Notas de la clase 5 – árbol generador mínimo

Francisco Soulignac

8 de abril 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. Introduccion (40 mins)

1.1. Árbol generador mínimo

- Problema de árbol generador mínimo (resp. máximo)

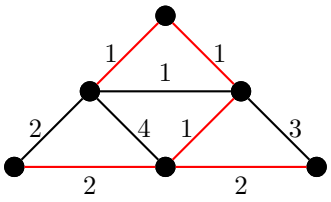
  Input: un grafo conexo $G = (V, E)$ y una función $c\colon E(G) \to \mathbb{R}$

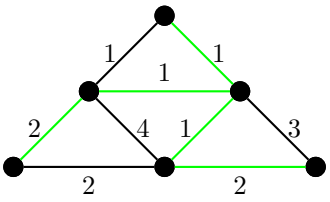
  Output: un árbol generador T de G que minimice (resp. maximice) $c_+(T) = \sum_{vw \in E(T)} c(vw)$.

- A la función c se la llama *costo* o *peso* en problemas de minimización y *beneficio* en problemas de maximización.

Ejemplo del problema de árbol generador mínimo

Los árboles rojo y verde son outputs posibles de la instancia (G, c), donde c se dibuja como etiquetas de las aristas.





- Ejemplos de problemas

  ▷ Queremos unir un conjunto de localidades $V = \{v_1, \ldots, v_n\}$ construyendo autopistas de forma tal que se pueda viajar entre cualquier par de localidades sin abandonar la autopista. Las posibles autopistas a construir están dadas por el conjunto de aristas E del grafo $G = (V, E)$, y cada arista vw tiene un costo de construcción $c(vw)$. El objetivo es minimizar el costo total de la construcción de las autopistas.

  ▷ Queremos unir un conjunto de chips usando cables, a fin de minimizar la cantidad de cable utilizada.

  ▷ En general, el diseño de cualquier red donde conectar las entidades tiene un costo.

▷ Se suele usar como procedimiento en varios problemas más sofisticados.

- El output del problema se puede codificar con la misma codificación que usamos para árboles enraizados que vimos la clase pasada. La raíz seleccionada es superflua.

- Decimos que en T es un *árbol generador mínimo* (resp. *máximo*) de (G, c) cuando $c_+(T) \leq c_+(T')$ (resp. $c_+(T) \geq c_+(T')$) para todo árbol generador T de G.

- Cuando usamos el acrónimo AGM en la frase "T es un AGM", lo hacemos para referirnos a un árbol generador **mínimo**. Por ejemplo, los árboles rojo y verde de la figura anterior son AGMs de (G, c).

- La razón es que ambos problemas son equivalentes y se pueden transformar uno en otro en $O(1)$ tiempo, como surge de la siguiente observación que no requiere demostración.

Observación 1. *Sea G un grafo y $c\colon E(G) \to \mathbb{R}$. Entonces, T es un AGM de (G, c) si y sólo si T es un árbol generador máximo de $(G, -c)$.*

- Existe una variante del problema de AGM para digrafos, pero no la estudiamos en este curso.

1.2. Camino minimax

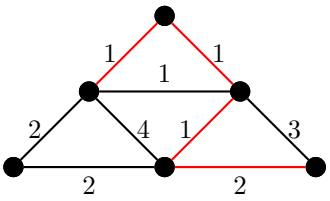
- Problema de camino minimax (resp. maximin) entre dos vértices

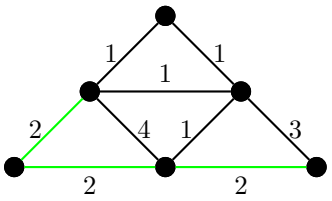
  Input: un grafo conexo $G = (V, E)$, una función $c\colon E(G) \to \mathbb{R}$ y dos vértices v y w.

  Output: un camino P de v a w que minimice (resp. maximice) $c_{\text{máx}}(P) = \text{máx}\{c(vw) \mid vw \in E(P)\}$ (resp. $c_{\text{mín}}(P) = \text{mín}\{c(vw) \mid vw \in E(P)\}$).

Problema del camino minimax

Dos caminos minimax de costo $c_{\text{máx}} = 2$ entre los mismos pares de vértices.





- Ejemplos de problemas

  ▷ Fijado un conjunto de routers $V = \{v_1, \ldots, v_n\}$ conectados en una red de enlaces E, cada uno con un ancho de banda $c(vw)$, queremos encontrar un camino P de un rounter v_i a otro v_j cuyo ancho de banda $c_{\text{mín}}(P)$ sea máximo.

  ▷ En general, en problemas de transporte donde las aristas limitan la carga o flujo a transportar, queremos encontrar un camino que nos permita llevar la mayor cantidad posible de material.

- El output del problema se puede codificar con una secuencia.

- Decimos que P es un *camino minimax* (resp. maximax) de (G, c) entre v y w cuando $c_{\text{máx}}(P) \leq c_{\text{máx}}(P')$ (resp. $c_{\text{mín}}(P) \geq c_{\text{máx}}(P')$) para todo camino P' entre v y w.

- Al igual que en el caso de AGM, ambos problemas son equivalentes y se puede transformar uno en otro en $O(1)$ tiempo.

Observación 2. *Sea G un grafo con dos vértices v y w y $c\colon E(G) \to \mathbb{R}$. Entonces, P es un camino minimax de (G, c) entre v y w si y sólo si P es un camino maximin de $(G, -c)$ entre v y w.*

- Al problema de camino maximin tambien se *camino más ancho*, ya que se trata de aumentar la capacidad de un flujo que se puede transmitir por el camino P. A la arista vw de P con $c(vw) = c_{\text{mín}}(P)$ se la llama *cuello de botella*.

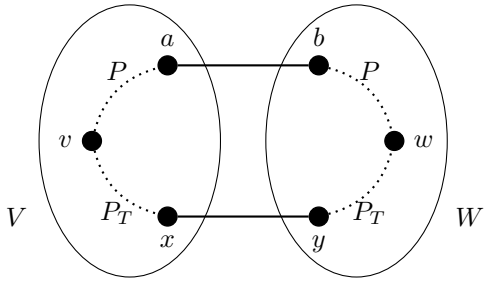
- Existen versiones del problema para digrafos que vemos la clase que viene.

- El siguiente teorema describe la relación entre el problema de AGM y el problema de camino minimax.

Teorema 1. *Sea T un árbol generador de un grafo G y $c\colon E(G) \to \mathbb{R}$. Entonces, T es un AGM de (G, c) si y sólo si todo camino de T es minimax de (G, c).*

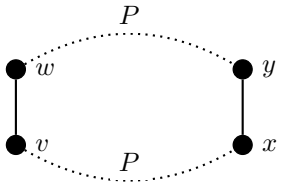
Demostración. Supongamos primero que T es un AGM de (G, c). (Seguir el diagrama de abajo.) Para dos vértices v y w cualesquiera, sea xy la arista del camino P_T de T entre v y w con $c(xy) = c_{\text{máx}}(P_T)$. Claramente, v y w están en componentes distintas de $T - xy$, llamémoslas V y W con $v \in V$ y $w \in W$. Luego, si P es un camino entre v y w, contiene una arista ab que une $a \in V$ y $b \in W$. En consecuencia, $T' = (T - xy) + ab$ es árbol, ya que es conexo y tiene $n - 1$ aristas. Como T es AGM, sabemos que $c_+(T) \leq c_+(T') = c_+(T) - c(xy) + c(ab)$. En consecuencia, $c_{\text{máx}}(P_T) = c(xy) \leq c(ab) \leq c_{\text{máx}}(P)$, lo que significa que todo camino de T es minimax de (G, c).

Los caminos de un AGM son minimax



Supongamos ahora que todo camino de T es minimax de (G, c) y consideremos el AGM T' de (G, c) tal que $|E(T) \cap E(T')|$ es máximo. (Seguir el diagrama de abajo.) Queremos ver que $T = T'$; supongamos, en busca de una contradicción, que T tiene una arista vw que no pertenece a T'. Sea xy la arista con máximo $c(xy) = c_{\text{máx}}(P)$, donde P es el único camino de T' entre v y w. Por hipótesis, $c(vw) \leq c(xy)$ porque vw es un camino de T. Luego, $c_+(U) \leq c_+(T')$ para $U = (T' + vw) - xy$ y, como U es un árbol (porque es acíclico y tiene $n - 1$ aristas, ¿por qué?), U es un AGM. Pero esto es imposible, porque $|E(T) \cap E(U)| = 1 + |E(T) \cap E(T')|$, lo que significa que $T = T'$. Es decir, T es un AGM de (G, c).

Si los caminos son minimax, el árbol es AGM



□

- El Teorema 1 muestra que podemos resolver el problema de camino minimax entre dos vértices v y w en dos pasos: primero se construye un AGM T de G y luego se retorna el camino de T entre v y w.

- Esta estrategia no tiene por qué ser óptima; quizá se puede encontrar un camino minimax entre v y w en forma más eficiente.[1]

- Empero, el Teorema 1 demuestra también que se puede resolver el problema de camino minimax entre todo par de vértices a la vez.

- Problema de camino minimax (maximin) entre todos los pares

  Input: un grafo conexo G y una función $c\colon E(G) \to \mathbb{R}$.

  Output: un árbol T tal que el camino de T entre v y w es minimax (maximim) de (G, c) para todo $v, w \in V(G)$.

- Por Teorema 1, este problema es equivalente al problema de AGM, razón por la cual nadie lo estudia en grafos.

- Sin embargo, es importante saber que ambos problemas son equivalentes y que su equivalente en digrafos es muy estudiado aunque no lo vemos en la materia.

2. Algoritmo genérico para el problema de AGM (30 mins)

- Fijemos un grafo G con una función de costos $c\colon E(G) \to \mathbb{R}$.

- Decimos que un bosque generador F de G es un *AGM parcial* cuando $E(F) \subseteq E(T)$ para algún AGM de (G, c).

AGMs parciales

A la izquierda, un AGM parcial formado por tres árboles de colores azul, rojo y verde. A la derecha, un bosque que no es un AGM parcial porque ningún AGM contiene las tres aristas verdes.

- Decimos que $e \in E(G) \setminus E(F)$ es *segura* cuando sus vértices pertenecen a distintos árboles de F.

- En el grafo de arriba a la izquierda, todas las aristas son segura excepto la gris que une dos vértices verdes.

- Una arista segura vw es *candidata para el árbol* T que contiene a v cuando $c(vw) \leq c(xy)$ para toda arista segura xy con $x \in V(T)$.

- Notar que vw puede o no ser candidata para el árbol T' que contiene a w. En efecto, en la figura de arriba a la izquierda, las dos aristas de peso 2 son candidatas para el árbol azul, pero ninguna de ellas es candidata para los árboles rojo y verde. Por otra parte, la arista de peso 4 no es candidata para ningún árbol, y la aristas aristas de peso 1 son candidatas tanto para el árbol rojo como para el verde.

- En general, una arista vw es *candidata* cuando es candidata para algún árbol. Con lo cual, las aristas de peso 1 y 2 son candidatas.

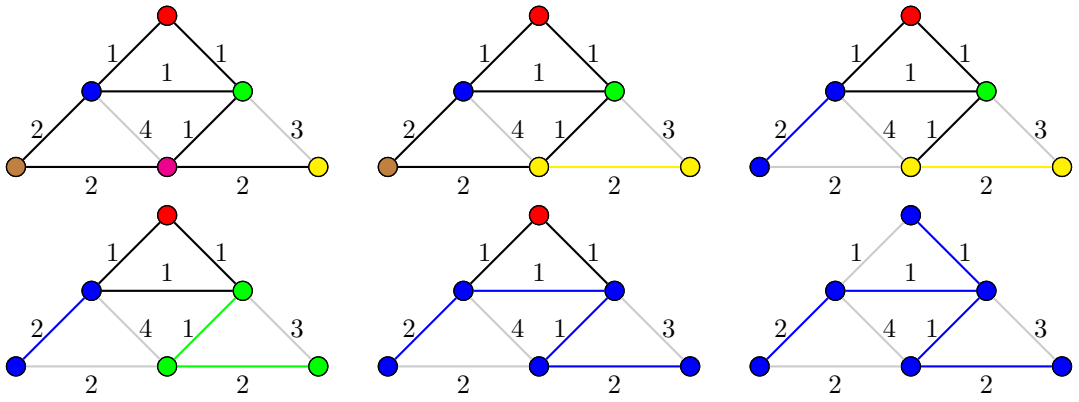
[1] De hecho no es una estrategia óptima. Esto no nos importa en la materia, ya que el problema que tenemos que estudiar es AGM y camino minimax viene como curiosidad.

- Observar que el hecho de ser candidato depende de G y c.
- El siguiente algoritmo goloso computa un AGM de (G, c).

Esquema de cómputo de un AGM

```
AGM(G, c):
  Sea F = (V(G), ∅) un bosque generador de G
  Para i = 1, ..., n − 1:
    Agregar a F una arista candidata de (G, c).
```

Una ejecución arbitraria de `AGM(`G`, `c`)`



Teorema 2. *Si G es un grafo conexo y c es una función de costos, entonces `AGM(`G`, `c`)` computa un AGM de (G, c).*

Demostración. Llamemos F_i al grafo F luego de la i-ésima iteración. Vamos a demostrar, por inducción, que F_i es un AGM parcial. Luego, como F_{n-1} tiene tantas aristas como el AGM T que lo contiene, obtenemos que $F_{n-1} = T$ es un AGM de (G, c). El caso base $i = 0$ es trivial, ya que $\emptyset \subseteq E(T)$ para todo AGM T de G. Para el paso inductivo $i + 1$, llamemos vw a la arista candidata que se agrega a F_i para obtener F_{i+1}. Sin perdida de generalidad, supongamos que vw es candidata para el árbol U de F_i que contiene a v. Notemos que F_{i+1} es un bosque generador de G porque vw, al ser segura, une dos árboles distintos de F_i. Por hipótesis inductiva, $E(F_i) \subseteq E(T)$ para un AGM T de (G, c). Si $vw \in E(T)$, entonces $E(F_{i+1}) \subseteq E(T)$ tal como deseamos. Supongamos, pues, que $vw \notin E(T)$. En este caso, $T + vw$ contiene un ciclo. Luego, el camino entre v y w en T tiene una arista que no pertenece a F_{i+1}. Más aún, alguna de estas aristas xy incide en un vértice $x \in V(U)$ porque $v \in U$. Notemos que y no pertenece a U porque T es acíclico. En consecuencia, xy es segura para F_i, lo que implica que $c(vw) \leq c(xy)$ porque vw es candidata para U. En consecuencia, $T' = (T - xy) + vw$ es un AGM de G porque $c(T') = c(T) - c(xy) + c(vw) \leq c(T)$. Finalmente, observemos que $E(F_{i+1}) \subseteq E(T')$ porque $E(F_i) \subseteq E(T) \setminus \{xy\}$, i.e., F_{i+1} es un AGM parcial. □

- El árbol final y los bosques intermedios dependen de qué candidato se elija en cada paso
- La primer implementación de este algoritmo, que fue dada Borůvka [1] en 1926 (c.f. [6]) y redescubierta muchas veces, funciona de la siguiente forma:

  Borůvka: primero se calcula una arista candidata para cada árbol. Luego, se agregan todas a la vez. Requiere que todos los costos sean distintos y funciona en $O(\log n)$ iteraciones (la cantidad de árboles se reduce a la mitad en cada iteración).

- Nosotros vamos a estudiar dos implementaciones clásicas del algoritmo `AGM`:

  Kruskal: en cada iteración se elije la arista segura de menor peso, que claramente es candidata.

  Prim: se elije un vértice v y en cada iteración se elige una candidata del árbol que contiene a v.

- Todos estos algoritmos son correctos por Teorema 2.

3. Implementación del algoritmo de Prim (30 mins)

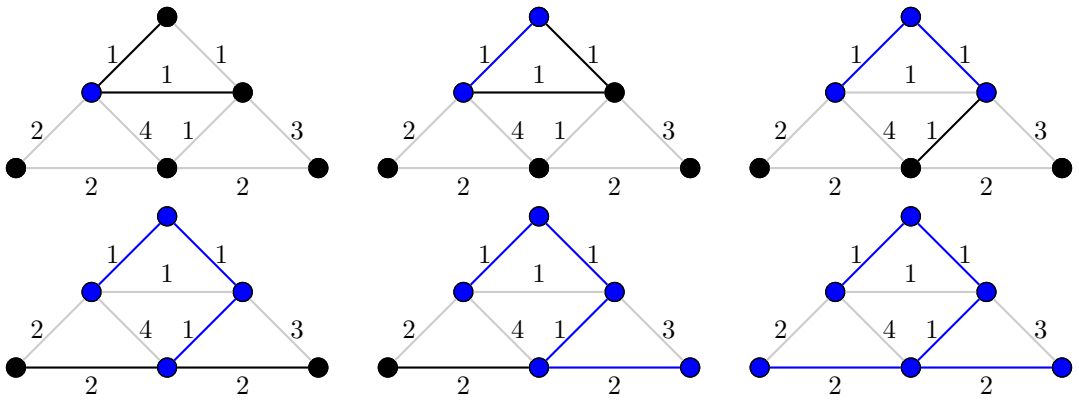
- Publicado por Prim [7] en 1957, este algoritmo fue publicado también por Dijkstra [3] en 1959 (en conjunto con su variante para camino mínimo), y también es atribuido a Jarník [4] quien lo publicó en 1930 en Checo.

- Dado un vértice inicial r, el algoritmo construye un AGM de (G, c) enraizado en r de forma iterativa. Empezando con el árbol $T_0 = (\{r\}, \emptyset)$, en el $(i+1)$-ésimo paso se construye T_{i+1} agregando una arista candidata de T_i.

- Recordemos que vw es candidata para T_i cuando $c(vw)$ es mínima entre las aristas con $v \in V(T_i)$ y $w \notin V(T_i)$.

Ejecución del algoritmo de Prim

Las aristas candidatas seleccionables se muestran en negro y el resto en gris.



3.1. Caso ralo

- Sea $S_i = \{(v, w) \mid v \in V(T_i), w \in V(G) \setminus V(T_i)\}$ el conjunto de pares ordenados que representan los aristas **seguras** que tienen el primer valor en T_i.

- Por definición, los **candidatos** de T_i son aquellas aristas vw con $(v, w) \in S_i$ que tienen $c(vw)$ mínimo. Para construir T_{i+1}, alcanza con agregar cualquiera de ellas a T_i.

- Luego de agregar vw a T_i, tenemos que actualizar S_i en S_{i+1}. Es fácil ver que $S_{i+1} = S_i \setminus S_w \cup \overline{S_w}$, donde S_w son los pares $(x, w) \in S_i$ y $\overline{S_w}$ son los pares (w, y) con $y \notin V(T)$.

- Teniendo en cuenta estas observaciones y las operaciones que se hacen con S, tenemos el siguiente algoritmo de Prim.

Algoritmo de Prim

```
1  Prim(G, c, r):
2    Sea T un vector con T[r] = r y T[w] = ⊥ para w ∈ V(T) \ {r}.
3    Sea S una cola de prioridad por c, conteniendo {(r, w) | w ∈ N(r)}
4    Para i = 1, ..., n − 1:
5      Sea (v, w) el tope de S.
6      Poner T[w] = v
7      Para cada z ∈ N(w): //notar que v ∈ N(w)
8        Si T[z] = ⊥, enconlar (w, z) a S
9        Caso contrario, remover (z, w) de S.
10   retornar T
```

- Notemos que S contiene $O(m) = O(n^2)$ aristas en cada paso.

- En consecuencia, si S se implementa de forma tal que tope, encolar y remover cuestan $O(\log |S|)$, entonces el algoritmo consume $T(n + m) = O(m \log n^2) = O(m \log n)$ tiempo.

- Por AED2, conocemos al menos dos formas de implementar S: 1. árbol binario de búsqueda balanceado (AVL, red-black, etc.), 2. heap indexado sobre árbol.

- Podemos simplificar un poco la implementación observando que no es necesario remover físicamente cada par (z, w) de S. En su lugar, podemos notar que w se "marca" con $T[w] \neq \perp$ cuando se elimina de S. Esto tiene sentido, porque significa que los pares (x, w) dejan de ser seguros. Luego, podemos evitar las remociones de estos pares y si al tomar el tope de S ignoramos los pares (x, w) en los que $T[w] \neq \perp$.

- En esta segunda implementación, que se muestra en el Apéndice A.1, no se utiliza la operación remover. Luego, alcanza con que tope y encolar cuesten $O(\log |S|)$ tiempo para obtener un algoritmo de complejidad temporal $T(n + m) = O(m \log n)$.

- Por AED2, conocemos pues una tercer forma de implementar a S que es más simple: 3. heap sobre vector (o arreglo).

- Esta forma de implementar es la que se suele usar en competencias de programación, dado que C++ provee una cola de prioridad con esta interfaz.

3.2. Caso denso

- Podemos mejorar el algoritmo anterior para el caso en que sabemos que G es un grafo denso.

- La idea es notar que una minoría de las aristas que se guardan en S_i son candidatas y sólo se necesita una de ellas.

- En efecto, si S_i contiene los pares (v, w) y (x, w) con $c(vw) \leq c(xw)$, entonces podemos descartar $c(xw)$, dado que siempre podemos agregar vw a T_i es su lugar.

- En consecuencia, podemos reemplazar S_i por un par de diccionarios P_i y C_i con claves en $V(G) \setminus V(T)$ tales que, para todo $w \in V(G) \setminus V(T)$:

  - $C_i[w] = \text{mín}\{c(vw) \mid v \in V(T)\}$; si w no tiene vecinos en T, entonces $C_i[w] = \infty$.
  - si $C_i[w] < \infty$, entonces $P_i[w] = v$ para algún $v \in V(T)$ con $c(vw) = C_i[w]$.

- En otras palabras, $C_i[w]$ contiene el costo de las aristas seguras de peso mínimo que inciden en w y $P_i[w]$ indica cuál es el otro extremo de una de tales aristas.

- Como antes, alcanza con agregar a T la arista $wP_i[w]$ con $C_i[w]$ mínimo; llamémosla wv.

- Una vez encontrado wv, hay que actualizar C_i y P_i en C_{i+1} y P_{i+1}, respectivamente. Obviamente, un primer paso es remover la clave w de C_i. Luego, las únicas claves de C_i que pueden ser afectadas son aquellas correspondientes a vecinos de $N(w)$. En particular, hay que **decrementar** $C_i[z]$ al valor $c(wz)$ cuando $C_i[z] > c(wz)$ para todo $z \in N(w)$. Para aquellos valores afectados, hay que actualizar $P_i[z]$ en forma acorde.

- El algoritmo resultante se puede implementar de la siguiente forma.

Algoritmo de Prim para grafos densos

```
1  Prim(G, c, r):
2    Sea T un vector con T[r] = r y T[w] = ⊥ para w ∈ V(T) \ {r}.
3    Sea C un diccionario tal que C[w] = c(rw) si w ∈ N(r) y C[w] = ∞ sino.
4    Sea P un diccionario tal que C[w] = r para todo w ∈ N(r).
5    Para i = 1, ..., n − 1:
6      Sea v = P[w] para el vértice w con costo mínimo en C.
7      Poner T[w] = v y eliminar w de C
8      Para cada z ∈ N(w) con T[z] = ⊥:
9        Si c(wz) < C[z], decrementar C[z] a c(wz) y poner P[z] = w.
10   retornar T
```

- El algoritmo anterior cuesta $O(n(\mu + \eta) + m\delta)$ donde μ es el costo de encontrar la clave mínima, η es el costo de eliminar la clave de valor mínimo, y δ es el costo de decrementar una clave.

- Si se usa un árbol binario de búsqueda balanceado para representar a C, entonces el costo queda $O(m \log n)$.

- Esto es lo mismo que para la implementación anterior. La ventaja de esta algoritmo es cuando C se implementa sobre otras estructuras.

- Implementando C como un vector de n posiciones donde la i-ésima posición contiene $C[i]$, el costo queda $O(n^2)$, que es lineal cuando $m = \Omega(n^2)$ (dado que $\mu = O(n)$, $\eta = O(1)$ y $\delta = O(1)$). Esta solución es fácil de implementar (ver Apéndice A.2).

- Finalmente, existe una estructura que no estudiamos en la materia, llamada fibonacci heap (ver [2]), que satisface $\mu = O(1)$, $\eta = O(\log n)$ y $\delta = O(1)$. Con esta estructura, el costo es $O(m + n \log n)$, lo que es lineal para cualquier grafo con $m = \Omega(n \log n)$.

Teorema 3. *Sea G un grafo y c una función de costos. Se puede computar un AGM de (G, c) aplicando Prim en $T(n + m) = O(m + n \log n)$ tiempo.*

4. Algoritmo de Kruskal (45 mins)

4.1. El tipo abstracto disjoint-set

- Antes de ver la implementación del algoritmo de Kruskal, definamos un nuevo tipo de datos llamado *disjoint-set*.

- Cada valor de tipo disjoint-set representa una partición $V_1, \ldots, V_k$ del conjunto $U = \{1, \ldots, n\}$.

- Cuenta con las siguientes operaciones

  Create(n): crea la partición inicial $V_1, \ldots, V_n$ donde $V_i = \{i\}$.

Unite(i, j): une los dos conjuntos de la partición que contienen a i y j en uno sólo.

Find(i): retorna un identificador del conjunto que contiene a i. Por identificador, nos referimos a un objeto comparable por igualdad tal que Find(i) = Find(j) si y sólo si i y j pertenecen al mismo conjunto de la partición.

Ejemplo de ejecución del tipo disjoint-set

Ejecución en un lenguaje ficticio; el identificador de cada conjunto es un miembro arbitrario del mismo.

1. Create(6) → {1}, {2}, {3}, {4}, {5}, {6}.
2. Find(5) → 5
3. Unite(2, 4) → {1}, {2, 4}, {3}, {5}, {6}
4. Unite(1, 3) → {1, 3}, {2, 4}, {5}, {6}
5. Unite(1, 4) → {1, 2, 3, 4}, {5}, {6}
6. Find(1) → 3
7. Find(1) = Find(2) and Find(2) = Find(3) and Find(3) = Find(4)? → true

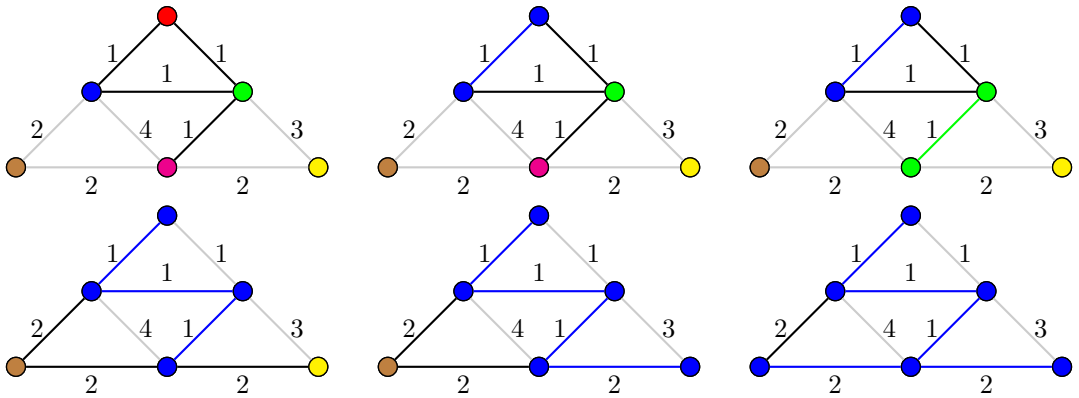
- Es fácil diseñar una estructura para disjoint-set que soporte Create y Unite en $O(n)$ y Find en $O(1)$ tiempo (ejercicio).
- La idea es usar el tipo abstracto por ahora y luego ver cómo implementarlo.

4.2. Implementación de Kruskal

- Publicado por Kruskal [5] en 1956 como simplificación del algoritmo de Borůvka.
- Empezando por el bosque $F_0 = \{V(G), \emptyset\}$, el algoritmo construye F_{i+1} a partir de F_i agregando una arista vw segura de costo mínimo en cada iteración.

Ejecución del algoritmo de Kruskal

Las aristas seleccionables se muestran en negro, el resto en gris.



- El algoritmo se puede implementar en dos pasos: 1. se ordenan todas las aristas de acuerdo a su costo y 2. se recorren las aristas en orden y se insertan aquellas que sean seguras.
- Recordemos que vw es segura si v y w pertenecen a árboles T_v y T_w distintos de F_i.
- Una vez que se agrega vw, los árboles T_v y T_w quedan unidos en un nuevo árbol T_{vw}.
- Aprovechando el tipo disjoint-set, obtenemos la siguiente implementación.
- Notar que el árbol resultante no es enraizado y se implementa como una simple lista de aristas. (La misma se puede enraizar usando BFS.)

Algoritmo de Kruskal

```
1 Kruskal(G, c):
2   Crear una lista de aristas T = ∅
3   Ordenar E(G) ascendentemente de acuerdo a c
4   Crear un disjoint-set U = Create(n)
5   Para cada vw ∈ E(G) en orden:
6     Si U.Find(v) ≠ U.Find(w): //Si vw es segura
7       Agregar vw a T
8       U.Unite(v, w)
9   retornar T
```

- El costo del algoritmo es $T(n+m) = O(s + mf + nu)$ donde:
  - ▷ s es el costo de ordenar las aristas,
  - ▷ f es el costo amortizado de de aplicar Find, y
  - ▷ u es el costo amortizado de aplicar Unite.
- Usando un ordenamiento basado en comparaciones con la implementación de la sección anterior (el ejercicio), el algoritmo cuesta $O(n^2)$ tiempo.
- Esto coincide con el costo de Prim para el caso denso.
- A continuación vemos otra implementación que usa la estructura de datos *union-find*.

4.3. La estructura de datos union-find

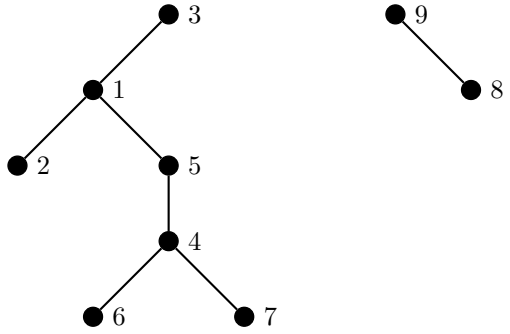
- La estructura de datos *union-find* sirve para representar al tipo abstracto disjoint-set.
- A diferencia de la implementación trivial, la idea es complejizar la operación find para mejorar la eficiencia de unite.
- Supongamos que queremos representar la partición $V_1, \ldots, V_k$ de $U = [1, n]$.
- Como invariante de representación, la estructura union-find mantiene un bosque enraizado F con k árboles $T_1, \ldots, T_k$, donde T_i representa a V_i para $1 \leq i \leq k$. Es importante remarcar que F nada tiene que ver con el bosque generador que computa Kruskal, dado que el tipo de datos es independiente del algoritmo de Kruskal.
- Cada árbol T_i tiene un vértice j por cada $j \in V_i$.
- La raíz de T_i se usa como su identificador, i.e., Find$(j) = r$ si y sólo si r es la raíz del árbol T_i que contiene a j.

- Importante: la estructura del árbol T_i es irrelevante. Dos árboles T y T' representan el mismo conjunto cuándo tienen los mismos vértices, independientemente de quién es padre de quién.

- La representación del bosque F es la misma que discutimos en la clase anterior. Se guarda un vector F de n posiciones, donde la i-ésima posición es j cuando el padre de i es j. En caso en que i sea la raíz, se almacena $F[i] = \perp$.

Estructura union-find

Representación del disjoint-set $\{1, \ldots, 7\}, \{8, 9\}$ con la estructura union-find. El algoritmo que vemos no necesariamente genera esta representación; se usa sólo para ejemplificar.

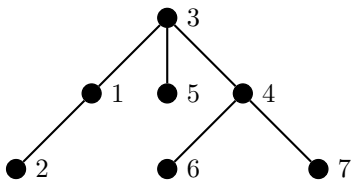


Create(n). Esta operación consiste en crear un nuevo árbol T_i con un único vértice v_i que queda como su raíz. El costo de esta operación es $O(n)$, ya que consiste en inicializar $F[i] = \perp$ para todo $1 \leq i \leq n$.

Find(i). Para implementar Find(i) alcanza con recorrer el camino $i = x(1), \ldots, x(k) = r$, donde r es la raíz del árbol T que contiene a i, y retornar $x(k)$. El costo de esta operación es $O(k)$ ya que consiste en k aplicaciones de $T[x(i)]$. En caso de ejecutar posteriormente una operación Find($x(i)$), con $1 \leq i \leq k$, el costo del algoritmo seria $O(k - i)$. Dado que la estructura de T (y F) es irrelevante, podemos mejorar esta implementación de la siguiente forma. Después de recorrer el camino $x(1), \ldots, x(k)$, seteamos $T[x(i)] = r$ para todo $2 \leq i \leq k$. De esta forma, cada $x(i)$ tiene como padre a la raíz y, en consecuencia, la aplicación sucesiva de Find cuesta $O(1)$ tiempo. Esta técnica, llamada *path relinking*, mejora la complejidad amortizada de Find.

Efectos de ejecutar Find

Árbol resultante de ejecutar Find(4) en la estructura union-find anterior cuando se aplica path relinking.

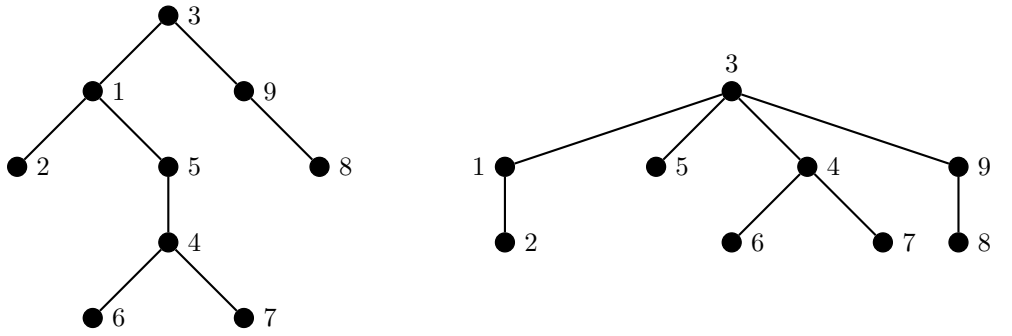


Unite(i, j). Esta operación se implementa agregando una de las aristas xy o yx a F, donde x es la raíz del árbol T_x que contiene a i e y es la raíz del árbol T_y que contiene a j. (Obviamente, esto suponiendo que $x \neq y$; caso contrario, no se agrega la arista.) Para la implementación, primero se ejecuta Find(i) y Find(j)

para obtener x e y, respectivamente. Luego, se realiza la asignación $T[x] = y$ o $T[y] = x$. El costo de esta operación está dado por el costo de Find. Una pregunta que surge es ¿que arista conviene agregar: xy o yx? A priori, dado que la estructura de F es irrelevante, se puede agregar cualquiera de las dos aristas. La idea, sin embargo, es minimizar el costo de Find que repercute también en el costo de Unite. Hay dos heurísticas básicas para decidir qué arista agregar. En la heurística *union-by-rank* se agrega xy cuando la altura de T_x es menor o igual a la de T_y. De esta forma, la altura del árbol resultante se minimiza. En la heurística *union-by-size* se agrega la arista xy cuando T_x tiene menos nodos que T_y. La idea en este caso es incrementar el nivel de la menor cantidad posible de nodos. Ambas heurísticas son igualmente buenas.

Efectos de ejecutar Unite

A la izquierda se muestra el resultado de Unite(3,7) y a la derecha el de Unite(4, 7), para la estructura de la figura de ejemplo. Suponemos la aplicación de path relinking y de las heursticas para la unión.



Teorema 4 ([8])**.** *Si la estructura union-find se implementa con la estrategia de path relinking y con alguna de las dos heurísticas union-by-rank o union-by-size, entonces Find (y por lo tanto Unite) requiere $T(n) = O(\alpha(n))$ tiempo amortizado, donde α es la inversa de la función de Ackermann.*

- La función f de Ackermann es una función cuyo origen fue demostrar que existen funciones totales que no son primitiva recursiva. El significado de esto queda para LyC.
- Lo importante es que f crece más rápido que cualquier función primitiva recursiva.
- En la práctica, $\alpha(n) \leq 4$ para cualquier n razonable (i.e., menor a 2^{1000}).
- En consecuencia, el algoritmo de Kruskal es "casi lineal" una vez que las aristas se encuentran ordenadas.

Teorema 5. *Sea G un grafo y c una función de costos. Se puede computar un AGM de (G, c) aplicando Kruskal en $T(n + m) = O(s(m) + \alpha(G)m)$ tiempo donde $s(m)$ es el costo de ordenar las aristas de G por costos.*

5. Prim vs. Kruskal (15 mins)

- Si bien Prim y Kruskal permiten computar un AGM de (G, c), los pasos intermedios que realizan son distintos.
- Tanto Prim como Kruskal pueden resolver problemas parciales, que describimos a continuación.
- Problema de árbol generador mínimo parcial.

  Input: un grafo G, una función de costos c, un vértice $v \in V(G)$ y un valor $k \leq n - 1$

  Output: el árbol T con $E(T) \subseteq E(G)$ de costo $c_+(T)$ mínimo que contiene a v y tiene k aristas.

- Problema de bosque generador mínimo.

  Input: un grafo G, una función de costos c y un valor $k \leq n-1$

  Output: el bosque generador F de G de costo $c_+(F)$ mínimo que contiene k aristas.

- Ejemplo de problemas

  ▷ Queremos conectar las localidades, pero sólo podemos construir k autopistas a la vez. Queremos seleccionarlas de forma tal que todas se conecten con una ciudad central. En este caso, queremos el árbol generador mínimo parcial.

  ▷ Tenemos n routers conectados a una red. Queremos instalar $k \ll n$ copias de una base de datos, que se accede de forma distribuida. La base de datos se puede instalar en cualquier router. Los routers en las que no se instala la base, tienen que comunicarse a través de la red, viajando por distintos routers. Pero, queremos maximizar el ancho de banda. En este caso, queremos obtener un bosque generador mínimo que tenga k bosques (i.e., $n-k$ aristas). Luego, podemos instalar la base en cualquier router de cualquier árbol del bosque.

- Ejercicio: demostrar que luego de k iteraciones, Prim resuelve el problema de árbol generador parcial.

- Ejercicio: demostrar que luego de k iteraciones, Kruskal resuelve el problema de bosque generador mínimo.

Referencias

[1] Otakar Borůvka. O jistém problému minimálním [on a certain problem of minimization]. *Práce Moravské Přírodovědecké Společnosti*, 3(3):37–58, 1926. In Czech.

[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] Vojtěch Jarník. O jistém problému minimálním. (z dopisu panu o. borůvkovi) [on a certain problem of minimization]. *Práce Moravské Přírodovědecké Společnosti*, 6(4):57–63, 1930. In Czech.

[5] Joseph B. Kruskal, Jr. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[6] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on minimum spanning tree problem: translation of both the 1926 papers, comments, history. *Discrete Math.*, 233(1-3):3–36, 2001. Graph theory (Prague, 1998).

[7] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, Nov 1957.

[8] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.*, 22:215–225, 1975.

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.[2]

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

A.1. Algoritmo Prim

El input del algoritmo esta dado por los valores n y m, seguidos de m triplas v, w, c que representan la arista vw con costo c. Para almacenarlas, vamos a extender la lista de adyacencias, guardando un par $(w, c(vw))$ en $N(v)$ por cada arista vw. La implementación del algoritmo de Prim visto en la Sección 3.1 a C++ es directa, salvo por el hecho de que utilizamos una cola de prioridad sin operación para eliminar. Pero, como se mencionó, alcanza simplemente con ignorar las aristas inseguras que se sacan de la cola. Un detalle importante es que la cola tiene que estar ordenada por c. Para esto, en lugar de guardar pares (v, w), guardamos triplas $(-c, v, w)$. La razón para negar c es para evitar tener que inicializar la cola con un comparador ya que, con esta representación, quedan ordenamos de mayor a menor lexicográficamente (es decir, primero por $-c$).

Implementación de Prim (prim.cpp)

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


const int none = -1;

int costo(neigh x) {return x.second;}
int vecino(neigh x) {return x.first;}

int main() {
  //transformacion de aristas a adyacencias
  int n, m, r; cin >> n >> m >> r;
  graph G(n);
  for(int i = 0; i < m; ++i) {
    int v, w, c; cin >> v >> w >> c;
    G[v].push_back({w,c});
    G[w].push_back({v,c});
  }

  //algoritmo de prim
  vector<int> T(n, none); T[r] = r;
  priority_queue<bridge> S;
  for(auto x : G[r]) S.push({-costo(x), r, vecino(x)});
  //el for se puede ejecutar hasta que S = vacio
  for(int i = 0; i < n-1;) {
    int c, v, w;
    tie(c,v,w) = S.top();
    S.pop();
    if(T[w] == none) {
      T[w] = v; ++i;
      //el if que sigue es opcional
      for(auto x : G[w]) if(T[vecino(x)] == none)
        S.push({-costo(x), w, vecino(x)});
    }
  }

  //output del algoritmo
  for(int i=0; i<n; ++i) cout << "T[" << i << "] = " << T[i] << "  "; cout << endl;
}
```

A.2. Algoritmo Prim para caso super denso

Si bien vamos a suponer el mismo input que para el caso anterior, vamos a aprovechar que suponemos que el grafo es denso para mostrar una variante. La idea es codificar el grafo pesado con una matriz de adyacencia C donde $C[v][w] = c$ cuando el costo de vw es c. En caso que la arista no esté, podemos definir $C[v][w] = \infty$. Luego, implementamos P como un vector donde $P[w] = v$ significa que vw es la arista de costo mínimo con $v \in V(T)$ y $w \notin V(T)$. Notar que no hace falta codificar el costo mínimo de w en un vector separado, porque es $C[w][P[w]]$. Para el caso en que $w \in V(T)$, simplemente definimos $P[w] = w$ porque $C[w][w] = \infty$ con lo cual nunca se va a seleccionar esta arista. La implementación que sigue es la adaptación del algoritmo de la Sección 3.2.

Implementación de prim para grafos con $\Omega(n^2)$ aristas (prim-denso.cpp)

```cpp
//matriz de costos
using costMatrix = vector<vector<int>>;
const int inf = numeric_limits<int>::max();
const int none = -1;

int main() {
  //transformacion de aristas a matriz
  int n, m, r; cin >> n >> m >> r;
  costMatrix C(n, vector<int>(n, inf));
  for(int i = 0; i < m; ++i) {
    int v, w, c; cin >> v >> w >> c;
    C[v][w] = C[w][v] = c;
  }

  //algoritmo de prim denso
  vector<int> T(n, none), P(n, r);
  T[r] = r;
  for(int i = 0; i < n-1; ++i) {
    int w = 0;
    for(int z = 0; z < n; ++z) if(C[z][P[z]] < C[w][P[w]]) w = z;
    T[w] = P[w]; P[w] = w;
    for(int z = 0; z < n; ++z) if(T[z] == none)
      if(C[w][z] < C[z][P[z]]) P[z] = w;
  }

  //output del algoritmo
  for(int i=0; i<n; ++i) cout << "T[" << i << "] = " << T[i] << "  "; cout << endl;
}
```

A.3. Disjoint-set con union-find + path relinking + union by size

A continuación se implementa la estructura union-find (Seccion 4.3) para representar el tipo disjoint-set (Sección 4.1). El bosque enraizado se implementa en el vector p. En el vector s se almacena la cantidad de nodos de cada árbol, de forma tal que $s[i]$ indica la cantidad de nodos del árbol con raíz i. Para aquellas posiciones i que no son raíces de un árbol, el valor $s[i]$ está indefinido. La función create se implementa en el constructor; en esta implementación definimos $p[i] = \perp$ para indicar que i es raíz. Recordemos que `find` consiste en encontrar la raíz. Esto lo resolvemos de forma recursiva: si $p[i] = \perp$, entonces i es una raíz; en caso contrario, buscamos la raíz desde el padre haciendo `find(p[i])`. La asignación `p[i] = find(p[i])` ocurre una vez que el `find` recursivo termina. En este punto, la expresión `find(p[i])` denota la raíz del árbol i. Luego, alcanza con hacer una asignación. Más aún, como se aplica recursivamente, esta expresión efectivamente asigna todo el camino desde i a su raíz. Finalmente, la función `unite` se implementa directamente.

Tipo de datos disjoint-set implementado con union-find (disjoint-set.h)

```
class DisjointSet {
  const int none = -1;
  //p = partent, s = size
  mutable std::vector<int> p, s;
public:
  DisjointSet(int n) {
    p.assign(n, none); s.assign(n, 1);
  }

  int find(int i) const {
    return p[i] == none ? i : (p[i] = find(p[i]));
  }

  void unite(int i, int j) {
    using std::swap;
    i = find(i), j = find(j);
    if(i != j) {
      if(s[i] < s[j]) swap(i, j);
      s[i] += s[j];
      p[j] = i;
    }
  }
};
```

A.4. Algoritmo de Kruskal

La implementación de Kruskal (Sección 4.2) es razonablemente directa una vez que ya esta implementado disjoint-set. La única observación interesante es que en este caso alcanza con considerar la lista de aristas de G. Pero, en lugar de almacenar solo un par (v, w) por cada arista vw, almacenamos la tripla $(c(vw), v, w)$. La razón de poner el costo en la primer posición de la tripla tiene que ver con que vamos a usar un sort por comparaciones básico provisto por C++, que ordena las triplas en orden lexicográfico.

Para este ejemplo, suponemos que la entrada es igual a la considerada para Prim, con excepción de la raíz que ya no es requerida.

Implementación de Kruskal (kruskal.cpp)

```
#include "disjoint-set.h"

using edge = tuple<int,int,int>;

int& cost(edge& e) {return get<0>(e);}
int& first(edge& e) {return get<1>(e);}
int& second(edge& e) {return get<2>(e);}

int main() {
  //lectura de la lista de aristas
  int n, m; cin >> n >> m;
  vector<edge> E(m);
  for(int i = 0; i < m; ++i)
    cin >> first(E[i]) >> second(E[i]) >> cost(E[i]);

```

```
16    //sort inicial
17    sort(E.begin(), E.end());
18    //algoritmo de Kruskal; imprime el output T
19    DisjointSet ds(n);
20    for(int i = 0, j = 0; i < n-1; ++i) {
21      while(ds.find(first(E[j])) == ds.find(second(E[j]))) ++j;
22      cout << "(" << first(E[j]) << "," << second(E[j]) << "), ";
23      ds.unite(first(E[j]),second(E[j]));
24    }
25    cout << endl;
26  }
```