Notas de la clase 7 – camino mínimo todos a todos

Francisco Soulignac

29 de marzo de 2019

Aclaración: este es un punteo de la clase para la materia AED3. Se distribuye como ayuda memoria de lo visto en clase y, en cierto sentido, es un reemplazo de las diapositivas que se distribuyen en otros cuatrimestres. Sin embargo, no son material de estudio y no suplanta ni las clases ni los libros. Peor aún, puede contener "herrorez" y podría faltar algún tema o discusión importante que haya surgido en clase. Finalmente, estas notas fueron escritas en un corto período de tiempo. En resumen: **estas notas no son para estudiar sino para saber qué hay que estudiar**.

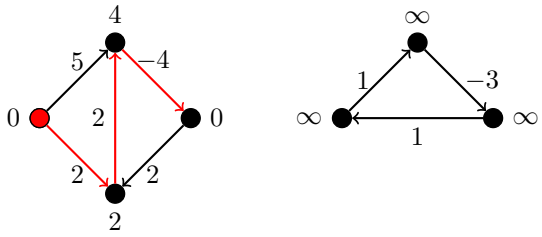
Tiempo total: 150 minutos

1. Camino mínimo todos a todos (20 mins)

- Consideremos un (di)grafo G y una función de costos c. Recordemos que:
  - ▷ Un vértice v es un origen cuando v alcanza a todos los vértices de G.
  - ▷ $\delta(v, w)$ denota el mínimo de los pesos de los caminos **simples** de v a w.
  - ▷ Un árbol de caminos mínimos desde v (v-ACM) de (G, c) es un árbol T tal que $\delta_G(v, w) = \delta_T(w, v)$ para todo $w \in V(G)$.
  - ▷ Un diccionario de pesos mínimos desde v (v-DPM) es un diccionario D tal que $D[w] = \delta_G(v, w)$ para todo $w \in V(G)$.
  - ▷ Si G no tiene ciclos de peso negativo, entonces existe un v-ACM y un v-DPM desde v, para todo origen v.
- Estas nociones se pueden extender trivialmente a vértices que no son origen.
- Si V es el conjunto de vértices alcanzables desde v, entonces:
  - ▷ El v-ACM de G es en rigor un bosque, donde w es aislado para $w \notin V$.
  - ▷ $D[w] = \infty$ para todo $w \notin V$ en el v-DPM de (G, c).
  - ▷ Si $G[V]$ no tiene ciclos de peso negativo, entonces existe un v-ACM y un v-DPM desde v.

Bosque de caminos mínimos

En la figura se muestra el bosque de caminos mínimos desde el vértice rojo y el diccionario de pesos mínimos corresponden a las etiquetas de los vértices.

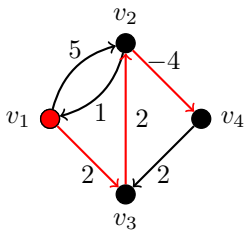


- El algoritmo de Bellman-Ford computa un v-ACM y un v-DPM de G en $O(nm)$ tiempo para cualquier $v \in V(G)$.
- Cuando $c \geq 0$, podemos aplicar Dijkstra para bajar el tiempo a $O(m + n \log n)$.
- Contar con un v-ACM permite construir un camino mínimo de v a cualquier otro vértice $w \in V$ en $O(k)$ tiempo, siendo k la longitud del camino mínimo de v a w.
- Contar con el v-DPM permite conocer la distancia de v a w en $O(1)$ tiempo.
- En ciertas aplicaciones (e.g., mapas) necesitamos responder queries para distintos vértices iniciales.
- Calcular un v-ACM y un v-DPM por cada vértice v resulta altamente costoso.
- En cambio, si almacenamos un v-ACM y v-DPM para cada vértice v de antemano, entonces podemos saber la distancia de v a w en $O(1)$ tiempo, y obtener un camino mínimo de v a w en $O(k)$ tiempo, siendo k la cantidad de aristas del camino, para todo $w \in V$.
- Este método tiene como obvia desventaja su costo espacial: almacenar $O(n)$ ACMs y DPMs consume $O(n^2)$ espacio.
- Por lo tanto, sólo podemos aplicarlo en instancias de tamaño moderado.[1]
- Para formalizar esta idea, decimos que:
  - ▷ un diccionario P con claves $V(G)$ es una *matriz de caminos mínimos* (MCM) cuando $P[v]$ es un v-ACM de G para todo $v \in V(G)$.
  - ▷ Informalmente, $P[v][w]$ denota el predecesor de w en un camino mínimo de v a w.
  - ▷ un diccionario D con claves $V(G)$ es una *matriz de pesos mínimos* (MPM) cuando $D[v]$ es un v-DPM para todo $v \in V(G)$.
  - ▷ Informalmente, $D[v][w]$ denota el peso de los caminos mínimos de v a w.
- La razón para llamarlos "matrices" es obvia: desde el punto de vista matemático satisfacen la definición de matriz; desde el punto de vista computacional, si $V(G) = \{1, \ldots, n\}$, entonces se pueden representar como vectores de vectores.

Matrices de caminos y pesos mínimos

De izquierda a derecha: un digrafo, su MCM y su MPM. En rojo se muestra el v-ACM del vértice rojo y las correspondientes filas de las matrices.

[1] También se puede dividir un grafo grande en clusters, aunque no lo hacemos en la materia.



	v_1	v_2	v_3	v_4
v_1	1	3	1	2
v_2	2	2	4	2
v_3	2	2	3	2
v_4	2	3	4	4

	v_1	v_2	v_3	v_4
v_1	0	4	2	0
v_2	1	0	-2	-4
v_3	3	2	0	-2
v_4	5	4	2	0

- Problema de camino mínimo todos a todos (problema APSP[2]):

  Input: un (di)grafo G y una función de costos c

  Output: un ciclo de peso negativo de G o un MPM y un MCM de (G, c).

- Para resolver este problema, alcanza con aplicar n veces Bellman-Ford con un costo temporal de $O(n^2 m)$.

- Si $c \geq 0$, podemos aplicar n veces Dijkstra, implementado sobre fibonacci heaps, para bajar el costo temporal a $O(nm + n^2 \log n)$.

- En esta clase vemos distintos algoritmos para mejorar la complejidad en el caso general:

  Floyd-Warshall computa los caminos en $O(n^3)$ tiempo y es fácil de implementar.

  Dantzig computa los caminos en $O(n^3)$ tiempo y es similar a Floyd-Warshall.

  Johnson combina Bellman-Ford con Dijkstra para reducir la complejidad a $O(nm + n^2 \log n) = O(n^3)$, igualando el costo asintótico para grafos densos y mejorando el costo para grafos con $o(n^2)$ aristas. Como contrapartida, la implementación es más compleja y la constante escondida en la notación O es más alta.

- En todos los casos, podemos detectar previamente si G tiene un ciclo de peso negativo (y computar uno eventualmente) en tiempo $O(nm)$:

  ▷ Primero creamos (H, c') desde (G, c) agregando una fuente s con una arista sv (de cualquier costo) para todo $v \in V(G)$.

  ▷ Como s no pertenece a ningún ciclo, obtenemos que H tiene un ciclo de peso negativo si y sólo si G tiene un ciclo de peso negativo.

  ▷ Luego, podemos detectar si G tiene ciclos negativos ejecutando Bellman-Ford desde s en (H, c').

  ▷ El costo temporal es $O(n + m)$ para crear H y $O(nm)$ para ejecutar Bellman-Ford.

- Igualmente, discutimos cómo detectar la presencia de ciclos negativos en los nuevos algoritmos, dado que es más simple que tener que implementar Bellman-Ford sólo para este propósito.

2. Algoritmo de Floyd-Warshall (60 mins)

- El algoritmo que vemos en esta sección, llamado Floyd-Warshall, fue descubierto varias veces en forma independiente en distintos contextos. Su nombre proviene de un algoritmo de Warshall [4] para (di)grafos con peso unitario (matrices booleanas), que fue implementado por Floyd [2] para (di)grafos (y matrices) con pesos arbitrarios.

- Como hicimos para Bellman-Ford, vamos a derivar el algoritmo de Floyd-Warshall como un ejercicio de programación dinámica.

[2]Por sus siglas en inglés: all pairs shortest path problem

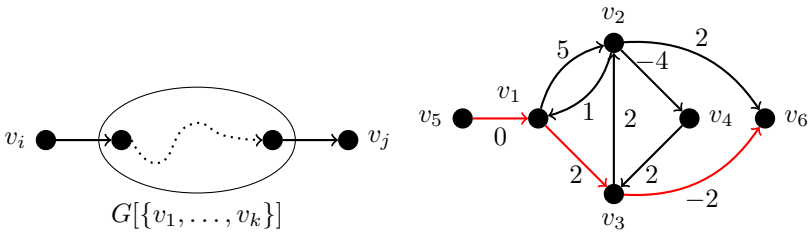
- La clase anterior diseñamos Bellman-Ford usando dos nuevas nociones de distancia:
  - ▷ $\varepsilon^i(v, w)$: mínimo peso de los caminos **simples** de v a w que usan i aristas, y
  - ▷ $\hat{\varepsilon}^i(v, w)$: mínimo peso de los caminos de v a w que usan a lo sumo i aristas.
- Para derivar Floyd-Warshall seguimos una estrategia similar: definimos dos nociones de distancia φ y $\hat{\varphi}$.
- Esta vez, sin embargo, la semántica de $\hat{\varphi}$ es difícil de describir coloquialmente.

2.1. Cálculo de matriz de pesos

- Sea $v_1, \ldots, v_n$ un orden lineal cualquiera de los vértices de un (di)grafo G.
- Para $0 \leq k \leq n$, digamos que un camino P de G es k-interno cuando sus vértices internos pertenecen a $\{v_1, \ldots, v_k\}$.
- Denotamos con $\varphi^k(v_i, v_j)$ el mínimo peso de entre los caminos **simples** k-internos de v_i a v_j.
- Por simplicidad, decimos que un camino k-interno P de v_i a v_j es *mínimo* cuando $c_+(P) = \varphi^k(v_i, v_j)$.

Caminos k-internos

Esquema de camino k-interno y ejemplo de camino 4-interno mínimo de v_5 a v_6: notar que los extremos del camino pueden o no pertenecer a $\{v_1, \ldots, v_k\}$.



- Claramente, todo camino simple es n-interno, con lo cual $\varphi^n = \delta$.
- En consecuencia, si pudiéramos computar eficientemente φ, también seríamos capaces de computar eficientemente δ.
- Como no se conocen algoritmo polinomiales para δ, tampoco se conocen para φ.
- Igualmente, nosotros queremos computar φ para el caso en que G no tiene ciclos de peso negativo.
- En este caso, nos podemos aprovechar de la siguiente generalización de un resultado ya visto.

Lema 1. *Sea $v_1, \ldots, v_n$ un ordenamiento de los vértices de un (di)grafo G que no tiene ciclos de peso negativo para una función de pesos c. Si P es un camino k-interno mínimo, entonces cada subcamino de P es k-interno mínimo.*

Demostración. Ejercicio. □

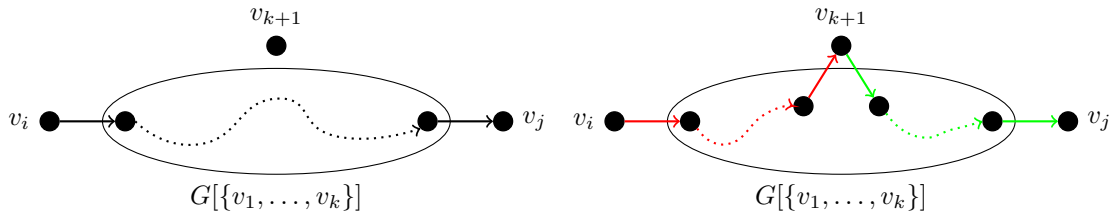
- Supongamos que G no tiene ciclos de peso negativo.
- Claramente, hay sólo dos posibilidades para un camino $(k+1)$-interno mínimo P que va de v_i a v_j.

Posibilidad 1: P no pasa por v_{k+1}. En este caso, P es también k-interno y, por lo tanto, $\varphi^{k+1}(v_i, v_j) = c_+(P) = \varphi^k(v_i, v_j)$

Posibilidad 2: P pasa por v_{k+1}. En este caso, P se puede descomponer en dos caminos k-internos P_1 y P_2 que van de v_i a v_{k+1} y de v_{k+1} a v_j, respectivamente. Como G no tiene ciclos de peso negativo, estos caminos son mínimos Lema 1. Por lo tanto, $\varphi^{k+1}(v_i, v_j) = c_+(P) = c_+(P_1) + c_+(P_2) = \varphi^k(v_i, v_{k+1}) + \varphi^k(v_{k+1}, v_j)$

Descripción gráfica de Floyd-Warshall

A la izquierda el caso en que el camino $(k+1)$-interno P no pasa por v_{k+1}. A la derecha el caso en que sí pasa y se descompone en un camino rojo y otro verde. Todos los caminos resultantes son k-internos.



- Por otra parte, el único camino 0-interno de v_i a v_j es $P = v_i, v_j$, que existe si y sólo si $v_i v_j \in E(G)$.

- Si consideramos que $c(vv) = 0$ y $c(vw) = \infty$ para $v \neq w$, entonces $\hat{\varphi}$ se calcula recursivamente de la siguiente forma.

$$\hat{\varphi}^k(v_i, v_j) = \begin{cases} c(v_i v_j) & \text{si } k = 0 \\ \min\{\hat{\varphi}^{k-1}(v_i, v_j), \hat{\varphi}^{k-1}(v_i, v_k) + \hat{\varphi}^{k-1}(v_k, v_j)\} & \text{caso contrario} \end{cases} \tag{1}$$

- De acuerdo a la discusión anterior, podemos observar que $\varphi = \hat{\varphi}$ cuando G no tiene ciclos de peso negativo.

- Remarcamos que $\hat{\varphi}^k(v_i, v_j)$ está bien definido todo $k \geq 0$, incluso cuando G tiene ciclos de peso negativo. Sin embargo, en tal caso, $\hat{\varphi} \neq \varphi$ y su significado es irrelevante para nuestros propósitos.

- Existan o no ciclos de peso negativo, a nosotros nos interesa computar $\hat{\varphi}^n(v_i, v_j)$.

- Observemos que, por definición, $\hat{\varphi}$ tiene la propiedad de superposición de subproblemas.

- Más aún, como la cantidad de instancias posibles es $\Theta(n^3)$ y cada instancia se puede computar en $O(1)$ tiempo, tenemos un algoritmo de programación dinámica que computa $\hat{\varphi}^n$ en tiempo $\Theta(n^3)$.

- La implementación más sencilla y eficiente en espacio es en forma bottom-up.

- Para cada $k \geq 0$, sea D^k un diccionario tal que $D^k[i][j] = \hat{\varphi}^k(v_i, v_j)$.

- Notemos que $D^0[i][j] = \hat{\varphi}^0(v_i, v_j) = c(v_i v_j)$.

- En cambio, si $k > 0$, entonces

$$\begin{aligned} D^k[i][j] &= \min\{\hat{\varphi}^{k-1}(v_i, v_j), \hat{\varphi}^{k-1}(v_i, v_k) + \hat{\varphi}^{k-1}(v_k, v_j)\} \\ &= \min\{D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j]\}. \end{aligned}$$

- De esta definición observamos que D^{k-j} no es requerido para calcular D^k para $j > 1$.

- Más aún, notemos que $\hat{\varphi}$ es no creciente por (1), con lo cual:
  - ▷ $\hat{\varphi}^n(v_i, v_i) \leq \hat{\varphi}^k(v_i, v_i) \leq 0 = \varphi^n(v_i, v_i) = \delta(v_i, v_j)$ para todo $0 \leq i, k \leq n$,
  - ▷ Si $\hat{\varphi}^k(v_i, v_i) < 0$, entonces $\hat{\varphi}^n \neq \varphi^n = \delta$ lo que implica que G tiene un ciclo de peso negativo.
- Luego, en el caso en que G no tiene ciclos de peso negativo, tenemos que $\hat{\varphi}^{k-1}(v_k, v_k) = 0$ y, por lo tanto:
  - ▷ $\hat{\varphi}^k(v_i, v_k) = \min\{\hat{\varphi}^{k-1}(v_i, v_k), \hat{\varphi}^{k-1}(v_i, v_k) + \hat{\varphi}^{k-1}(v_k, v_k)\} = \hat{\varphi}^{k-1}(v_i, v_k)$, y
  - ▷ $\hat{\varphi}^k(v_k, v_j) = \min\{\hat{\varphi}^{k-1}(v_k, v_j), \hat{\varphi}^{k-1}(v_k, v_j) + \hat{\varphi}^{k-1}(v_k, v_k)\} = \hat{\varphi}^{k-1}(v_k, v_j)$.
  - ▷ Reescribiendo 1, $\hat{\varphi}^k(v_i, v_j) = \min\{\hat{\varphi}^{k-1}(v_i, v_j), \hat{\varphi}^k(v_i, v_k) + \hat{\varphi}^k(v_k, v_j)\}$.
  - ▷ O, en matriz, $D^k[i][j] = \min\{D^{k-1}[i][j], D^k[i][k] + D^k[k][j]\}$.
- Esto tiene una explicación obvia en lenguaje coloquial: el camino k-interno mínimo de v_i a v_k (resp. de v_k a v_j) es $(k-1)$-interno porque no puede contener a v_k en el interior.
- Algoritmicamente, es indiferente si usamos D^k o D^{k-1} cuando queremos actualizar $D^{k-1}[i][j]$.
- En otras palabras, podemos utilizar una sola matriz que vamos pisando para calcular los resultados.
- Por simplicidad, mostramos el algoritmo para calcular $\hat{\varphi}$ directamente en C++, ya que son sólo tres for anidados.

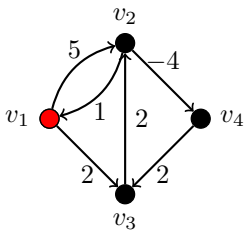
Algoritmo de Floyd-Warshall

```cpp
using edge   = tuple<int, int, int>;        //v, w, c(vw)
using graph  = vector<edge>;                //lista de aristas
using matrix = vector<vector<int>>;
const int infty = std::numeric_limits<int>::max() / 2 - 1; //valor apropiado

//Precondición: G no tiene ciclos de peso negativo, n = |V(G)|, hay arista vv con c(vv) = 0
matrix FloydWarshall(const graph& G, int n) {
  matrix D(n, vector<int>(n, infty));
  for(auto e: G) D[v(e)][w(e)] = c(e);      //v = get<0>, w = get<1>, c = get<2>
  for(int k = 0; k < n; ++k) for(int i = 0; i < n; ++i) for(int j = 0; j < n; ++j)
    D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
  return D;
}
```

Funcionamiento de Floyd-Warshall

En cada paso se marcan en rojo los valores cambiados.



$$D_0 = \left(\begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline v_1 & 0 & 5 & 2 & \infty \\ v_2 & 1 & 0 & \infty & -4 \\ v_3 & \infty & 2 & 0 & \infty \\ v_4 & \infty & \infty & 2 & 0 \end{array}\right) \quad D_1 = \left(\begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline v_1 & 0 & 5 & 2 & \infty \\ v_2 & 1 & 0 & 3 & -4 \\ v_3 & \infty & 2 & 0 & \infty \\ v_4 & \infty & \infty & 2 & 0 \end{array}\right)$$

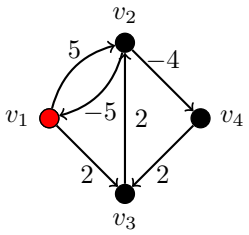
$$D_2 = \left(\begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline v_1 & 0 & 5 & 2 & 1 \\ v_2 & 1 & 0 & 3 & -4 \\ v_3 & 3 & 2 & 0 & -2 \\ v_4 & \infty & \infty & 2 & 0 \end{array}\right) \quad D_3 = \left(\begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline v_1 & 0 & 4 & 2 & 0 \\ v_2 & 1 & 0 & 3 & -4 \\ v_3 & 3 & 2 & 0 & -2 \\ v_4 & 5 & 4 & 2 & 0 \end{array}\right) \quad D_4 = \left(\begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline v_1 & 0 & 4 & 2 & 0 \\ v_2 & 1 & 0 & -2 & -4 \\ v_3 & 3 & 2 & 0 & -2 \\ v_4 & 5 & 4 & 2 & 0 \end{array}\right)$$

2.2. Detección de ciclos de peso negativo

- Recordemos que podemos invocar Bellman-Ford para detectar si G tiene un ciclo de peso negativo.
- Sin embargo, esto implica un gasto extra en programación si pretendemos igualmente ejecutar Floyd-Warshall.
- En su lugar podemos detectar si G tiene un ciclo de peso negativo directamente en Floyd-Warshall.
- En la sección previa vimos que si $\hat{\varphi}^k(v_i, v_j) < 0$, entonces G tiene un ciclo de peso negativo.
- Afortunadamente, la vuelta también vale.

Funcionamiento de Floyd-Warshall cuando hay ciclos negativos

Columna v_3, que es la única que cambia en la iteración 1: notar que $D_2[3][3] < 0$.



$$D_0 = \left(\begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline v_1 & 0 & 5 & 2 & \infty \\ v_2 & 1 & 0 & \infty & -4 \\ v_3 & \infty & 2 & 0 & \infty \\ v_4 & \infty & \infty & 2 & 0 \end{array}\right) \quad D_1 = \left(\begin{array}{c|c} & v_3 \\ \hline v_1 & 2 \\ v_2 & -2 \\ v_3 & 0 \\ v_4 & 2 \end{array}\right) \quad D_2 = \left(\begin{array}{c|c} & v_3 \\ \hline v_1 & 2 \\ v_2 & -2 \\ v_3 & -1 \\ v_4 & 2 \end{array}\right)$$

Lema 2. *Sea $v_1, \ldots, v_n$ un ordenamiento de los vértices de un (di)grafo G y c una función de costos. Para todo $1 \leq i, j, k \leq n$ ocurre que $\hat{\varphi}^k(v_i, v_j) \leq \varphi^k(v_i, v_j)$.*

Demostración. Por inducción en k, el caso base siendo trivial porque $\varphi^1 = \hat{\varphi}^1$. Para el paso inductivo, consideremos cualquier camino simple $(k+1)$-interno P con peso $\varphi^{k+1}(v_i, v_j)$. Como vimos antes, o bien P es k-interno o bien $P = P_1 + P_2$ para caminos k-internos P_1 y P_2 de v_i a v_{k+1} y de v_{k+1} a v_j, respectivamente. En el primer caso, $\varphi^{k+1}(v_i, v_j) = \varphi^k(v_i, v_j) \geq \hat{\varphi}^k(v_i, v_j) \geq \hat{\varphi}^{k+1}(v_i, v_j)$ por (1) e hipótesis inductiva. En el segundo caso, $\varphi^{k+1}(v_i, v_j) = c_+(P_1) + c_+(P_2) \geq \varphi^k(v_i, v_{k+1}) + \varphi^k(v_{k+1}, v_j)$. Luego, por hipótesis inductiva, $\varphi^{k+1}(v_i, v_j) \geq \hat{\varphi}^k(v_i, v_{k+1}) + \hat{\varphi}^k(v_{k+1}, v_j) = \hat{\varphi}^{k+1}(v_i, v_j)$. □

Corolario 1. *Sea $v_1, \ldots, v_n$ un ordenamiento de los vértices de un (di)grafo G y c una función de costos. Entonces G tiene un ciclo de peso negativo si y sólo si $\hat{\varphi}^k(v_i, v_i) < 0$ para algún par $1 \leq i, k \leq n$.*

Demostración. Si G no tiene ciclos de peso negativo, entonces $\hat{\varphi}^n = \varphi^n = \delta$. En consecuencia, $0 = \hat{\varphi}^0(v_i, v_i) \geq \hat{\varphi}^k(v_i, v_i) \geq \hat{\varphi}^n(v_i, v_i) = \delta(v_i, v_i) = 0$.

Supongamos ahora que G contiene un ciclo $C = v_{p(1)}, \ldots, v_{p(q)}, v_{p(1)}$ de peso negativo. Sin perdida de generalidad, supongamos que $p(q) > p(j)$ para todo $1 \leq j < q$. Luego, $P_1 = v_{p(1)}, \ldots, v_{p(q)}$ y $P_2 = v_{p(q)}, v_{p(1)}$

son caminos simples $(p(q)-1)$-internos. Luego:

$$\begin{aligned}
\hat{\varphi}^{p(q)}(v_{p(1)}, v_{p(1)}) &\leq \hat{\varphi}^{p(q)-1}(v_{p(1)}, v_k) + \hat{\varphi}^{p(q)-1}(v_k, v_{p(1)}) && \text{(por (1))} \\
&\leq \varphi^{p(q)-1}(v_{p(1)}, v_k) + \varphi^{p(q)-1}(v_k, v_{p(1)}) && \text{(por Lema 2)} \\
&\leq c_+(P_1) + c(P_2) && \text{(por ser caminos } (p(q)-1)\text{-internos)} \\
&= c_+(C) < 0.
\end{aligned}$$

□

2.3. Matriz de caminos mínimos

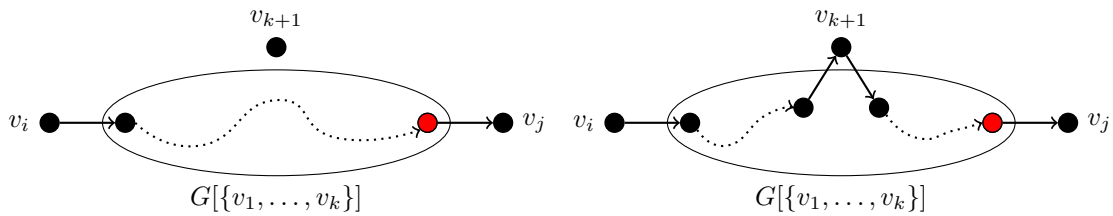
- La clase pasada vimos cómo computar un v-ACM a partir de un v-DPM en $O(n+m)$ tiempo.
- Aplicando n veces el algoritmo podemos construir la matriz de caminos mínimos a partir de la matriz de pesos mínimos.
- Sin embargo, podemos también calcular esta matriz directamente en Floyd-Warshall sin necesidad de implementar un algoritmo adicional.
- Para ello, alcanza con almacenar la matriz de k-caminos mínimos para cada k.
- Recordemos que, de acuerdo a (1), hay dos posibilidades para $\hat{\varphi}^k(v_i, v_j)$:

  Posibilidad 1: $\hat{\varphi}^k(v_i, v_j) = \hat{\varphi}^{k-1}(v_i, v_j)$, en cuyo caso no hay necesidad de cambiar el camino mínimo de la iteración anterior.

  Posibilidad 2: $\hat{\varphi}^k(v_i, v_j) = \hat{\varphi}^{k-1}(v_i, v_k) + \hat{\varphi}^{k-1}(v_k, v_j)$, en cuyo caso el nuevo camino llega a v_j de la misma forma en que lo hace el camino $\hat{\varphi}^{k-1}(v_i, v_j)$. En consecuencia, debemos actualizar el padre de v_j en el v-ACM hacia v_i para que se corresponda al padre de v_j en el v-ACM hacia v_k.

Actualización del predecesor en Floyd-Warshall

A la izquierda el caso en que el camino $(k+1)$-interno P no pasa por v_{k+1} y a la derecha el caso en que sí pasa. En ambos casos, el nuevo predecesor es el vértice rojo.



- El algoritmo de Floyd-Warshall completo en C++ queda de la siguiente forma.

Algoritmo de Floyd-Warshall (floyd-warshall.cpp)

```
1 //definición de los tipos de datos ...
2 const int none = -1;
3
4 //Precondición: n = |V(G)|, hay arista vv con c(vv) = 0
5 tuple<matrix,matrix,bool> FloydWarshall(const graph& G, int n) {
```

```
  matrix D(n, vector<int>(n, infty));  //Matriz de pesos
  matrix P(n, vector<int>(n, none));   //Matriz de caminos
  bool c = false;                      //Existe ciclo negativo
  for(auto e: G) {D[v(e)][w(e)] = c(e); P[v(e)][w(e)] = v(e);}
  for(int k = 0; k < n and not c; ++k) for(int i = 0; i < n and not c; ++i) {
    for(int j = 0; j < n; ++j) if(D[i][j] > D[i][k] + D[k][j]) {
      D[i][j] = D[i][k] + D[k][j];
      P[i][j] = P[k][j];
    }
    c = D[i][i] < 0;
  }
  return {D,P,c};
}
```

2.4. Variante inductiva

- En esta subsección discutimos brevemente una variante presentada por Dantzig [1].
- Por simplicidad, suponemos que el (di)grafo G no tiene ciclos de peso negativo y que sus vértices son $v_1, \ldots, v_n$.
- Generalicemos la función σ vista la clase pasada de forma tal que, para $k \geq 0$, $\sigma^k(v_i, v_j)$ es el mínimo de los pesos de los caminos de v_i a v_j que usan sólo vértices de $G_k = G[\{v_1, \ldots, v_k\}]$.
- El objetivo es computar $\sigma^n = \delta$.
- Para ello, tomemos un camino mínimo P desde v_i a v_j en G_k y consideremos los siguientes casos:

  Caso 1: P no contiene a v_k. En este caso, P es un camino de G_{k-1}.

  Caso 2: $j = k$ (resp. $i = k$). En este caso, P se puede descomponer en un camino de G_{k-1} desde v_i (resp. v_p) a v_p (resp. v_j) y una arista $v_p v_k$ (resp. $v_k v_p$).

  Caso 3: $j \neq k$ y P contiene a v_k. En este caso, $P = P_1 + P_2$, donde P_1 es un camino de v_i a v_k en G_k y P_2 es un camino de v_k a v_j en G_k.

- Ejercicio: esquematizar los casos y discutir con los compañeros de cursada.
- Luego, tenemos la siguiente definición recursiva para σ.

$$\sigma^k(v_i, v_j) = \begin{cases} 0 & \text{si } i = j = k \\ \text{mín}\{\sigma^{k-1}(v_i, v_p) + c(v_p v_k) \mid v_p \in N^{\text{in}}(v_k)\} \mid \text{si } i < j = k & \\ \text{mín}\{\sigma^{k-1}(v_p, v_j) + c(v_k v_p) \mid v_p \in N^{\text{out}}(v_k)\} \mid \text{si } k = i > j & \\ \text{mín}\{\sigma^{k-1}(v_i, v_j), \sigma^k(v_i, v_k) + \sigma^k(v_k, v_j)\} & \text{si } i, j < k \end{cases}$$

- El algoritmo resultante se puede implementar en $O(n^3)$ tiempo y $O(n^2)$ espacio.
- La detección de ciclos de peso negativo y la construcción del MCM pueden ser agregadas en una forma similar a como hicimos con Floyd-Warshall (en particular, es necesario cambiar la ecuación para el caso $i = j = k$).
- Los detalles se dejan como ejercicio y se ven en mayor profundidad en la práctica.

3. Algoritmo de Johnson

- Notemos que si $c \geq 0$, el algoritmo más eficiente que tenemos para calcular la MCM consiste en la aplicación sucesiva de Dijkstra.
- Su costo temporal es $O(nm + n^2 \log n)$ que es $o(n^3)$ cuando $m = o(n^2)$. Es decir, mejora el caso ralo.
- El algoritmo de Johnson [3] explota este hecho en dos pasos:

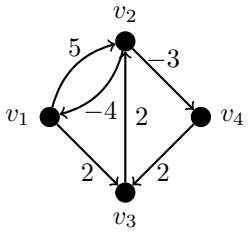
  Paso 1: construir una función de costos $\hat{c} \geq 0$ que respete el orden de los pesos de los caminos entre un par de extremos. Es decir, $\hat{c}_+(P) \leq \hat{c}_+(Q)$ para dos caminos P y Q de v a w si y sólo si $c_+(P) \leq c_+(Q)$. En consecuencia, P es un camino mínimo de $(G, \hat{c})$ si y sólo si P es un camino mínimo de (G, c).

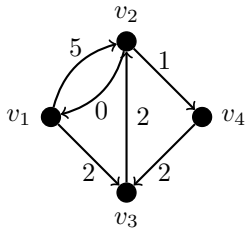
  Paso 2: construir la MCM P de $(G, \hat{c})$ que, por construcción, también es una MCM de (G, c).

  Paso opcional: construir la MPM de (G, c) a partir de P.

Cambio de pesos de un digrafo

A la izquierda un digrafo con pesos negativos y la derecha un digrafo con pesos positivos y misma matriz P de caminos mínimos.



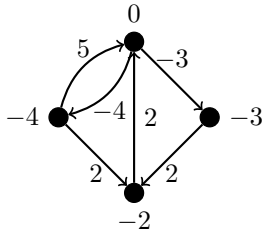


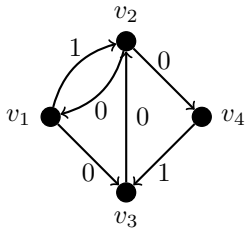
$$P = \left(\begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline v_1 & 1 & 3 & 1 & 2 \\ v_2 & 2 & 2 & 4 & 2 \\ v_3 & 2 & 2 & 3 & 2 \\ v_4 & 2 & 3 & 4 & 4 \end{array}\right)$$

- Sea H el (di)grafo que se obtiene de G agregando una fuente s y una arista sv de costo 0 para todo $v \in V(G)$.
- Para $vw \in E(H)$, definamos $\hat{c}(vw) = c(vw) + \delta(s, v) - \delta(s, w)$.
- El siguiente teorema muestra que $\hat{c}$ satisface las condiciones requeridas cuando G no tiene ciclos de peso negativo.

Generación de $\hat{c}$ con la función de Johnson

A la izquierda el grafo G donde cada vértice esta etiquetado con $\delta(s, v)$ y cada arista roja esta en el s-ACM, para la fuente s que no se muestra. A la derecha el grafo pesado con $\hat{c}$.





Teorema 1. *Sea H un (di)grafo con una fuente s que tiene una arista sv de costo $c(sv) = 0$ para todo $v \in V(H)$, donde c es una función de costos de H. Para una función $f\colon V(G) \to \mathbb{R}$, definamos $\hat{c}(v, w) = c(vw) + f(v) - f(w)$ para todo $vw \in E(H)$. Luego:*

1. $\hat{c}_+(P) = c_+(P) + f(v) - f(w)$ *para todo camino* P *de* v *a* w,
2. $\hat{c}_+(P) \geq \hat{c}_+(Q)$ *si y sólo si* $c_+(P) \geq c_+(Q)$ *para todo par de caminos* P *y* Q *de* v *a* w, *y*
3. *si* (H, c) *no tiene ciclos de peso negativo y* $f(v) = \delta_c(s, v)$ *para todo* v, *entonces* $\hat{c}(vw) \geq 0$ *para todo* $vw \in E(H)$.

Demostración. 1. Sea $P = v_1, \ldots, v_k$ un camino de H. Por definición:

$$\begin{aligned}\hat{c}_+(P) &= \sum_{i=1}^{k-1} \hat{c}(v_i v_{i+1}) = \sum_{i=1}^{k-1} (c(v_i v_{i+1}) + f(v_i) - f(v_{i+1})) \\ &= \sum_{i=1}^{k-1} c(v_i v_{i+1}) + f(v_1) - f(v_2) + f(v_2) - f(v_3) + \ldots + f(v_{k-1}) - f(v_k) \\ &= c_+(P) + f(v_1) - f(v_k)\end{aligned}$$

2. Por 1, $\hat{c}_+(P) = c_+(P) + h(v) - h(w) \geq c_+(Q) + h(v) - h(w) = \hat{c}_+(Q)$ si y sólo si $c_+(P) \geq c_+(Q)$.

3. Como (H, c) no tiene ciclos de peso negativo, $\delta_c(v, w)$ denota también el peso del camino mínimo no necesariamente simple de v a w. Luego, considerando que al agregar vw a cualquier camino de s a v obtenemos un camino de s a w, tenemos que $\delta_c(s, w) \leq \delta_c(s, v) + c(vw) \leq \hat{c}(vw) + \delta(s, w)$. □

- De acuerdo al Teorema 1, podemos resolver el paso 1 en tres subpasos:
  1. Crear el grafo H agregando la fuente s
  2. Invocar Bellman-Ford desde s para calcular δ; en caso que Bellman-Ford encuentre un ciclo de peso negativo, el algoritmo termina informando que G tiene un ciclo de peso negativo.
  3. Computar $\hat{c}$ a partir del s-DPM de G
- Para el paso 2 aplicamos Dijkstra iterativamente.
- El algoritmo completo, cuyo costo temporal es $T(n + m) = O(nm + n^2 \log n)$, se describe a continuación.

Algoritmo de Johnson

```
Johnson(G, c):
  Crear el (di)grafo H = G + s agregando una arista sv de peso 0 para todo v ∈ V(G)
  Ejecutar BellmanFord(H, c, s) para obtener un s-DPM D.
  En caso que BellmanFord encuentre un ciclo de peso negativo, terminar.
  Crear la función ĉ tal que ĉ(vw) = c(vw) + D[v] − D[w] para vw ∈ E(H)
  Crear una matriz M con dominio V(G).
  Para v ∈ V(G):
    Ejecutar Dijkstra(H, ĉ, v) para computar el v-ACM T de (H, ĉ)
    Poner M[v] = T − s
  Retornar M
```

Referencias

[1] George B. Dantzig. All shortest routes in a graph. In P. Rosenstiehl, editor, *Theorie Des Graphes*, pages 91–92. Dunod, Paris, 1966.

[2] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[3] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. Assoc. Comput. Mach.*, 24(1):1–13, 1977.

[4] Stephen Warshall. A theorem on boolean matrices. *J. Assoc. Comput. Mach.*, 9:11–12, 1962.

A. Implementación de los algoritmos en C++

En esta sección implementamos algunos de los algoritmos en C++. El objetivo es mostrar cómo se traducen los algoritmos coloquiales a C++, eliminando posibles ambigüedades.[3]

A.1. Algoritmo de Johnson

Para la implementación del algoritmo de Johnson copiamos los algoritmos de Bellman-Ford y Dijkstra vistos la clase pasada. La función `weight` es la encargada de implementar el nuevo costo $\hat{c}$. Notar que, en consecuencia, la cola de prioridad ordena por el valor de $\hat{c}_+$ mientras que costo real c_+ del camino se almacena en la última componente de cada elemento.

Implementación del algoritmo de Johnson (johnson.cpp)

```cpp
using neigh = pair<int, int>;             //first = to, second = cost
using graph = vector<vector<neigh>>;
using matrix = vector<vector<int>>;
using bridge = tuple<int, int, int, int>; //costo repesado, v, w, costo real

int main() {
  //leer lista de aristas y pasarlo a adyacencias

  //Agregar una fuente
  for(int v = 0; v < n; ++v) G[n].push_back({v,0});

  //Bellman-Ford en su propio scope para no contaminar: copia del visto
  vector<int> BF(n, infty);
  {
    vector<int> M(n, false);
    BF[r] = 0; bool changed = M[r] = true;
    for(int i = 0; i <= n and changed; ++i) {
      changed = false;
      for(int v = 0; v < n; ++v) if(M[v]) {
        M[v] = false;
        for(auto e: G[v]) if(BF[v] + cost(e) < BF[to(e)])
        {
          M[to(e)] = changed = true;
          BF[to(e)] = BF[v] + cost(e);
        }
      }
    }
    if(changed) {cout << "Ciclo negativo detectado"; return 0;}
  }

  //función de costos nueva
  auto weight = [&](int v, neigh e){return cost(e) + BF[v] - BF[to(e)];};

  //Dijkstra ya visto ejecutado n veces.  S ordena por peso nuevo; D mantiene el correcto
  matrix D(n, vector<int>(n));
  matrix P(n, vector<int>(n, none));
  for(int r = 0; r < n; ++r) {
    P[r][r] = r; D[r][r] = 0;
    priority_queue<bridge> S;
```

[3] Por falta de tiempo, los algoritmos fueron testeados superficialmente.

```
    for(auto x : G[r]) S.push({-weight(r, x), r, to(x), cost(x)});
    while(not S.empty()) {
      int p, v, w, c; tie(p,v,w,c) = S.top();
      S.pop();
      if(P[r][w] == none) {
        P[r][w] = v; D[r][w] = c;
        for(auto x : G[w]) if(P[r][to(x)] == none)
          S.push({p-weight(w, x), w, to(x), c+cost(x)});
      }
    }
  }

  //output D + P
}
```