Notas de la clase 6 – camino mínimo con único origen

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: 180 minutos

1. Problemas de camino mínimo con único origen (40 mins)

1.1. Camino mínimo elemental

- Recordemos que un vértice v de un (di)grafo G *alcanza* a un vértice w cuando existe un camino de v a w en G.
- La distancia $d(v, w)$ de v a w es el mínimo de la cantidad de aristas de los caminos de v a w.
- En la clase 4 vimos cómo resolver el siguiente problema del camino más corto.
- Problema del camino más corto:

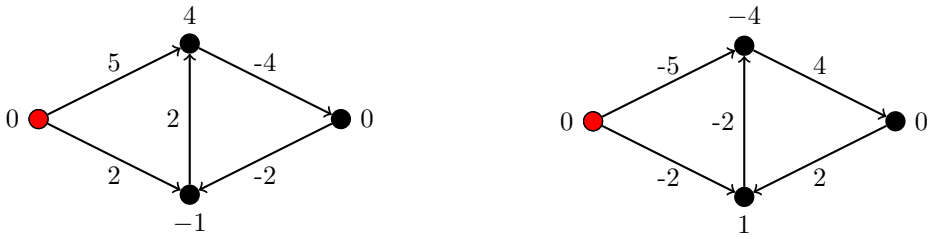
  Input: un (di)grafo G y dos vértices v y w tal que v alcanza a w.

  Output: un camino de v a w con $d(v, w)$ aristas.

- Como ya vimos, el problema del camino más corto se puede resolver en tiempo $O(n + m)$ aplicando BFS.
- En esta clase consideramos el problema de camino mínimo (resp. máximo) donde las aristas pueden tener distintos pesos.
- Dado un (di)grafo G con una función de costos $c\colon E(G) \to \mathbb{R}$, el *peso* o *costo* de un camino P es $c_+(P) = \sum_{vw \in E(P)} c(vw)$. (Nota: es la misma función de costos que usamos para AGM, i.e., sumar los pesos de las aristas.)
- Definamos $\delta_{G,c,+}(v, w) = \text{mín}\{c_+(P) \mid P \text{ es un camino } \textbf{simple} \text{ de } v \text{ a } w\}$.
- Como ejemplo, notar que $\delta_{G,1,+}(v, w) = d(v, w)$.
- A cada camino simple de peso $\delta_{G,c,+}(v, w)$ se lo llama *camino (de peso) mínimo* de (G, c).
- En general, omitimos algunos subíndices de $\delta_{G,c,+}$ cuando no hay posibilidad de confusiones. Por ejemplo, si escribimos $\delta_G(v, w)$ nos referimos a $\delta_{G,c,+}(v, w)$, mientras que si escribimos $\delta_{c,+}$ nos referimos a $\delta_{G,c,+}$.

Caminos de peso mínimo y máximo

En el digrafo de la izquierda, cada arista está etiquetada con su peso y cada vértice está etiquetado con el costo del camino mínimo desde el vértice rojo. En el digrafo de la derecha, el costo el inverso aditivo del costo del digrafo izquierdo, mientras que la etiqueta de cada vértice v indica máx$\{c_+(P)\}$ de entre los caminos que terminan en v. Observar que los caminos de ambos ejemplos coinciden.



Observación 1. *Sean P_1 y P_2 dos caminos de un (di)grafo G y $f = -c$ para una función de costos c. Entonces, $c_+(P_1) \leq c_+(P_2)$ si y sólo si $f_+(P_1) \geq f_+(P_2)$.*

- En función de la observación anterior, decimos que P es un *camino (de peso) máximo* de (G, c) cuando P es un camino de peso mínimo de $(G, -c)$.
- Problema de camino mínimo (resp. máximo) **elemental**:

  Input: un (di)grafo G, una función de costos c, un vértice v y un vértice w alcanzable por v.

  Output: un camino de peso mínimo de (G, c) (resp. $(G, -c)$) desde v a w.

- El problema del camino más corto es un caso particular del problema de camino mínimo (resp. máximo) elemental en el cual $c(vw) = 1$ (resp. $c(vw) = -1$) para todo $vw \in E(G)$.
- Ejemplo de un aplicación del problema de camino mínimo elemental:

  ▷ Supongamos que tenemos un camión de fletes con el que queremos visitar una serie de clientes $V = v_1, \ldots, v_n$.

  ▷ Los clientes se pueden visitar en cualquier orden, y hay un costo monetario (que surge del tiempo, nafta, etc.) $c(v_iv_j) \geq 0$ para viajar a v_j inmediatamente después de atender a v_i.

  ▷ El camión parte de un garage v_0 y, al finalizar el recorrido, se debe llevar de nuevo al garage $v_{n+1} = v_0$. En consecuencia, también hay un costo $c(v_0v_i)$ y $c(v_iv_{n+1})$.

  ▷ Por cada cliente que se atiende se obtiene un beneficio monetario $b(v_i) \geq 0$ (y suponemos $b(v_0) = b(v_{n+1}) = 0$).

  ▷ A priori no es necesario atender a todos los clientes, pero sí se sabe que cada cliente se atiende una única vez.

  ▷ Dado que cada cliente requiere poco espacio del camión, podemos considerar que el camión tiene capacidad infinita.

  ▷ Si queremos encontrar la ruta que maximice el beneficio, podemos modelar el problema usando un digrafo G que tenga una arista v_iv_j para cada $0 \leq i, j \leq n+1$ $(i \neq j)$ cuyo costo es $\bar{c}(v_iv_j) = c(v_iv_j) - b_j$.

  ▷ El peso de cada camino simple representa el saldo (costo si es positivo; beneficio si es negativo) de visitar a los clientes del camino.

  ▷ Luego, el camino de beneficio máximo es el camino mínimo elemental de $(G, \bar{c})$.

- Desafortunadamente, no se conocen algoritmos polinomiales para el problema de camino mínimo elemental.

- Peor aún, hay evidencias fuertes que sugieren que dicho algoritmo no existe.

- Por esta razón, se suelen estudiar problemas de camino mínimo restringidos, para los que sí se conocen algoritmos polinomiales.

1.2. Camino mínimo (no elemental)

- Decimos que v es un *origen*[1] cuando v alcanza a todos los otros vértices de G.

- Para los problemas de esta clase vamos a suponer que G contiene, al menos, un origen. Esto implica que G es conexo, aunque no tiene por qué ser fuertemente conexo.

- Nuestra suposición simplifica un poco la terminología y no hay una mayor perdida de generalidad, ya que al buscar un camino desde un vértice v, los vértices no alcanzables por v son irrelevantes.

- En la clase 4 no sólo resolvimos el problema de camino más corto de la sección anterior. En rigor, resolvimos esta generalización.

- Problema del camino más corto desde un origen, refraseado:

  Input: un (di)grafo G y un origen v.

  Output: un diccionario D y árbol generador T de G (enraizado en v) tal que $D[w] = d_G(v, w) = d_T(w, v)$ para todo $w \in V(G)$.

- La importancia de contar con el árbol T es que se pueden responder queries de camino mínimo tal cual vimos para el problema del camino más corto: dado w, se puede obtener el predecesor de w en un camino mínimo en $O(1)$ tiempo.

- El concepto del árbol BFS se puede generalizar para cualquier función de costos.

- Para un origen $v \in V(G)$, un *árbol de caminos mínimos desde* v de (G, c) es un árbol generador T de G con raíz v tal que $\delta_G(v, w) = \delta_T(w, v)$ para todo $w \in V(G)$.

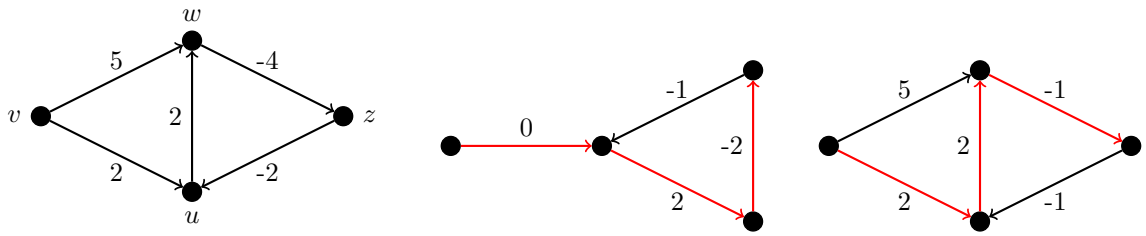
- Por simplicidad, vamos a decir que T es un v-ACM de (G, c) en lugar de decir que T es un árbol de caminos mínimos desde v.

- Como ejemplo, el árbol BFS desde v es un v-ACM de $(G, 1)$.

- A diferencia de lo que ocurre con el árbol BFS, puede ocurrir que un digrafo G no tenga v-ACMs de (G, c).

Árbol de caminos mínimos desde v

El grafo de la izquierda no admite v-ACMs, ya que todas las aristas del ciclo u, w, z son requeridas por algún camino mínimo. El del centro y la derecha sí admiten v-ACMs que corresponden a las aristas rojas.



[1] En inglés es source, que tiene otro significado que, en esta materia, traducimos como fuente. No confundir las dos acepciones del término source.

- Hay una diferencia muy importante entre los tres grafos de la figura previa: el de la derecha es el único que no tiene un ciclo de peso negativo.

- Más adelante vemos que el problema de camino mínimo para grafos sin ciclos negativos es un caso importante del problema.

- Por ahora, simplemente mostramos que los grafos sin ciclos de peso negativo admiten ACMs.

- Más aún, estos grafos tienen una segunda propiedad importante: los subcaminos de los caminos mínimos son a su vez caminos mínimos.

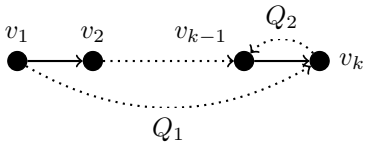
- Esta propiedad resulta fundamental para construir algoritmos golosos y de programación dinámica.

Teorema 1. *Sea v un origen de un (di)grafo G y c una función de costos. Si G no tiene ciclos de peso negativo, entonces:*

1. *todos los subcaminos de un camino mínimo son camino mínimos,*
2. *existe un v-ACM de (G, c),*
3. *si $c(xy)$ y $\delta(v, x)$ se pueden obtener en $O(1)$ tiempo para todo $x, y \in V(G)$, entonces se puede computar un v-ACM de G en tiempo $O(n + m)$.*

Demostración. 1. Sea $P = v_1, \ldots, v_k$ un camino mínimo de G. Veamos por inducción que todos sus subcaminos son mínimos a no ser que G tenga un ciclo de peso negativo. El caso base $k = 1$ es trivial. Supongamos, pues, que $k > 2$. Si $P - v_k$ y $P - v_1$ son caminos mínimos, entonces todos los subcaminos de P son mínimos por hipótesis inductiva. Supongamos, entonces, que $P - v_k$ no es un camino mínimo (el caso $P - v_1$ es análogo). Esto significa que existe otro camino simple Q de v_1 a v_{k-1} con $c_+(Q) < c_+(P - v_k)$. Ciertamente, $Q + v_k$ es un camino de v_1 a v_k con $c_+ < c_+(P)$. Luego, como P es mínimo, esto significa que $Q + v_k$ no es simple. Pero, como Q sí es simple, la única posibilidad es que Q contenga a v_k. Luego, Q se puede descomponer en un camino Q_1 desde v_1 a v_k y otro Q_2 desde v_k a v_{k-1}. Como Q_1 es un camino simple, obtenemos que $c_+(Q_1) \geq c_+(P)$. Pero entonces, como $c_+(Q + v_k) < c_+(P)$, esto significa que $c_+(Q_2 + v_k) < 0$. Es decir, el ciclo $Q_2 + v_k$ tiene peso negativo.

Los subcaminos de un camino mínimo son mínimos



2. Sea H el sub(di)grafo generador de G tal que $xy \in E(H)$ si y sólo si $\delta(v, y) = \delta(v, x) + c(xy)$. (Ver figura abajo.) Si xy es la última arista de un camino mínimo P de G, entonces $P - y$ es un camino con peso $\delta(v, y) - c(xy)$. Por 1., $c_+(P - y) = \delta(v, x)$, con lo cual xy pertenece a H. Como esto vale para cualquier camino mínimo, 1. implica que todas las aristas de P pertenecen a H. En consecuencia, (1) v es un origen de H. Recíprocamente, (2) si P es un camino (no necesariamente simple) de H cuya última arista es xy, entonces $c_+(P) = \delta(v, y)$. En efecto, $\delta(v, y) = \delta(v, x) + c(xy)$ por definición. Luego, como $c_+(P - y) = \delta(v, x)$ por hipótesis inductiva, obtenemos que $c_+(P) = c_+(P - y) + c(xy) = \delta(v, x) + c(xy) = \delta(v, y)$.

Grafo de caminos mínimos

Un digrafo G a la izquierda y su subdigrafo H a la derecha; la etiqueta de cada $y \in V(G)$ es $\delta(v, y)$ donde v es el vértice rojo.

Como v es un origen de H por (1), cualquier árbol generador T de H enraizado en v tiene un único camino simple P_y de y a v para todo $y \in V(G)$. (En la figura, las aristas rojas forman un árbol generador T.) Más aún, como el camino reverso de P_y es un camino de H, entonces $\delta_T(y, v) = c_+(P_y) = \delta_G(v, y)$ por (2). En otras palabras, T es un v-ACM de (G, c).

3. En cuanto a la complejidad de computar T, notemos primero que podemos construir H en $O(n + m)$ tiempo. Para esto alcanza con recorrer cada arista xy y revisando si $\delta(v, y) = \delta(v, x)+c(xy)$; esta comparación requiere $O(1)$ tiempo por hipótesis. Una vez obtenido H, se puede computar T con cualquier algoritmo lineal, e.g., BFS o DFS. □

- Un *diccionario de pesos mínimos* de v es un diccionario D con claves en $V(G)$ tal que $D[w] = \delta(v, w)$ para todo $w \in V(G)$. Decimos que D es un v-DPM.

- El Teorema 1 justifica la definición del siguiente problema.

- Problema de camino mínimo (resp. máximo) desde un origen:[2]

  Input: un (di)grafo G, una función de costos c, y un origen v.

  Output: un ciclo de peso negativo (resp. positivo) o un v-DPM y un v-ACM de (G, c) (resp. $(G, -c)$).

- Al igual que para el problema del camino más corto, la importancia de contar tanto con un v-DPM y con v-ACM es que se pueden responder queries de distancia y camino mínimo en $O(1)$ tiempo.

- Sin embargo, para el problema de camino mínimo alcanza con mostrar cómo computar uno de ellos, dado que el otro se puede obtener en tiempo lineal por Teorema 1.[3]

- Es importante observar la terminología. Aún cuando δ_G está bien definido para cualquier (di)grafo G, en el problema de "camino mínimo" a secas el output puede ser un ciclo de peso negativo. En cambio, al problema de encontrar los caminos para grafos con pesos arbitrarios se lo llama problema de "camino mínimo **elemental**".[4]

- Quizá sería mejor llamar problema del "recorrido mínimo" a uno y de "camino mínimo" al otro. Las razones para no hacerlo son históricas y es importante entender la terminología para evitar confusiones.

- El motivo por el cual el problema de camino mínimo (a secas) excluye a los (di)grafos que tienen ciclos de peso negativo es que en estos grafos el valor $\min\{c_+(P) \mid P \text{ camino de } G\}$ no está bien definido. En cambio, este valor coincide con δ cuando cuando no hay ciclos negativos.

Teorema 2. *Sea v un origen de un (di)grafo G, c una función de costos. Entonces, G tiene un ciclo de peso negativo si y sólo si $\hat{\delta}(v, w) = \min\{P \mid P \text{ es un camino de } v \text{ a } w\}$ existe para todo $w \in V(G)$. Más aún, si G no tiene ciclos de peso negativo, entonces $\hat{\delta}(v, w) = \delta(v, w)$ para todo $w \in V(G)$.*

[2] Por sus siglas en ingles, se lo suele llamar problema SSSP (single source shortest path).

[3] Y esto nos simplifica un poco esta clase larga; sin embargo, muchas veces es conveniente computar ambos resultados a la vez porque es más simple de implementar.

[4] Por cuestiones históricas, en algunos textos se llama problema de "camino máximo" al problema de "camino máximo elemental" y "camino mínimo" al problema de "camino mínimo no elemental". Creemos que esto es desafortunado y provoca confusiones.

Demostración. Si $c_+(C) < 0$ para un ciclo $C = v_1, \ldots, v_k, v_1$, entonces kC es un ciclo de longitud $kc_+(C) < \ell$ para cualquier $k \in \mathbb{Z}$ y $\ell \in \mathbb{R}$ con $kc_+(C) < \ell$. Luego, $\hat{\delta}(v, w)$ no puede ser ℓ y, por tal motivo, $\hat{\delta}(v, w)$ no está bien definido.

Supongamos ahora que $c_+(C) \geq 0$ para todo ciclo C de G. Si P es un camino simple de G de v a w, entonces $c_+(P) \geq \delta(v, w)$ por definición. En cambio, si $P = v_1, \ldots, v_k$ contiene un ciclo $C = v_i, \ldots, v_j$, entonces $P' = v_1, \ldots, v_i, v_{j+1}, \ldots, v_k$ es un camino tal que $c_+(P') = c_+(P) - c_+(C)$. Por hipótesis, $c_+(C) \geq 0$, con lo cual $c_+(P) \geq c_+(P') \geq \delta(v, w)$ por inducción. Luego, $\hat{\delta}(v, w)$ está bien definido y es mayor o igual a $\delta(v, w)$. Como, obviamente, $\hat{\delta}(v, w) \leq \delta(v, w)$ cuando está definido, concluimos que $\hat{\delta}(v, w) = \delta(v, w)$. □

- En la Sección 3.4 damos un aplicación del problema de camino mínimo no elemental.

1.3. Casos particulares de camino mínimo

- Hay dos casos particulares del problema de camino mínimo que tienen gran importancia y que se pueden resolver más eficientemente con algoritmos especialmente diseñados para estos casos.

- Problema de camino mínimo (resp. máximo) con pesos no negativos (resp. positivos) desde un origen.

  Input: un (di)grafo G, una función de costos $c \geq 0$ (resp. $c \leq 0$) y un origen v.

  Output: un v-DPM y un v-ACM de (G, c) (resp. $(G, -c)$).

- Este problema es un caso particular de camino mínimo, porque ningún ciclo tiene peso negativo.

- Las aplicaciones de este problema son las más conocidas, como la búsqueda del camino más rápido en un mapa.

- Veremos otras aplicaciones en las clases prácticas.

- Decimos que un (di)grafo es *acíclico* cuando no contiene ciclos.

- A estos (di)grafos los llamamos DAGs por sus siglas en inglés (directed acyclic graphs).

- Por definición, los árboles orientados son DAGs.

- Un vértice v de un digrafo es una *fuente* cuando $d^{\text{in}}(v) = 0$ y es un sumidero cuando $d^{\text{out}}(v) = 0$.

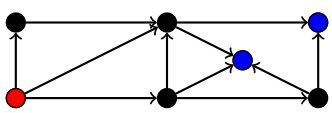
Lema 1. *Todo DAG G tiene al menos una fuente y un sumidero.*

Demostración. Sea $P = v_1, \ldots, v_k$ el camino simple de mayor longitud de G. Si $v_0 v_1 \in E(G)$ para algún $v_0 \in V(G)$, entonces $v_0 = v_i$ porque $v_0, \ldots, v_k$ no es un camino simple. Como esto es imposible, porque G no puede tener un ciclo $v_0, \ldots, v_i$, obtenemos que $d^{\text{in}}(v_1) = 0$. Análogamente se puede demostrar que $d^{\text{out}}(v_k) = 0$, hecho que también se puede demostrar tomando la orientación reversa de G. □

Corolario 1. *Si un DAG G tiene un origen, entonces G tiene una única fuente que es el origen.*

DAGs, fuentes y sumideros

Una fuente roja y dos sumideros azules.



- Problema de camino mínimo (resp. máximo) en un DAG.

Input: un DAG G con un origen y una función de costos c.

Output: un v-DPM y un v-ACM de (G, c) (resp. $(G, -c)$) para el origen v de G.

- Hasta ahora tenemos definidas cuatro variantes del problemas de camino mínimo: elemental, no elemental, con costos no negativos, y acíclico dirigido.
- En el resto de la clase vemos algoritmos para las tres variantes no elementales y damos posibles aplicaciones, que se profundizan en las clases prácticas y de laboratorio.
- Si bien en todas las variantes el objetivo es construir un v-DPM y un v-ACM, muchas veces uno está interesado en encontrar un camino entre dos vértices particulares v y w.
- No se conocen algoritmos generales que mejoren la eficiencia en peor caso para este problema.
- Sí se conocen heurísticas (como A*) para acelerar los algoritmos a este caso puntual. En esta teórica ignoramos estas variantes.

2. Camino mínimo con pesos no negativos (30 mins)

- La clase pasada vimos el problema de camino minimax entre todos los pares de vértices de un grafo.
- Ahora consideramos el problema de camino minimax en (di)grafos, pero partiendo desde un origen.
- Recordemos que un camino P de v a w en un (di)grafo G es minimax (resp. maximin) cuando $c_{\text{máx}}(P) \leq c_{\text{máx}}(P')$ (resp. $c_{\text{mín}}(P) \leq c_{\text{mín}}(P')$) para todo camino P' de v a w.
- Problema de camino minimax (maximin) desde un origen

  Input: un (di)grafo G, una función de costos c, y un origen v.

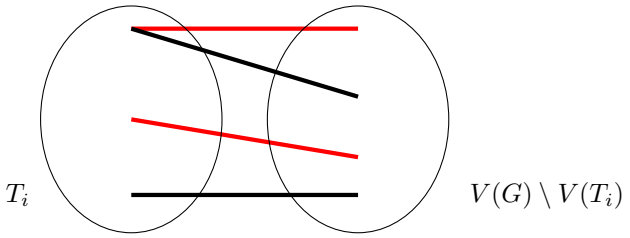
  Output: un árbol generador T de G tal que el camino de v a w en T es minimax (maximin) para todo $w \in V(G)$.

- Para el caso no dirigido, vimos que este problema se puede resolver computando un AGM T de G.[5]
- Vamos a ver que podemos generalizar el algoritmo de Prim para resolver también el caso dirigido y, de paso, resolver también el problema de camino mínimo con pesos no negativos.
- Recordemos que en el paso i Prim mantiene un AGM T_i de $G[V_i]$ para un conjunto V_i.
- Una arista xy en *segura* para T_i cuando $x \in V_i$ e $y \in V(G) \setminus V_i$.
- Para extender T_i a T_{i+1}, Prim agrega una arista segura xy de costo mínimo.

Esquema del algoritmo de Prim

Las aristas que cruzan son seguras, las rojas son candidatas.

[5] Reitero que este algoritmo no es el mejor, salvo que queramos encontrar los caminos minimax entre todos los pares de vértices a la vez.

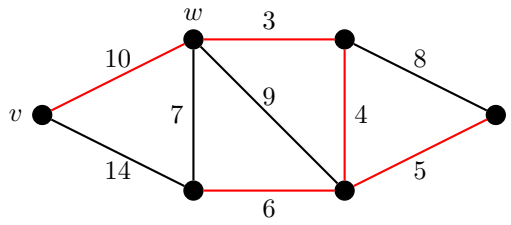


- Si en lugar de elegir una arista segura de peso mínimo, eligiéramos cualquier arista segura, al final también obtenemos un árbol generador T de G, aunque no podemos afirmar que T sea un AGM.

- En el caso de camino minimax tenemos otras opciones: si P es un camino de v a x, entonces $c_{\text{máx}}(P+y) = c_{\text{máx}}(P)$ para cualquier xy seguro con $c(xy) \leq c_{\text{máx}}(P)$.

Árbol generador mínimo versus caminos minimax

El grafo tiene un único AGM que esta pintado, aunque todos los caminos simples que contienen a vw son minimax porque todas las aristas que no inciden en v pesan menos que $c(vw)$.



- En consecuencia, podemos agregar cualquier arista segura xy de costo $c(xy) \leq c_{\text{máx}}(P)$, sea o no mínima.

- Para formalizar y generalizar esta idea, supongamos que tenemos una función $\bullet\colon \mathbb{R} \to \mathbb{R}$ de forma tal que el costo de $c_\bullet(P)$ de un camino $P = v_1, \ldots, v_k$ es:

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

- Por ejemplo, si $\bullet = +$, tenemos c_+, mientras que si $\bullet = \text{máx}$, tenemos $c_{\text{máx}}$.

- Decimos que $c_\bullet$ es *no decreciente* cuando $c_\bullet(P + y) = c_\bullet(P) \bullet c(xy) \geq c_\bullet(P)$ para todo camino $P + y$ con última arista xy.

- Ejemplos de funciones de caminos no decrecientes, donde c es una función de costos:
  - $c_{\text{máx}}$, ya que $c_{\text{máx}}(P + y) \geq c_{\text{máx}}(P)$.
  - c_+ cuando $c \geq 0$, ya que $c_+(P + y) = c_+(P) + c(xy) \geq c_+(P)$.
  - $c_\times$ para $c \geq 1$, ya que $c_\times(P + y) = c(xy) \cdot c_\times(P) \geq \phi(P)$.

- Contraejemplos, donde c es una función de costos:
  - c_+ en el caso general, ya que si $c(xy) < 0$, entonces $c_+(P + y) = c_+(P) + c(xy) < c_+(P)$

▷ $c_\times$ para el caso general (ejercicio).

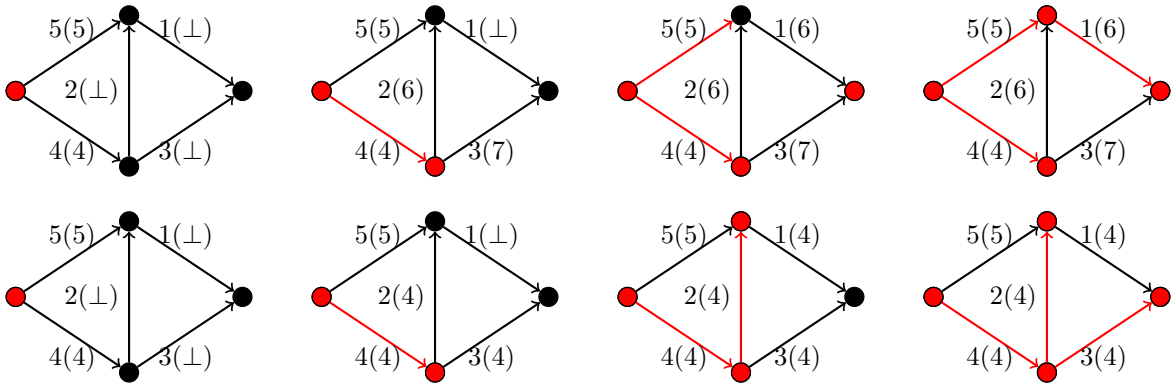
- Generalizando la sección anterior, podemos definir el $\bullet$-*peso* de un camino P como $c_\bullet(P)$.
- Luego, $\delta_{G,c,\bullet}(v,w) = \min\{c_\bullet(P) \mid P \text{ es un camino } \textbf{simple} \text{ de } v \text{ a } w\}$.
- A cada camino simple de peso $\delta_{G,c,\bullet}(v,w)$ lo podemos llamar *camino* $\bullet$-*mínimo* de (G,c). E.g., P es un camino de peso minimo cuando P es $+$-mínimo, mientras que P es minimax cuando P es máx-minimo.
- Más aún, podemos decir que un árbol generador T de G es un *árbol de caminos* $\bullet$-*mínimos* de v, o $(v,\bullet)$-*ACM*, cuando $\delta_{G,c,\bullet}(v,w) = \delta_{T,c,\bullet}(w,v)$ para todo $w \in V(G)$.
- Para una arista segura xy de un árbol T enraizado en v, llamemos $c_\bullet(xy) = c_\bullet(P) + c(xy)$ donde P es el único camino de x a y.
- El siguiente algoritmo, publicado por Dijkstra [3] en 1959, es una variante del algoritmo de Prim que funciona cuando $c_\bullet$ es no decreciente.

Algoritmo de Dijkstra para una función $\bullet$

```
1 Dijkstra-•(G, c, v):
2   Sea T = ({v}, ∅) un árbol
3   Para i = 1, ..., n − 1:
4     Agregar a T una arista segura xy de T con mínimo c•(xy).
```

Ejemplos de ejecución del algoritmo de Dijkstra

Arriba una ejecución de `Dijkstra-+` y abajo una de `Dijkstra-máx`, ambas desde el vértice rojo. En cada arista xy se muestra el valor $c(xy)$ junto al valor $c_\bullet(xy)$ entre paréntesis.

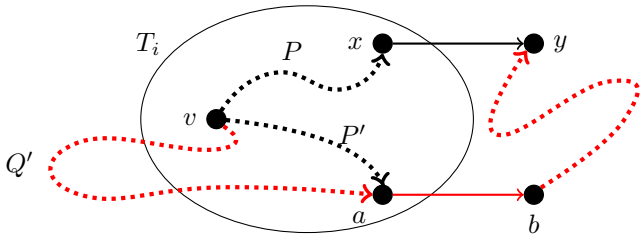


Teorema 3. *Si G un (di)grafo, c una función de costos, $\bullet\colon \mathbb{R} \to \mathbb{R}$ y v un origen. Si $c_\bullet$ es no decreciente, entonces `Dijkstra-•(G,c,v)` computa un $(v,\bullet)$-ACM T de (G,c).*

Demostración. Con la teoría desarrollada en la clase 4, es fácil ver que T es un árbol generador de G. Ciertamente, el camino simple de v a v en T tiene el menor valor de $c_\bullet$ porque es único. Supongamos, por inducción, que el camino de v a x en el árbol T_i computado hasta la i-ésima iteración es $\bullet$-minimo para todo $x \in V(T_i)$. Supongamos que en la iteración i el algoritmo elige la arista xy y llamemos P al camino de v a x en T_i. Claramente, cualquier camino $Q + y$ desde v a y tiene una arista ab tal que $a \in V(T_i)$ y ninguna arista del subcamino de Q desde b a y pertenece a T_i. (Ver figura abajo donde Q es el camino rojo, considerando que puede ocurrir que $a = x$ o $b = y$.) Luego, si llamamos Q' al subcamino de Q desde v hasta a y P' al

camino de T_i desde v hasta a, obtenemos que $c_\bullet(P') \leq c_\bullet(Q')$ por hipótesis inductiva. Más aún, como ab es segura para T_i y el algoritmo eligió xy, obtenemos que $c_\bullet(P + y) = c_\bullet(xy) \leq c_\bullet(ab) = c_\bullet(P' + b)$. Y, finalmente, como $c_\bullet$ es no decreciente, obtenemos que $c_\bullet(Q' + b) \leq c_\bullet(Q + y)$. En consecuencia, $c_\bullet(P + y) \leq c_\bullet(P' + b) \leq c_\bullet(Q' + b) \leq c_\bullet(Q + y)$. □

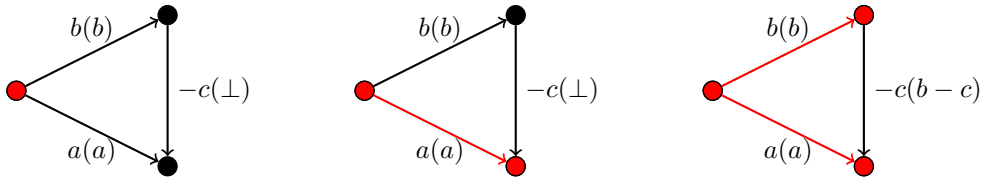
Demostración de la corrección de Dijkstra



- El algoritmo `Dijkstra-+` construye un v-ACM de (G, c), resolviendo el problema de camino mínimo con pesos no negativos.
- Remarcamos que, por definición, $\delta(v, y) = c_+(xy)$ cuando agregamos y a T.
- En consecuencia, podemos computar el v-DCM de (G, c) en forma simultánea (ver Apendice A.1).
- Es fácil mostrar un caso en que el algoritmo falla cuando hay aristas de peso negativo; alcanza con seguir la demostración y ver dónde se usa la hipótesis.

Ejemplo de que el algoritmo de Dijkstra falla con pesos negativos

Supongamos $0 < a < b$, $0 < b - a < c$. Aunque todas las distancias sean positivas, el algoritmo falla porque una vez que se agrega una arista al árbol T, ya no se modifica aunque después aparezca una arista mejor (después de todo, es un algoritmo goloso).



- Para implementar `Dijkstra-•`, podemos usar las mismas ideas que usamos para el algoritmo de Prim.
- La única salvedad es que en lugar de guardar $c(xy)$ para cada arista segura xy, tenemos que almacenar $c_\bullet(xy) = c_\bullet(P) \bullet c(xy)$, donde P es el único camino de v a x.
- El valor de $c_\bullet(P)$ ya lo tenemos calculado, de cuando agregamos a x al árbol T.
- Luego, tenemos tres implementaciones posibles, cuyas complejidades suponen que $\bullet$ se puede computar en $O(1)$ tiempo:

  Caso ralo: usar una cola de prioridad sobre heap con todas las aristas seguras. Costo $T(n + m) = O(m \log n)$

  Caso general: usar diccionario de costo que contenga el valor de la mejor arista segura para cada $y \notin V(T)$. Costo $T(n + m) = O(m + n \log n)$.

Caso denso: implementar el diccionario de costos sobre un vector. Costo $T(n+m) = O(n^2)$

Teorema 4. *El problema de camino mínimo (resp. máximo) con pesos no negativos (resp. positivos) desde un origen se puede resolver en tiempo* $O(m + n \log n)$.

Intervalo (10 mins)

3. Camino mínimo no elemental (40 mins)

- Recordemos que para resolver el problema de camino mínimo tenemos que computar, para un (di)grafo G con una función de costos c: un ciclo de peso negativo; o un v-DPM y un v-ACM de (G, c).

- Para esta clase vamos a relajar el problema, dejando la solución final para investigar en la bibliografía y en las clases prácticas.

- En particular, en lugar de computar un ciclo de peso negativo, vamos a restringirnos a detectar si dicho ciclo existe, pero sin computarlo explícitamente.

- Además, como un v-ACM se puede obtener en tiempo $O(n + m)$ desde un v-DPM (Teorema 1), vamos a concentrar la teoría casi exclusivamente en el computo del v-DPM.

- El algoritmo que vamos a ver se lo llama Bellman-Ford y se debe a Bellman [1], Ford [4] y Moore [5] (y probablemente alguien más).

- Una forma de presentar Bellman-Ford es como un algoritmo iterativo de relajación, en el que se construye en v-ACM de G iterativamente, permitiendo mejorar los caminos que se detectan no óptimos (e.g. [2]). [6]

- En esta clase, sin embargo, vamos a desarrollar Bellman-Ford como un ejercicio de programación dinámica, razón por la cual nos concentramos en las distancias y no en el árbol.

- En caso de querer computar un ciclo de peso negativo, si existe, conviene ir construyendo el v-ACM candidato mientras se aplica Bellman-Ford, siguiendo el razonamiento de relajación.

3.1. Bellman-Ford, versión 1.0

- Para todo $i \geq 0$, definamos:[7]

  - ▷ $\varepsilon^i(v, w) = \min\{c_+(P) \mid P$ es un camino **simple** de v a w con a lo sumo i aristas$\}$.
  - ▷ $\hat{\varepsilon}^i(v, w) = \min\{c_+(P) \mid P$ es un camino de v a w con a lo sumo i aristas$\}$.

- En caso en que algún valor no esté definido porque el conjunto es vacío, consideramos que $\varepsilon^i(v, w) = \infty$ o $\hat{\varepsilon}^i(v, w) = \infty$.

- Notemos que tanto $\varepsilon^i(v, w)$ como $\hat{\varepsilon}^i(v, w)$ están bien definidos porque, al igual que ocurre para $\delta(v, w)$, hay una cantidad finita de caminos de v a w de longitud menor o igual a i.

- Claramente $\delta(v, w) \leq \varepsilon^i(v, w)$ y, como todo camino simple tiene a lo sumo $n - 1$ aristas, también es cierto que $\delta(v, w) \geq \varepsilon^{n-1}(v, w) \geq \hat{\varepsilon}^{n-1}(v, w)$.

- Por otra parte, el Teorema 1 garantiza que si G no tiene ciclos de peso negativo, entonces $\delta(v, w) \leq \hat{\varepsilon}^i(v, w)$ para todo $i \geq 0$.

[6] Por este motivo, al algoritmo de Bellman-Ford se lo considera de "label correcting", mientras que a Dijkstra se lo considera de "label setting" en algunos libros. En estos contextos, label significa camino.

[7] ε es por distancia con restricción de aristas (εdges).

- Como corolario, en este caso particular, $\delta(v, w) = \hat{\varepsilon}^{n-1}(v, w)$.

- Luego, para resolver el problema simplificado de camino mínimo alcanza con computar $\hat{\varepsilon}^{n-1}(v, w)$ si G no tiene ciclos de peso negativo, o informar que G tiene un ciclo de peso negativo en caso contrario.

- Vemos primero cómo computar $\hat{\varepsilon}^{n-1}(v, w)$ y luego cómo determinar si G tiene un ciclo de peso negativo.

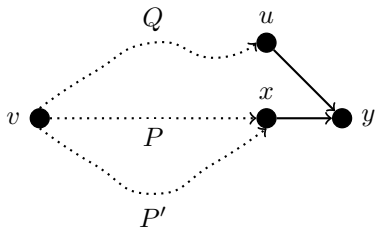
- Consideremos cualquier camino $P + y$ desde v cuya última arista es xy.

- Claramente, $P + y$ es un camino de peso mínimo de v a y con a lo sumo i aristas si y sólo si:

  1. P es un camino de peso mínimo de v a x con a lo sumo $i - 1$ aristas.
  2. $c_+(P) + c(xy) \leq c_+(Q) + c(uy)$ para cualquier camino Q de v a $u \in N^{\text{in}}(y)$ que use a lo sumo $i - 1$ aristas.

Corrección de la recurrencia de $\hat{\delta}$

Si $c_+(P') < c_+(P)$ y $|P'| < i$, entonces podemos reemplazar P con P'. De la misma forma, si $c_+(Q + y) < c_+(P + y)$, entonces tomaríamos $Q + y$. Notar que esto es un esquema y los caminos no tienen por qué ser disjuntos en general.



- Luego, podemos calcular $\hat{\varepsilon}^i$ con la siguiente recurrencia.

$$\hat{\varepsilon}^i(v, y) = \begin{cases} 0 & \text{si } i = 0 \text{ y } y = v \\ \infty & \text{si } i = 0 \text{ e } y \neq v \\ \text{mín}\{\hat{\varepsilon}^{i-1}(v, x) + c(xy) \mid x \in N^{\text{in}}[y]))\} & \text{caso contrario} \end{cases} \tag{1}$$

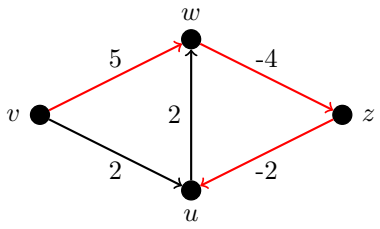
- Paréntesis: notar que estamos considerando $x \in N^{\text{in}}[y]$ en la tercer ecuación. En el caso $x = y$, suponemos $c(xy) = 0$. La razón de esto es simplificar la definición de $\hat{\delta}$ evitando indefiniciones.

- Es importante remarcar ahora que δ no se puede computar de la misma forma.

- En efecto, no es cierto que si $P + y$ es un camino **simple** con peso $\varepsilon^i(v, y)$, entonces P es un camino con peso $\varepsilon^{i-1}(v, x)$.

Diferencias entre δ y $\hat{\delta}$ en el caso general

El camino rojo v, w, z, u es un camino simple con peso mínimo $\varepsilon^3(v, u) = -1$, pero su subcamino v, w, z no es un camino con peso $\varepsilon^2(v, z) = 0$.



- Visto desde una perspectiva incremental, la razón por la que no podemos extender el camino mínimo v, u, w, z a u es porque deja de ser simple.

- Esto, en cambio, es irrelevante cuando uno calcula $\hat{\delta}$ ya que no garantizamos que los caminos encontrados sean simples.

- Volviendo a la definición de $\hat{\delta}$, podemos observar que la misma tiene la propiedad de *superposición de subproblemas*.

- En efecto, la cantidad de llamadas recursivas es exponencial (porque considera todos los posibles caminos), mientras que la cantidad de instancias posibles es $O(n^2)$ (porque $0 \leq i < n$, $y \in V(G)$ y el primer parámetro es siempre v).

- Esto garantiza que podemos aplicar un esquema de programación dinámica usando un diccionario de memoización.

- Si bien podemos implementar directamente el algoritmo en forma top-down (usando $\Theta(n^2)$ espacio), hagamos un paso más para implementar el algoritmo en forma bottom-up (visto en las clases prácticas) a fin de reducir el espacio consumido.

- Para cada $i \geq 0$, definamos un diccionario D^i tal que $D^i[w] = \hat{\varepsilon}^i(v, w)$ para todo vértice $v \in V(G)$.

- Por (1), $D^0[v] = \hat{\varepsilon}^0(v, v) = 0$ y $D^0(y) = \hat{\varepsilon}^0(v, y) = \infty$ para todo $y \neq v$.

- Por otra parte, para $i > 0$, $D^i[y] = \hat{\varepsilon}^i(v, y) = \min\{\hat{\varepsilon}^{i-1}(v, x) + c(xy) \mid x \in N^{\text{in}}[y])\} = \min\{D^{i-1}[x] + c(xy) \mid x \in N^{\text{in}}[y]\}$.

- Claramente, podemos ordenar las instancias de forma tal que D^i se calcule luego de D^{i-1}.

- Como vimos antes, si G no tiene ciclos de peso negativo, entonces $\hat{\varepsilon}^{n-1} = \delta$ y, por lo tanto, D^{n-1} es un v-DPM de (G, c); justo lo que queremos computar.

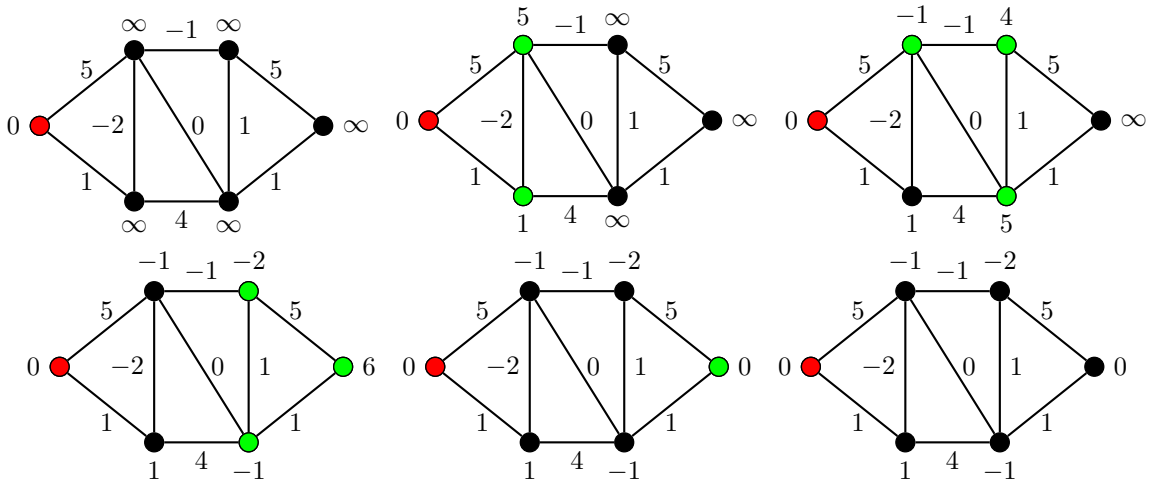
- Como consecuencia de este desarrollo, tenemos el siguiente algoritmo para calcular D^{n-1}.

Algoritmo de Bellman-Ford

```
BellmanFord(G, c, v):
  Crear un diccionario D^0 con D^0[v] = 0 y D^0[y] = ∞, y ≠ v
  Para i = 1, ..., n − 1:
    Crear un diccionario D^i[y] = mín{D^{i−1}[x] + c(xy) | x ∈ N^in[y]} para todo y ∈ V(G).
  retornar D^{n−1}
```

Ejecución de Bellman-Ford

`BellmanFord` desde un nodo rojo donde el valor calculado para D se muestra como etiquetas de los vértices y los vértices cuyos valores cambiaron de la iteración anterior se marcan en verde.



- El algoritmo realiza $O(n)$ iteraciones y cada iteración cuesta $O(m)$, ya que consiste en recorrer $N^{\text{in}}[y]$ para todo $y \in V(G)$.

- Notemos que D^{i-j} no es requerido para computar D^i con $j < i-1$, razón por la cual se pueden mantener sólo dos vectores: D^{i-1} y D^i.

- Luego, este algoritmo computa D^{n-1} en $O(nm)$ tiempo y espacio $O(n+m)$ (porque además de $O(n)$ espacio para D, requerimos $O(n+m)$ espacio para las listas de adyacencias).

3.2. Detectando ciclos de peso negativo

- Notemos que el algoritmo retorna $D^{n-1} = \hat{\varepsilon}^{n-1}(v, \bullet)$ existan o no ciclos con peso negativo.

- El siguiente teorema permite detectar si G tiene ciclos de peso negativo a partir de D^{n-1} y el diccionario D^n que se obtiene si se ejecuta una iteración más de `BellmanFord`.

Teorema 5. *Sea G un (di)grafo, c una función de costos, y v un origen de G. Entonces, G tiene algún ciclo de peso negativo si y sólo si $\hat{\varepsilon}n \neq \hat{\varepsilon}^{n-1}$.*

Demostración. Por (1), $\hat{\varepsilon}i + 1$ depende de los valores de $\hat{\varepsilon}^i$ y de c, siendo independiente de i. Luego, si $\hat{\varepsilon}n = \hat{\varepsilon}^{n-1}$, entonces $\hat{\varepsilon}n + i = \hat{\varepsilon}^{n-1}$ para todo $i \geq 0$. Esto significa que $c_+(P+y) \geq \hat{\varepsilon}^{n-1}(v,y)$ para todo camino P. En consecuencia G no tiene ciclos de peso negativo por Teorema 2.

Supongamos ahora que G no tiene ciclos de peso negativo y consideremos cualquier vértice $y \in V(G)$. De entre todos los caminos de v a y de peso $\hat{\varepsilon}^n(v,y)$, tomemos uno $P = v_0, \ldots, v_k$ cuya longitud sea mínima. Si P tuviera dos vértices $v_i = v_j$ con $i < j$, entonces, como $c_+(v_i, \ldots, v_j) \geq 0$, obtenemos que $v_0, \ldots, v_i, v_{j+1}, \ldots, v_k$ es un camino con $c_+ \leq c_+(P)$ y longitud menor a $|P|$. Como esto es imposible, P es un camino simple, con lo cual $k < n$. Esto implica que $\hat{\varepsilon}^k(v,y) \leq c_+(P)$, con lo cual $c_+(P) = \hat{\varepsilon}n(v,y) \leq \hat{\varepsilon}^{n-1}(v,y) \leq \ldots \leq \hat{\varepsilon}^k(v,y) \leq c_+(P)$. Aplicado a cada vértice y, esto implica que $\hat{\varepsilon}^n = \hat{\varepsilon}^{n-1}$. □

- Por el Teorema 5, el algoritmo de abajo detecta correctamente si G tiene un ciclo de peso negativo.

- Reiteramos que el algoritmo no encuentra dicho ciclo. Para esto hay que ir construyendo el v-ACM candidato paso a paso.

Algoritmo de Bellman-Ford con detección de ciclos negativos

```
BellmanFord(G, c, v):
  Crear un diccionario D^0 con D^0[v] = 0 y D^0[y] = ∞, y ≠ v
  Para i = 1, ..., n − 1:
    Crear un diccionario D^i[y] = mín{D^{i−1}[x] + c(xy) | x ∈ N^in[y]} para todo y ∈ V(G).
  Si D^{n−1}[y] ≤ D^{n−1}[x] + c(xy) para todo xy ∈ E(G): retornar D^{n−1}
  Caso contrario, informar que G tiene ciclos de peso negativo
```

3.3. Bellman-Ford, versiones > 1.0: posibles mejoras

- Como se discute en la demostración del Teorema 5, si $D^i = D^{i-1}$, entonces $D^{n-1} = D^{i-1}$ porque c es invariante.

- Esto significa que podemos interrumpir el algoritmo ni bien detectamos que $D^i = D^{i-1}$, ahorrando iteraciones en algunos casos prácticos.

- Por otra parte, notemos que para calcular $\hat{\varepsilon}^i(v, y)$ utilizamos **exclusivamente** a $\hat{\varepsilon}^{i-1}(v, x)$. La razón es semántica: pedimos que $\hat{\varepsilon}^i(v, y)$ fuera la distancia con **a lo sumo** i **aristas**. En consecuencia, si $\hat{\varepsilon}^i(v, x)$ se decrementa porque existe un camino P de longitud i con menor peso, en lugar de considerar que podemos extender P, nos restringimos a considerar el camino que conociamos de antemano cuyo peso es $\hat{\varepsilon}^{i-1}(v, x) > \hat{\varepsilon}^i(v, x)$. Pero esto es desafortunado, porque en la iteración $i + 1$ vamos a tener que considerar P y, con alta probabilidad, nos va a obligar a cambiar $\hat{\delta}(v, y)$ nuevamente.

- Esto ocurre en el ejemplo de la ejecución de `BellmanFord`. Si el vértice de arriba a la izquierda se considera antes que el de arriba a la derecha, entonces en la primer iteración ya sabríamos que podemos actualizar su valor a 4.

- Para formalizar esta idea, notemos que los valores de D^i se calculan en algún orden de los vértices, digamos $v_1, \ldots, v_n$.

- Luego, para actualizar $D^i[v_j]$ usamos $D^i[v_k]$ cuando $k < j$ y $D^{i-1}[v_k]$ cuando $k > j$, para todo $v_k \in N^{\text{in}}[v_j]$.

- Aunque parezca complicado, la implementación se simplifica, ya que alcanza con ir "pisando" los valores en D.

- En cuanto a la semántica de lo que D calcula, esta sí queda más fea, ya que $D^i[y]$ es la distancia de un camino de v a y tal que $D^i[y] \leq \hat{\varepsilon}^i(v, y)$ para todo $y \in V(G)$. Es decir, no tenemos el mínimo de los caminos de v a y de longitud a lo sumo i, sino algo más chico (que por lo tanto es mejor, porque acelera el algoritmo).

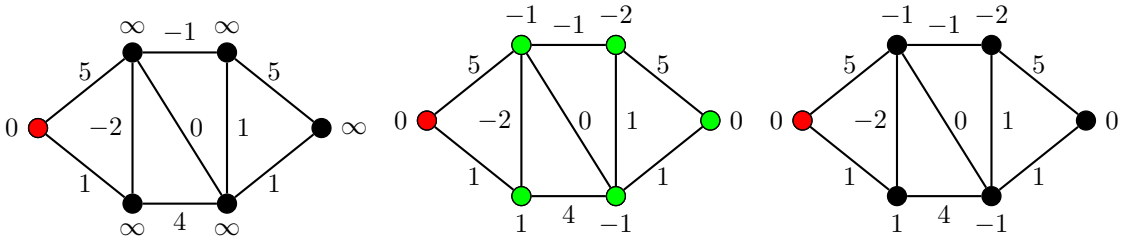
- El algoritmo resultante con estas mejoras queda implementado de la siguiente forma.

Algoritmo de Bellman-Ford mejorado

```
BellmanFord(G, c, v):
  Crear un diccionario D con D[v] = 0 y D[y] = ∞, y ≠ v
  Poner modificado = true
  Para i = 1, ..., n mientras modificado:
    Poner modificado = false
    Para cada arista xy ∈ E(G) tal que D[x] + c(xy) < D[y]
      Poner D[y] = D[x] + c(xy) y modificado = true
  Si no modificado: retornar D
  Caso contrario, informar que G tiene ciclos de peso negativo
```

Ejecución de Bellman-Ford mejorado

Se considera que los vértices se procesan en el orden de izquierda a derecha y de abajo a arriba. Este es el mejor caso; otros ordenamientos requieren más iteraciones.



- Como tercer mejora, notemos que podemos calcular el v-ACM directamente.
- Cuando actualizamos $D[y]$ con $D[x] + c(xy)$ es porque detectamos un mejor camino para llegar a y que lo hace a través de x.
- En consecuencia, podemos definir que x es el padre de y.
- Ejercicio: demostrar que si G no tiene ciclos de peso negativo, entonces el grafo T que se computa de esta forma es un bosque.
- Ejercicio 2: demostrar que G tiene un ciclo de peso negativo si T no es un bosque, mostrando cómo computar un ciclo de peso negativo en $O(n)$ tiempo.
- Finalmente, observemos que no se produce ningún efecto en $D[y]$ cuando $D[x]$ no se modificó en la iteración anterior para todo $x \in N^{\text{in}}[y]$. En consecuencia, es posible saltearse estos vértices.
- Una forma simple de implementar esta idea es manteniendo un conjunto de pares (x, c) donde x es un vértice modificado y c es un valor.
- En cada paso se saca un par (x, c) y se lo procesa sólo si $D[x] = c$. Para procesarlo, se actualiza cada $y \in N^{\text{out}}[x]$ tal que $D[x] + c(xy) < D[y]$. Para cada uno de ellos, se guarda el nuevo par $(y, D[y])$ en el conjunto. La razón de incluir el costo es para evitar procesar aquellos pares (y, c) donde c ya es un valor obsoleto.
- La implementación de este algoritmo la discutimos en la materia Problemas, Algoritmos y Programación; en esta materia queda de ejercicio.

3.4. Sistemas de restricciones de diferencias (20 mins)

- En esta sección vemos una aplicación del problema de camino mínimo a la resolución de sistemas de restricciones de diferencias.
- Un sistema de restricciones de diferencias es un sistema $\mathcal{S}$ con n indeterminadas $x_1, \ldots, x_n$ y m desigualdades de la forma: $x_j - x_i \leq c_{ij}$ para una constante c_{ij} cualquiera. (Notemos que hay a lo sumo una ecuación por cada par ordenado i, j).

Ejemplos de sistemas de restricciones de diferencias

$$\mathcal{S}_1 = \begin{cases} x_2 - x_1 \leq 1 \\ x_3 - x_2 \leq 3 \\ x_1 - x_3 \leq 4 \\ x_4 - x_1 \leq 3 \\ x_1 - x_4 \leq -2 \\ x_3 - x_4 \leq -2 \\ x_4 - x_3 \leq 3 \end{cases} \qquad \mathcal{S}_2 = \begin{cases} x_2 - x_1 \leq 1 \\ x_3 - x_2 \leq -3 \\ x_1 - x_3 \leq 4 \\ x_4 - x_1 \leq 3 \\ x_1 - x_4 \leq -2 \\ x_3 - x_4 \leq -2 \\ x_4 - x_3 \leq 3 \end{cases}$$

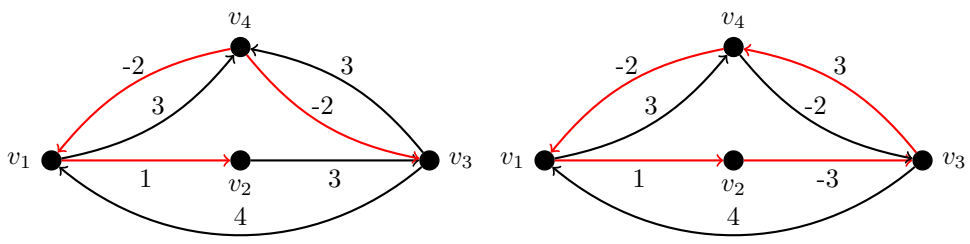
- Para modelar este problema, podemos construir un digrafo $G(\mathcal{S})$ con un vértices v_i por cada $1 \leq i \leq n$ y un vértice especial v_0.
- Desde v_0 hay una arista a cada vértice v_i de costo 0.
- La idea es que v_i represente a x_i de forma tal que, si una solución existe, entonces $x_i = d(v_0, v_i)$ es una solución factible.
- Para que esto funcione, teniendo en cuenta la ecuación $x_j - x_i \leq c_{ij}$ y el hecho de que $x_i = d(v_0, v_i)$ y $x_j = d(v_0, v_i)$, necesitamos que $d(v_0, v_j) \leq d(v_0, v_i) + c_{ij}$.
- Esta condición la podemos modelar agregando una arista $v_i v_j$ de peso c_{ij}, luego el camino hasta v_i de peso $d(v_0, v_i)$ se puede extender a un camino de peso $d(v_0, v_i) + c_{ij}$ hasta v_j.
- En resumen, $G(\mathcal{S})$ tiene $n+1$ vértices $v_0, \ldots, v_n$ donde hay una arista $v_0 v_i$ de costo 0 para todo $1 \leq i \leq n$ y una arista $v_i v_j$ de costo c_{ij} por cada desigualdad $x_j - x_i \leq c_{ij}$.

Digrafos de restricciones

A la izquierda y derecha se muestran los digrafos $G(\mathcal{S}_1) - v_0$ y $G(\mathcal{S}_2) - v_0$, respectivamente. Para $\mathcal{S}_1$ obtenemos una solución si definimos $x_1 = \delta(v_0, v_1) = -2$, $x_2 = \delta(v_0, v_2) = -1$, $x_3 = \delta(v_0, v_3) - 2$, $x_4 = \delta(v_0, v_4) = 0$. Para $\mathcal{S}_2$ esto no funciona: las distancias no definen una solución. Sin embargo, hay un ciclo v_1, v_2, v_3, v_4, v_1 de costo negativo en cuyas aristas corresponden al subsistema:

$$\begin{cases} x_2 - x_1 \leq 1 \\ x_3 - x_2 \leq -3 \\ x_4 - x_3 \leq 3 \\ x_1 - x_4 \leq -2 \end{cases}$$

Como la suma de estas inecuaciones genera la inecuación $0 \leq -1$, el sistema no tiene soluciones.



Teorema 6. *Sea $\mathcal{S}$ un sistema de restricciones de diferencias y $G = G(\mathcal{S})$ su digrafo asociado. Luego, $\mathcal{S}$ tiene una solución si y sólo si todo ciclo de G tiene peso no negativo. Más aún, si G no tiene ciclos de peso negativo, entonces $x_i = d(v_0, v_i)$ es una solución a $\mathcal{S}$*

Demostración. Supongamos primero que $\mathcal{S}$ tiene solución. Notemos que v_0 no pertenece a ningún ciclo, porque $d^{\text{in}}(v_0) = 0$. Luego, por definición, si $C = w_1, \ldots, w_{k+1}$ es un ciclo de G (con $w_{k+1} = w_1$), entonces la arista $w_i w_{i+1}$ corresponde a una ecuación de la forma $x(w_{i+1}) - x(w_i) \leq c(w_i w_{i+1})$. Cómo $\mathcal{S}$ tiene solución, obtenemos que

$$0 = \sum_{i=1}^{k} (x(w_{i+1}) - x(w_i)) \leq \sum_{i=1}^{k} c(w_i w_{i+1}) = c_+(C).$$

Para la vuelta, supongamos que G no tiene ciclos de peso negativo. Queremos ver que la asignación $x_i = \delta(v_0, v_i)$ es una solución factible de $\mathcal{S}$. Para ello, consideremos cualquier ecuación $x_j - x_i \leq c_{ij}$. Por definición, hay un camino P_i de v_0 a v_i con $c_+(P_i) = \delta(v_0, v_i) = x_i$. Luego, como el camino $P_i + v_j$ (que puede no ser simple) une v_0 con v_j y G no tiene ciclos de peso negativo, el Teorema 2 garantiza que $x_j = \delta(v_0, v_j) = \hat{\delta}(v_0, v_j) \leq c_+(P_i + v_j) = \delta(v_0, v_i) + c_{ij} = x_i + c_j$. □

Corolario 2. *Cualquier sistema de restricciones de diferencias con n indeterminadas y m ecuaciones se puede resolver en tiempo $T(n + m) = O(nm)$ y espacio $O(n + m)$.*

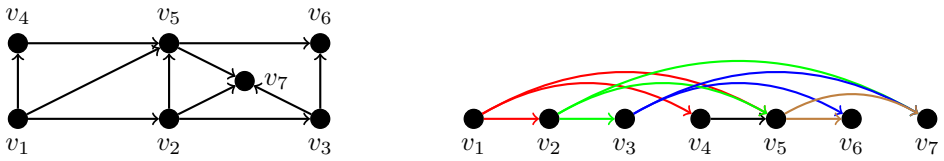
Intervalo (10 mins)

4. Camino mínimo en un DAG (30 mins)

- En esta sección consideramos el problema de camino mínimo en un DAG.
- Obviamente, este caso se puede resolver con el algoritmo de Bellman-Ford en tiempo $O(nm)$, pero pretendemos resolverlo en tiempo lineal.
- Nuevamente, la idea es diseñar un algoritmo de programación dinámica.
- Para mejorar la complejidad de Bellman-Ford, vamos a considerar un orden particular para computar las distancias.
- Decimos que un ordenamiento $S = v_1, \ldots, v_n$ de $V(G)$ es un *orden topológico* de un digrafo G cuando $j > i$ para toda arista $v_i v_j$ de G.

Orden topológico

El orden $v_1, \ldots, v_7$ es topológico, lo que se observa fácilmente dibujando los vértices de izquierda a derecha de acuerdo al orden.



- Obviamente, si G admite un orden topológico, entonces G es acíclico.
- La vuelta también vale por el siguiente teorema que caracteriza a los DAGs.

Teorema 7. *Un digrafo admite un orden topológico si y sólo si es un DAG.*

Demostración. La ida es trivial, porque si $w_1, \ldots, w_k$ es un camino de G, entonces w_i aparece antes que w_{i+1} en el orden topológico para todo $1 \leq i < k$. Entonces $w_k \neq w_1$.

La vuelta sale por inducción en n, cuyo caso base $n = 1$ es trivial. Para el paso inductivo, recordemos que G tiene un sumidero v_n por Lema 1. Ciertamente, $G - v_n$ es un DAG que, por hipótesis inductiva, admite un orden topológico $v_1, \ldots, v_{n-1}$. Como $d^{\text{out}}(v_n) = 0$, todas las aristas que inciden en v_n son de la forma $v_j v_n$ para algún j. En consecuencia, $v_1, \ldots, v_n$ es un orden topológico de G. □

- Consideremos un orden topológico $v_1, \ldots, v_n$ de un digrafo G; notar que si G tiene un origen, éste debe ser v_1.
- Por definición, si P es un camino de v_1 a v_k entonces P es un camino de $G[\{v_1, \ldots, v_k\}]$, ya que no tenemos aristas desde $v_{k+1}, \ldots, v_n$ hacia $v_1, \ldots, v_k$.
- Por lo tanto, si definimos $\sigma(v_1, v_k)$ como el mínimo de los pesos de los caminos de v_1 a v_k que usan sólo vértices de $G[\{v_1, \ldots, v_k\}]$, obtenemos que $\delta(v_1, v_k) = \sigma(v_1, v_k)$.[8]
- Aplicando el argumento recursivamente, si la ultima arista de P es $v_j v_k$, entonces $P - v_k$ es un camino de $G[\{v_1, \ldots, v_j\}]$.
- Luego, $\delta(v_1, v_k) = \sigma(v_1, v_k) = \sigma(v_1, v_j) + c(v_j v_k) = \delta(v_1, v_j) + c(v_j v_k)$ para algún $j \in N^{\text{in}}(v_k)$.
- Obviamente, v_j es el vértice que minimiza la expresión.
- Luego, podemos calcular δ desde el origen $v = v_1$ con la siguiente formula recursiva.
- Es importante notar que no necesitamos saber cuál es el orden topológico, sólo nos alcanza con saber que existe.

$$\delta(v, y) = \begin{cases} 0 & \text{si } y = v \\ \min\{\delta(v, x) + c(xy) \mid x \in N^{\text{in}}(y)\} & \text{caso contrario} \end{cases}$$

- Ciertamente esta fórmula admite n instancias, una por cada vértice y.
- Calcular la instancia correspondiente a y requiere un recorrido de $N^{\text{in}}(y)$.
- Luego, con memoización, el algoritmo top-down mostrado abajo requiere $O(n + m)$ tiempo.
- Recordar que se puede "reconstruir" la solución en $O(n + m)$ tiempo que, en este caso, es el v-ACM de G (ver Apéndice A.3).
- Remarcamos que el algoritmo es robusto y funciona también cuando G tiene muchas fuentes; en este caso, $D[w]$ es la distancia mínima desde alguna de las fuentes hacia v.

Camino mínimo en un DAG (top-down)

```
1 spdag(G, c):
2   Crear un vector D con D[v] = ∞ para todo v ∈ V(G).
3   Para cada w = 1,...,n, poner D[w] = spdag(G, c, D, w)
4   Retornar D
5
6 spdag(G, c, D, w):
```

[8] σ es por distancia inducida en un subconjunto (σubset).

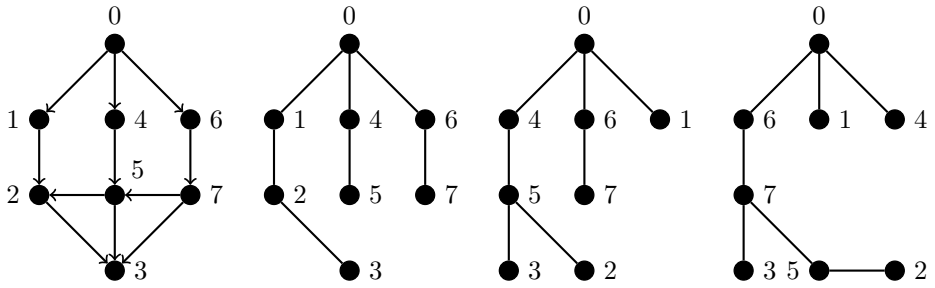
```
7    Si d^in(w) = 0 retornar 0
8    Si D[w] = ∞:
9      Poner D[w] = mín{spdag(G, c, D, w) + c(uw) | u ∈ N^in(w)}
10   Retornar D[w]
```

4.1. Cómputo de un orden topológico

- En caso de querer resolver el problema en forma bottom-up, necesitamos memoizar en el orden correcto.
- Esto es, como necesitamos conocer $D[v_i]$ para calcular $D[v_j]$ si $i < j$ cuando $v_1, \ldots, v_n$ es un orden topológico, tenemos que calcular $D[v_i]$ antes que $D[v_j]$.
- Si no tenemos un orden topológico no podemos implementar el algoritmo bottom-up.
- Por Teorema 7, podemos querer determinar si un digrafo G tiene un orden topológico para reconocer si G es un DAG. De esta forma, podemos aplicar algoritmos especializados sobre el orden topológico (e.g., camino mínimo).
- En caso de existir, el orden topológico de G se puede computar en $O(n + m)$ aplicando el algoritmo que surge de la demostración del Teorema 7: mientras G tiene algún vértice w con $d^{\text{out}}(w) = 0$, sacarlo de G y agregarlo al final del orden correspondiente a $G - w$. Se omiten los detalles.
- Otra forma de calcular un orden topológico es con DFS.
- Supongamos que T es un árbol DFS de G enraizado en la fuente v. Ordenemos cada conjunto de vértices hermanos de acuerdo al orden en que son descubiertos por DFS. Por ejemplo, si v tiene dos hijos u, w y u se descubre antes que w, entonces u está a la izquierda de w.
- Ejercicio: demostrar que G es un DAG si y sólo si el reverso del postorder de T es un orden topológico de G.

DFS y orden topológico

Árboles DFS respetando el orden izquierda a derecha los hijos de cada nodo, con sus correspondientes postordenes.



postorder: 3,2,1,5,4,7,6,0 3,2,5,4,7,6,1,0 3,2,5,7,6,1,4,0

- Más allá de la dificultad de la demostración, el algoritmo DFS que calcula el orden es simple.
- A diferencia del orden DFS donde agregamos los vértices ni bien se descubre, acá seguimos el postorden. Es decir, agregamos v al orden topológico una vez encontrados sus hijos.

Ordenamiento topológico usando DFS

```
toposort(G):
  Desmarcar todos los vértices de G
  Crear S = ∅
  Para cada v ∈ V(G): toposort(G, S, v)
  Retornar S

toposort(G, S, v):
  Si v está marcado, retornar
  Marcar v //acá se agregaría en el orden DFS
  Para cada w ∈ N^out(v): toposort(G, S, w)
  Agregar v al inicio de S.  //acá se agrega en el toposort dfs: postorden y reverso
```

- Vale destacar que el algoritmo DFS siempre retorna un orden, independientemente de si G es acíclico.
- Para determinar si G es aciclíco, hay que verificar que $i < j$ para toda arista $v_i v_j$; esto se puede hacer en tiempo $O(n + m)$.
- Ejercicio: dar un algoritmo eficiente para encontrar un ciclo a partir del orden candidato cuando G no es un DAG.

Teorema 8. *El problema de reconocer si un digrafo G es acíclico se puede resolver en tiempo $O(n + m)$. Más aún, existe un algoritmo de tiempo $O(n + m)$ que retorna un orden topológico o un ciclo de G.*

4.2. Gestión de proyectos

- Una aplicación típica de camino mínimo en un DAG es la gestión de proyectos.
- Supongamos que tenemos n actividades $a_1, \ldots, a_n$.
- Cada actividad a_i tiene un costo (usualmente tiempo) $c(a_i)$.
- En los proyectos suelen haber ciertas precedencias en el orden en que se pueden realizar las actividades.
- Estas precedencias son de la forma: no se puede comenzar a_j hasta que a_i no termine.
- Podemos definir un digrafo G de precedencias con n vértices $a_1, \ldots, a_n$ donde $a_i a_j$ es una arista cuando a_i tiene que terminar antes que se puede comenzar a_j.
- Es de esperar que G sea acíclico; sino el proyecto es inviable.
- Sin perdida de generalidad, podemos suponer que existe una actividad inicial, digamos a_1, y otra final, digamos a_n, ambas de costo 0, de forma tal que a_1 precede a todas las actividades y todas las actividades preceden a a_n. (No hace falta que la precedencia sea directa.)
- Si definimos el costo de cada arista $a_i a_j$ como $c(a_i a_j) = c(a_i)$, entonces el costo del camino **máximo** desde a_1 hasta a_i nos indica cuándo es lo mas temprano que podemos iniciar a_i.
- Por otra parte, el costo del camino máximo de v_1 a v_n nos indica cuánto dura el proyecto como mínimo.
- Finalmente, los caminos de costo máximo son *caminos crítico*, en el sentido de que un atraso en cualquiera de sus actividades impacta en un atraso global del proyecto.
- Este método es la base de lo que se conoce como CPM por sus siglas en inglés (*critical path method*) y se suele usar junto con PERT (*program evaluation and review technique*) que también aplica el algoritmo de camino mínimo en un DAG donde los tiempos se miden en las transiciones en lugar de las actividades.

- Las herramientas de gestión de proyectos suelen implementar toposort y algoritmos de camino mínimo en DAG para auxiliar en la toma de decisiones.

- Desde su creación, CPM y PERT crecieron como hasta llegar a ser una metodología de administración de proyectos con una bibliografía propia (que no incluyo porque desconozco; una idea rápida surge de buscar en Internet).

Referencias

[1] Richard Bellman. On a routing problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, third edition, 2009.

[3] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.

[4] Lester R. Jr. Ford. Network flow theory. paper p-923. Technical report, Santa Monica, California: RAND Corporation, 1956.

[5] Edward F. Moore. The shortest path through a maze. In *Proc. Internat. Sympos. Switching Theory 1957, Part II*, pages 285–292. Harvard Univ. Press, Cambridge, Mass., 1959.

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.[9]

A.1. Algoritmo de Dijkstra

La implementación de Dijkstra en tiempo $O(m \log n)$ es una copia de la implementación de Prim vista en la clase pasada. Las tres diferencias son: ahora el grafo de entrada es dirigido; en lugar de ordenar la cola por valor de $c(xw)$, se guarda por $c + c(xw)$ donde c es el costo del camino del origen a x; guardamos $\delta(r, x)$ en un diccionario D para todo $x \in V(G)$.

Implementación de Dijkstra

```
using graph = vector<vector<neigh>>;
using neigh = pair<int, int>;          //vecino, costo
using bridge = tuple<int,int,int>;     //costo, v, w

const int none = -1;

int main() {
  //lectura de digrafo de lista de aristas a lista adyacencias

  //algoritmo de dijkstra
  vector<int> T(n, none), D(n); T[r] = r; D[r] = 0;
  priority_queue<bridge> S;
  for(auto x : G[r]) S.push({-costo(x), r, vecino(x)});
  while(not S.empty()) {
    int c, v, w;
    tie(c,v,w) = S.top();
```

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

```
    S.pop();
    if(T[w] == none) {
      T[w] = v; D[w] = -c;
      for(auto x : G[w]) if(T[vecino(x)] == none)
        S.push({c-costo(x), w, vecino(x)});
    }
  }

  //output del algoritmo: T + D
}
```

A.2. Algoritmo de Bellman-Ford

El algoritmo de Bellman-Ford implementado en esta sección es casi una traducción a C++ del que se describe en la Sección 3.3. Las dos diferencias son que: construimos el r-ACM de (G, c) en un `vector<int> T`; y mantenemos un `vector<int> M` cuya posición i indica si el i-ésimo vértice fue modificado y no procesado en la iteración anterior. Esto se puede mejorar aún más, manteniendo sólo los vértices modificados a procesar en lugar de recorrerlos todos en cada iteración.

Implementación de Bellman-Ford (bellman-ford.cpp)

```
using neigh = pair<int, int>;          ////vecino, costo
using graph = vector<vector<neigh>>;

const int infty = numeric_limits<int>::max() / 2 - 1;

int main() {
  //lectura de digrafo de lista de aristas a lista adyacencias

  //algoritmo de Bellman-Ford
  vector<int> D(n, infty), T(n, none), M(n, false);
  D[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(D[v] + cost(e) < D[to(e)])
      {
        M[to(e)] = changed = true;
        D[to(e)] = D[v] + cost(e);
        T[to(e)] = v;
      }
    }
  }

  if(changed) cout << "Ciclo negativo detectado" << endl;
  else //output del algoritmo: T + D
}
```

A.3. Camino mínimo en un DAG top-down

El algoritmo de camino mínimo en un DAG en versión top-down sigue la estrategia básica explicada en la Clase 2. La reconstrucción de la solución (i.e., el camino de la raíz a un vértice v) se implementa en `prev`.

Camino mínimo en un DAG

```cpp
//vecino de entrada, costo
using neigh = pair<int, int>;
using graph = vector<vector<neigh>>;

vector<int> D;  //estructura de memoización: DPM

int spdag(const graph& G, int v) {
  if(G[v].empty()) return 0;
  if(D[v] == infty) for(auto e : G[v])
    D[v] = min(D[v], spdag(G, from(e)) + cost(e));
  return D[v];
}

int prev(const graph& G, int v) {
  if(not G[v].empty()) for(auto e : G[v])
    if(spdag(G,v) == spdag(G, from(e)) + cost(e)) return from(e);
  return -1;
}

//precondicion: G es un dag
int main() {
  //lectura de DAG de lista de aristas a adyacencias guardando |N^in| en lugar de |N^out|

  //Algoritmo de camino mínimo en DAG
  D.assign(n, infty);
  for(int v = 0; v < n; ++v) D[v] = spdag(G, v);

  //output del algoritmo: T[v] = prev(G,v) y D[v] = spdag(G,v)
}
```

A.4. Orden topológico y reconocimiento de DAGs (sin DFS)

En esta sección implementamos la demostración del Teorema 7 para construir un orden topológico. La idea es ir removiendo las fuentes hasta quedarnos sin vértices. Pero, en lugar de una remoción física, simplemente actualizamos el grado de entrada de cada vértice es un `vector<int> indeg` cuyo computo inicial no se muestra. Si con este proceso no logramos remover todos los vértices, entonces hay un ciclo en el grafo resultante. Por simplicidad, mostramos cómo encontrar un camino no simple en lugar de un ciclo aplicando DFS (`print_circuit`). Para encontrar el ciclo en forma iterativa, se pueden seguir las mismas ideas discutidas en la Clase 4 para `leaf_or_cycle`.

Reconocimiento de DAGs sin usar DFS para el orden topológico (esDAG.cpp)

```cpp
using graph = vector<vector<int>>;

bool print_circuit(const graph& G, int v, const vector<int>& indeg, vector<bool>& mark) {
  if(mark[v]) {cout << v; return true;}
  mark[v] = true;
```

```
  for(auto w : G[v]) if(indeg[w] > 0 and print_circuit(G, w, indeg, mark)) {
    cout << " <- " << v; return true;
  }
  return false;
}

int main() {
  //leer la lista de aristas en adyacencias guardando grado de entrada en indeg

  //computo del orden topologico
  vector<int> F, topo;
  for(int v = 0; v < n; ++v) if(indeg[v] == 0) F.push_back(v);

  while(not F.empty()) {
    topo.push_back(F.back());
    F.pop_back();
    for(auto w : G[topo.back()]) if(--indeg[w] == 0) F.push_back(w);
  }

  //output del algoritmo
  if(topo.size() < n) {
    vector<bool> mark(n, false);
    int v = 0; while(indeg[v] == 0 or G[v].empty()) ++v;
    cout << "Camino no simple detectado: "; print_circuit(G, v, indeg, mark);
  } else {
    cout << "Orden topologico: "; for(auto v : topo) cout << v << " ";
  }
}
```

A.5. Orden topológico y reconocimiento de DAGs (con DFS)

La parte central de este método es el algoritmo DFS que construye el árbol DFS en `vector<int> parent` mientras calcula su postorder en `vector<int> order`. Para poder determinar si una arista esta invertida, se guarda la posición de cada vértice v en el postorder usando el `vector<int> rankk`.

Implementación DFS para encontrar un orden toplógico

```
using graph = vector<vector<int>>;

const int none = -1;
vector<int> order, rankk, parent;

void DFS(const graph& G, int v, int p) {
  if(parent[v] == none) {
    parent[v] = p;
    for(auto w : G[v]) DFS(G, w, v);
    rankk[v] = order.size();
    order.push_back(v);
  }
}

void postorder_DFS(const graph& G) {
  order.clear();
```

```
  rankk.assign(G.size(), none);
  parent.assign(G.size(), none);
  for(int v = 0; v < G.size(); ++v) DFS(G, v, v);
}

int main() {
  //leer de listas de aristas a adyacencias

  //ordenamiento topológico reverso
  postorder_DFS(G);

  //detección de arista invertida
  int v = -1, w;
  for(int x = 0; x < n and v == -1; ++x) for(auto y : G[x])
    if(rankk[x] < rankk[y]) {v = x; w = y; break;}

  //output del algoritmo
  if(v != -1) {
    cout << "Ciclo detectado - Ejercicio: encontrar ciclo.";
    //Ayueda: demostrar que w es un ancestro de v por ser el árbol DFS
  } else {
    cout << "Orden topologico: ";
    for(auto v = order.rbegin(); v != order.rend(); ++v) cout << *v << " ";
  }
}
```