo.

Problemas de camino mínimo

- Si G es no orientado, el problema de camino mínimo en G va a ser el mismo que en el digrafo que se obtiene reemplazando cada arista por dos orientadas una para cada lado y con la misma longitud que la original. En ese caso, aristas de longitud negativa implicarían ciclos negativos.



Problemas de camino mínimo

Principio de optimalidad

Todo subcamino de un camino mínimo es un camino mínimo.

Demo: Sea P un camino mínimo entre s y t y sea P_1 un subcamino de P, con extremos v y w. Si existe un camino P_1' en G entre v y w tal que $\ell(P_1') < \ell(P)$ entonces reemplazando P_1 por P_1' en P, obtendríamos un camino entre s y t de longitud menor que la de P, absurdo. □

Arbol de caminos mínimos

Propiedad

Si G no tiene aristas de costo negativo y s es un vértice de G tal que para todo $v \in V(G)$ existe un camino de s a v, entonces existe un árbol orientado T con raíz s tal que para todo $v \in V(G)$, el camino de s a v en T es un camino mínimo de s a v en G.

Demo: Por inducción. Si $V(G) = \{s\}$, vale. Si $|V(G)| \geq 2$, para cada x, sea P_x un camino mínimo de s a x que con cantidad de aristas mínima. Sea v tal que para todo $v' \in V(G)$, $d(s, v') \leq d(s, v)$. Ante empates en la distancia, P_v es el que tiene más aristas. Como no hay aristas de costo negativo y por la forma de elegir v y P_x, para todo v', v no pertenece a $P_{v'}$. Por lo tanto, en $G - v$ existen caminos de s a todos los vértices. Por hipótesis inductiva, existe T' árbol de caminos mínimos de $G - v$. Sea w el anteúltimo vértice en P_v. Entonces $d(s, v) = d(s, w) + \ell(wv)$. Luego, agregando a T' el vértice v y la arista wv obtenemos un árbol de caminos mínimos de G. □

Arbol de caminos mínimos

La propiedad vale igual para grafos con aristas negativas pero sin ciclos negativos, aunque la demostración es distinta.

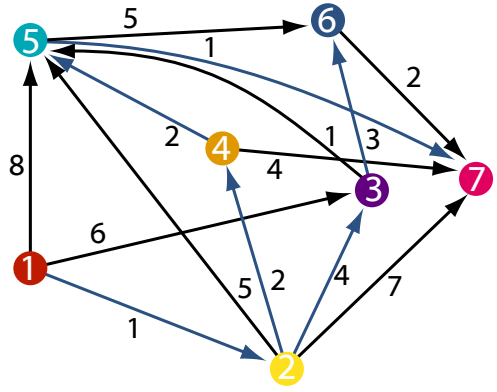
Algoritmo de Dijkstra & Moore

- Sirve para calcular un árbol de caminos mínimos desde un vértice s a todos los vértices alcanzables desde s.
- Funciona cuando el grafo no tiene aristas negativas.

- Input del algoritmo: grafo $G = (V, E)$, $V = \{1, \ldots, n\}$, longitudes $\ell : E \to \mathbb{R}^+$.
- Output del algoritmo: árbol de caminos mínimos desde el vértice 1.

Algoritmo de Dijkstra & Moore

conjunto S;
vectores π y P de longitud n;
$S = \emptyset$;
$\pi(1) = 0$; $\pi(i) = \infty$ para $2 \leq i \leq n$;
$P(1) = 1$; $P(i) = 0$ para $2 \leq i \leq n$;
Mientras $S \neq V$
  Elegir $j \notin S$ tq $\pi(j) = \text{mín}_{i \notin S}\, \pi(i)$
  $S = S \cup \{j\}$;
  Para cada $i \in suc(j)$ tq $i \notin S$
    Si $\pi(i) > \pi(j) + \ell(ji)$
      $\pi(i) = \pi(j) + \ell(ji)$
      $P(i) = j$



$S = \{1, 2, 4, 3, 5, 7, 6\}$

	1	2	3	4	5	6	7
π	0	∞	∞	∞	∞	∞	∞
	0	1	6	∞	8	∞	∞
	0	1	5	3	6	∞	8
	0	1	5	3	5	∞	7
	0	1	5	3	5	8	7
	0	1	5	3	5	8	6
P	1	0	0	0	0	0	0
	1	1	1	0	1	0	0
	1	1	2	2	2	0	2
	1	1	2	2	4	0	4
	1	1	2	2	4	3	4
	1	1	2	2	4	3	5

Algoritmo de Dijkstra & Moore

- En cada iteración agrega un nuevo vértice a S, por lo tanto termina en n iteraciones.
- El costo de cada iteración es $O(n + d(j))$ u $O(\log n + d(j))$ según como se implemente S. Como j recorre V, el costo total del algoritmo es $O(n^2)$ u $O(m + n \log n)$, respectivamente.
- Notación: $\pi^*(i) = d(1, i)$; $\pi_S(i) =$ longitud del camino mínimo de 1 a i que sólo pasa por vértices de S.
- Lema: Al terminar cada iteración, para todo $i \in V$ vale:
  1. $i \in S \Rightarrow \pi(i) = \pi^*(i)$.
  2. $i \notin S \Rightarrow \pi(i) = \pi_S(i)$.
- Teorema: El algoritmo funciona.
  1. Termina.
  2. Cuando termina $S = V$, y por el lema, $\forall i \in V \quad \pi(i) = \pi^*(i)$.

Demostración del lema

Por inducción en la cantidad de iteraciones. Como inicialmente $\pi(1) = 0$ y $\pi(i) = \infty$ para $i \neq 1$, luego de la primera iteración, $S = \{1\}$. Además, por como se actualizó π, vale $\pi(1) = 0 = \pi^*(1)$, $\pi(i) = \ell(1i) = \pi_S(i)$ para los sucesores de 1 y $\pi(i) = \infty = \pi_S(i)$ para el resto de los vértices.

Supongamos por hipótesis inductiva que el lema vale para S y veamos que vale para $S \cup \{j\}$ luego de completar la siguiente iteración. Como π no se modifica para los elementos de S, vale $\pi(i) = \pi^*(i)$ para $i \in S$.

Falta probar entonces

1. $\pi(j) = \pi^*(j)$ y
2. $\pi(i) = \pi_{S \cup \{j\}}(i)$ para $i \notin S \cup \{j\}$.

Demostración del lema

1. $\pi(j) = \pi^*(j)$

Por HI sabíamos que $\pi(j) = \pi_S(j)$. Tomemos un camino P de 1 a j.[1] Sea i el primer vértice fuera de S en P (podría ser j), y sea P_i el subcamino de P que va de 1 a i. Como no hay aristas negativas, $\ell(P) \geq \ell(P_i) \geq \pi_S(i) \geq \pi_S(j)$ por HI y la forma de elegir j. Luego $\pi^*(j) = \pi_S(j) = \pi(j)$.

2. $\pi(i) = \pi_{S\cup\{j\}}(i)$ para $i \notin S \cup \{j\}$

Sea $i \notin S$, $i \neq \{j\}$. Claramente, $\pi_{S\cup\{j\}}(i) \leq \text{mín}\{\pi_S(i), \pi^*(j) + \ell(ji)\}$. Veamos la desigualdad inversa. Sea P un camino de 1 a i que sólo pasa por $S \cup \{j\}$. Sea v el último vértice antes de i en P. Si $v = j$, entonces $\ell(P) \geq \pi^*(j) + \ell(ji)$ $(*)$. Si $v \in S$, sea P' el camino que se obtiene de reemplazar en P el subcamino de 1 a v por un camino de 1 a v de longitud $\pi^*(v)$ que sólo pase por S, que por HI sabemos que existe. Entonces $\ell(P) \geq \ell(P') \geq \pi_S(i)$ $(**)$.
Por $(*)$ y $(**)$, vale $\pi_{S\cup\{j\}}(i) \geq \text{mín}\{\pi_S(i), \pi^*(j) + \ell(ji)\}$. □

[1] Podemos considerar que existen todas las aristas pero algunas pesan ∞.

Teorema

Sea $G = (V, E)$, $s \in V$, sin ciclos negativos. Para cada $j \in V$, sea d_j la longitud de un camino de s a j. Los números d_j representan las distancias de s a j si y sólo si $d_s = 0$ y $d_j \leq d_i + \ell(ij) \quad \forall ij \in E$.

Demo: $\Rightarrow$) Es claro que $d_s = 0$. Supongamos que $d_j > d_i + \ell(ij)$ para algún $ij \in E$. Como d_i es la longitud de un camino P entre s e i, entonces $\ell(P + ij) = d_i + \ell(ij) < d_j$, por lo tanto d_j no es la distancia entre s y j.
$\Leftarrow$) Sea $P = s = i_1 i_2 \ldots i_k = j$ un camino de s a j.

$$\begin{array}{rcl}
d_j = d_{i_k} & \leq & d_{i_{k-1}} + \ell(i_{k-1} i_k) \\
d_{i_{k-1}} & \leq & d_{i_{k-2}} + \ell(i_{k-2} i_{k-1}) \\
& \vdots & \\
d_{i_2} & \leq & d_{i_1} + \ell(i_1 i_2) = \ell(i_1 i_2) \\
\hline
d_j = d_{i_k} & \leq & \sum_{ij \in P} \ell(ij)
\end{array}$$

Entonces d_j es cota inferior de la longitud de cualquier camino de s a j. $\square$

Algoritmo genérico de corrección de etiquetas

vectores d y P de longitud n;

$d(s) = 0$; $P(s) = 0$;

$d(j) = \infty$ para $1 \leq j \leq n$, $j \neq s$;

Mientras exista $ij \in E$ con $d(j) > d(i) + \ell(ij)$

$d(j) = d(i) + \ell(ij)$;

$P(j) = i$;

Propiedad

Todo ij en el grafo de predecesores tiene $\ell(ij) + d_i - d_j \leq 0$.

Demo: Cuando se agrega una arista ij, $d_j = d_i + \ell(ij) \Rightarrow$ $\ell(ij) + d_i - d_j = 0$. En las siguientes iteraciones,

- si d_i disminuye, entonces $\ell(ij) + d_i - d_j \leq 0$.
- si d_j disminuye, se elimina la arista ij del grafo.

$\square$

Propiedad

Si no hay ciclos negativos, el grafo de predecesores tiene un único camino desde s hacia todo otro vértice k alcanzable desde s, y este camino tiene longitud a lo sumo d_k.

Demo: Por inducción en la cantidad de iteraciones del algoritmo, hay un único camino desde s hacia todo otro vértice k alcanzable desde s (se usa que no hay ciclos negativos para probar que a s nunca se le asigna un padre distinto de 0). Sea P el camino de s a k. $0 \geq \sum_{ij \in P} (\ell(ij) + d_i - d_j)$ $= \sum_{ij \in P} \ell(ij) + d_s - d_k = \sum_{ij \in P} \ell(ij) - d_k \Rightarrow \sum_{ij \in P} \leq d_k$. □

Corolario

Si el algoritmo termina, el grafo de predecesores es un árbol generador de caminos mínimos (y se puede probar que si no hay ciclos negativos, termina).

Algoritmo de Bellman & Ford

vectores d^k, $k = 0, \ldots, n$, de longitud n;

$d^0(1) = 0$; $d^0(i) = \infty \ \forall i \neq 1$; $k = 1$;

$T_1 =$ falso; $T_2 =$ falso;

Mientras $\neg(T_1 \vee T_2)$

  $d^k(1) = 0$;

  Para i desde 2 hasta n

    $d^k(i) = \text{mín}\{d^{k-1}(i), \text{mín}_{ji \in E}(d^{k-1}(j) + \ell(ji))\}$;

  Si $d^k(i) = d^{k-1}(i)$ para todo i

    $T_1 =$ verdadero;

  Si no

    Si $k = n$

      $T_2 =$ verdadero;

    Si no

      $k = k + 1$;

Algoritmo de Bellman & Ford

- El invariante del algoritmo es $d^k(i) =$ camino mínimo de 1 a i que usa a lo sumo k aristas.
- Si no hay ciclos negativos, hay camino mínimo con a lo sumo $n - 1$ aristas.
- Entonces, con este invariante, el algoritmo funciona.
- La complejidad es $O(nm)$ y se puede hacer fácil usando 2 arreglos y con cuidado, usando 1 arreglo.

Algoritmo de Dantzig

Los algoritmos de Dantzig y Floyd calculan los caminos mínimos entre todos los pares de vértices. Sea $L[i,j]$ la distancia de i a j, $\ell(i,j)$ la longitud de la arista ij o ∞ si no existe.

- El algoritmo de Dantzig en cada paso k considera el subgrafo inducido por los vértices $1,\ldots,k$.
- Entonces L^k es la matriz de distancias en ese subgrafo:
  - $L^{k+1}[i,k+1] = \min_{1\le j\le k} L^k[i,j] + \ell(j,k+1)$.
  - $L^{k+1}[k+1,i] = \min_{1\le j\le k} L^k[j,i] + \ell(k+1,j)$.
  - $L^{k+1}[k+1,k+1] = \min\{0, \min_{1\le i\le k} L^{k+1}[k+1,i] + L^{k+1}[i,k+1]\}$.
  - $L^{k+1}[i,j] = \min\{L^k[i,j], L^{k+1}[i,k+1] + L^{k+1}[k+1,j]\}$.
- L^n es la matriz buscada. Para ver si hay ciclos negativos, hay que ver si hay números negativos en la diagonal.
- Es $O(n^3)$. Aplicar Dijkstra n veces puede ser más eficiente si m es chico, no hay aristas negativas y se implementa en forma óptima.

Algoritmo de Floyd

- El algoritmo de Floyd en cada paso k calcula el camino mínimo de i a j con vértices intermedios en el conjunto $\{1, \ldots, k\}$.
- L^k es la matriz del paso k:
  - $L^0[i,i] = 0$ y para $i \neq j$, $L^0[i,j] = \ell(ij)$ si $ij \in E$ y $L^0[i,j] = \infty$ si $ij \notin E$.
  - $L^{k+1}[i,j] = \text{mín}\{L^k[i,j], L^k[i,k+1] + L^k[k+1,j]\}$.
- L^n es la matriz buscada. Para ver si hay ciclos negativos, hay que ver si hay números negativos en la diagonal.
- En ambos algoritmos se puede trabajar corrigiendo sobre la misma matriz (los superíndices son a fines de entender el algoritmo). La complejidad temporal es $O(n^3)$.