Grafos.

Recordemos lo que alguna vez dijimos en este documento: un grafo es una pareja o tupla de dos conjuntos: vértices y aristas.

$$G(V,E)\text{: }\{V\ es\ un\ conjunto\ de\ vértices,\ E\ es\ un\ conjunto\ de\ aristas\}$$

$$G(V,E)\text{: }\{\ \{v1, v2,\dots\ , vn\}, \{(v1, v2), (v2, v3),\dots\ , (vi, vj)\}\ \}$$

No precisamente los vértices tienen que ser esos, es un ejemplo para hacer "light" la definición formal.

Existen tres grandes tipos de grafos:

Grafo no-dirigido.

a. No existe una direccionalidad de las aristas. Burdamente podríamos decir que aquí no hay una dirección con flechas. Con esto podríamos decir que las parejas de vértices no son ordenadas. Un ejemplo es facebook, pues las amistades en esta plataforma son "bi-direccionales", es decir, cuando se acepta una solicitud de amistad, Carmen es amiga de Sebastián y Sebastián es amigo de Carmen.
b. Dos vértices son adyacentes (vecinos) si están conectados por una arista.
c. Grafo Completo: es aquél que tiene todas las posibles aristas; en donde toda pareja de vértices está conectada por una arista.
d. Un camino entre v_a y v_b es una secuencia de aristas en E que los conectan. La longitud del camino es el número de artistas que los conectan.
e. Un Grafo es conexo si para cualesquier dos nodos (vértices) en V existe un camino que los conecta.
f. Un ciclo es un camino que comienza en va y termina en va. Un grafo sin ciclos se llama acíclico.
g. Qué es un árbol: Un grafo conexo no-dirigido, acíclico con un vértice designado como la raíz.

Grafo dirigido.

a. Existe una direccionalidad de las aristas y las representamos con una flecha. Con esto podríamos decir que las parejas de vértices son ordenadas. En twitter, a diferencia de facebook, es diferente, pues Sebastián puede seguir a Carmen, pero ella tal vez no a él.

b. No es lo mismo (v_a, v_b) que (v_b, v_a).
c. Un camino entre v_a y v_b es una secuencia de aristas (v_a, v_i) , (v_i, v_j) , … , (v_j, v_b)

Grafo ponderado.

a. En este caso se puede tener tanto a un grafo dirigido como a uno no-dirigido. En las aristas existen "etiquetas", es decir, que la arista puede tener asociado un peso, un número, un valor. Un ejemplo rápido es la creación de un grafo que en sus vértices representan ciudades y en sus aristas las carreteras entre las ciudades. La relación entre los vértices puede tener peso basado en el costo de casetas, la distancia, etcétera. Waze, por ejemplo, contempla algo de esta lógica para sus rutas [un peso puede ser el tráfico].

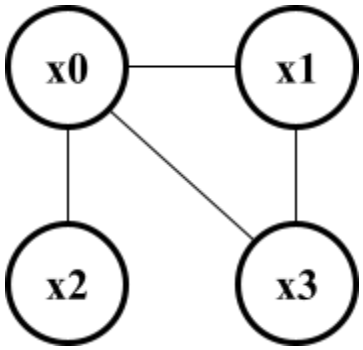
Lo bonito de los grafos, en palabras del profesor, es que son un formalismo para modelar fenómenos, un montón de cosas en el mundo. Ayuda a modelar ciudades, relaciones de personas, etcétera. Un juego cool es el de *6 degrees of Kevin Bacon*: tienes que nombrar películas al azar e ir pasándote de una a otra por personajes que tengan en común y tiene carácter de ser un grafo: es un camino en el espacio de película en donde las ligas están dadas por los personajes en común.
Uno de los resultados más importantes en sociología es lo que descubrió Stanley Milgram: el fenómeno del mundo chico. Este era un psicólogo que estudió en Yale. Hizo el experimento en el que quería ver qué tan conectada estaba la sociedad norteamericana. Es interesante conocer este experimento. Incluso hay *grafos de mundo chico* gracias a ello.
Incluso se puede modelar un grafo para analizar estrategias de comunicación para que se distribuyan ideas en un grupo, o estrategias de vacunación para una epidemia inesperada. También ayuda a modelar internet por medio de los links; worldwide web es una red de páginas interconectadas [*page ranking algorithm* es un algoritmo que inventaron los fundadores de Google que determina el orden de la importancia de las páginas].

Esponda cree que las cosas más difíciles de entender del mundo tienen que ver con las relaciones de las cosas. Todos nosotros somos un conjunto de elementos: podemos determinar exactamente cuánto de Carbono, Hidrógeno, Oxígeno y Nitrógeno tiene en su organismo e ir a comprar a la tiendita exactamente esas cantidades de esos elementos que nos conforman, pero al momento de meterlos en una bolsa para combinarlos no va a salir una persona como tú tal cual ni nada parecido. ¿Por qué? Tenemos los ingredientes, pero lo que importa son las relaciones, cómo se conectan uno con otro. En

estas conexiones es donde está la inteligencia, la vida, etcétera. Un grafo es un formalismo que tenemos para modelar las relaciones.

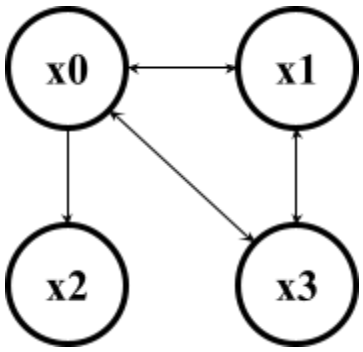
*¿Cómo **representamos** un grafo en una computadora?*
Existen dos maneras:

1. **Matriz de adyacencia**. Adyacente significa que esté al ladito o juntos.



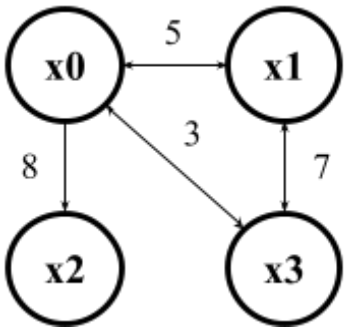
	x0	**x1**	**x2**	**x3**
x0	0	1	1	1
x1	1	0	0	1
x2	1	0	0	0
x3	1	1	0	0

Este grafo, por ejemplo, es un grafo no-dirigido. Además, la matriz asociada a ella es una matriz simétrica. Si fuera un grafo dirigido, su matriz no sería simétrica:



	x0	x1	x2	x3
x0	0	1	1	1
x1	1	0	0	1
x2	0	0	0	0
x3	1	1	0	0

Si el grafo fuera ponderado, en vez de poner "1" en sus relaciones, podríamos poner el entero que representa al peso de esa relación.



	x0	x1	x2	x3
x0	0	5	8	3
x1	5	0	0	7
x2	0	0	0	0
x3	3	7	0	0

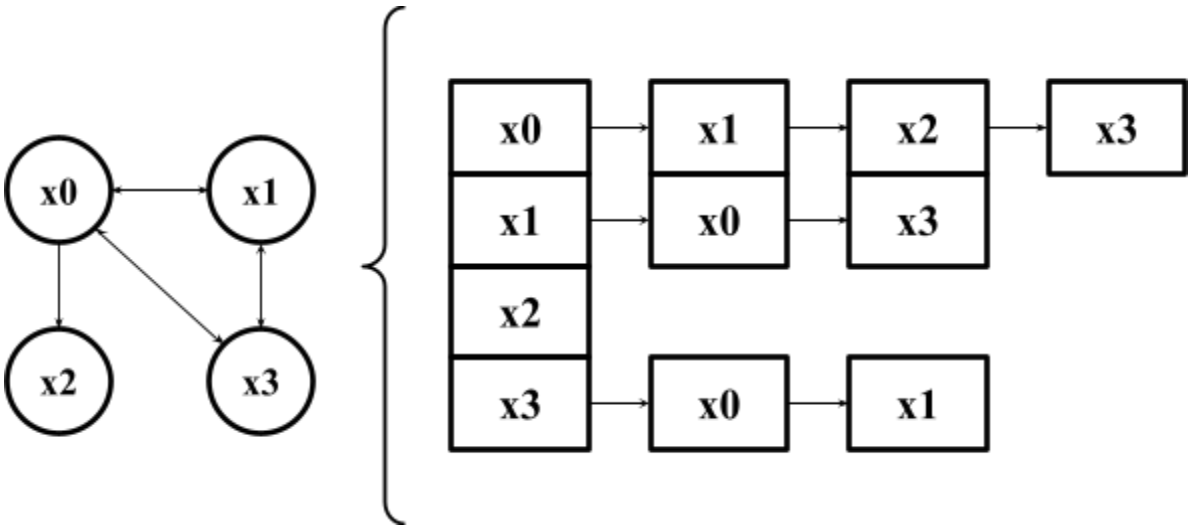
Desventaja:

La matriz puede ser rala si el conjunto de aristas es chico | E | respecto al de vértices | V |. Esto es porque aquí se representa con quiénes se relaciona **x0**, pero también con quiénes no. Esto es lo que genera un desperdicio de espacio y de todo.

Ventaja:

Saber si dos vértices están conectados es de tiempo constante *O(1)*. Además, otra ventaja es que como es una matriz, podemos aplicarle operaciones. Aquí entra todo el bagaje de *Álgebra Lineal* para poder extraer la información y las características importantes de un grafo representado en una matriz.

2. **Lista de adyacencia**. Aquí se tiene un arreglo con una entrada por vértices en sus casillas, pero en cada vértice se tiene una lista ligada con todos los vértices con los que él está ligado. Aquí tenemos una lista de con quiénes se relaciona **x0** nada más.



Para expresar ponderación se puede tener en el nodo un atributo que represente su peso. Para grafos muy grandes es conveniente utilizar esta representación porque estos suelen ser ralos, suelen haber pocas aristas en relación al número de aristas posibles.

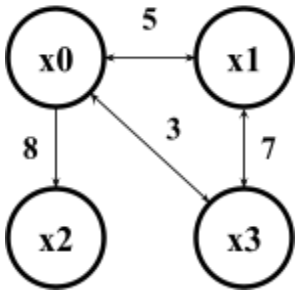
Una desventaja rápida que podemos inferir es que podemos tener el problema de que exista un desempeño de *O(n)* en la búsqueda de un elemento. Esto es porque si se tiene el caso de que un vértice está unido con todos [todos] los vértices del grafo, si queremos conocer si un vértice está conectado con él, tendríamos que recorrer una lista ligada de *n* datos si es que la cantidad de vértices es de *n*. Pero bueno, ya veremos después qué onda.

*¿Cómo **recorrer** un grafo?*

Recordemos que en un árbol existen 3 recorridos principales: el pre-order, el in-order y el post-order. La diferencia entre un árbol y un grafo general es que aquí ya no tienen sentido esos recorridos porque ya no hay solamente dos o un hijo. Aquí nada más tenemos dos recorridos: uno a profundidad **DFS** [me visito yo, luego visito a uno de los hijos, después a uno de los hijos de mi hijo, etcétera (se dejan los otros hijos pendientes)] y otro en anchura **BFS** [me visito yo, visito mis vecinos inmediatos y luego los vecinos inmediatos de mis vecinos inmediatos, etcétera]. Siempre se tiene que empezar por algún vértice, pues no hay una raíz como tal. Otra diferencia es que esto no es un árbol: un grafo conexo acíclico; por lo tanto, hay dos cosas que tenemos que considerar: cómo le hacemos para no caer en un ciclo y si no es conexo, cómo le hacemos para llegar a los otros elementos del grafo.

Representación en Python.

La representación de esta estructura en Python se implementará con ayuda de diccionarios, que emulan la representación de la lista de adyacencia. ¿Cómo? Se va a tener un diccionario de vértices como llave donde cada vértice tiene otro diccionario como valor con los vértices con los que está relacionado y su ponderación. Este es un ejemplo sencillo:



Representación:

$$C = \{\, x0: \{\, x1: 5\,,\ x3: 3\,,\ x2: 8\,\}\,,\ x1: \{\, x0: 5\,,\ x3: 7\,\}\,,\ x3: \{\, x0: 3\,,\ x1: 7\,\}\,,\ x2: \{\,\}\,\}$$

De esta manera podremos obtener el valor **C[x1][x3] = 7**, por ejemplo.

Si nos damos cuenta, cada vértice tiene asociado un diccionario con los vértices como llave y la ponderación como valor en donde podemos ubicar las relaciones entre cada vértice en el grafo. Aquí podemos observar la ventaja de utilizar diccionarios en Python para representar esta estructura de

datos, puesto que es una tabla de hash y para estas alturas del documento, ya sabemos las implicaciones.

Algunas anotaciones sobre los recorridos en Python.

Al hacer los recorridos, nos podemos encontrar con la mala suerte de "volver a visitar" nodos que ya habían sido procesados. Es por ello que lo que se va a implementar en código es otro diccionario con el cual se tendrá el control sobre si un nodo ya fue visitado o no. Este diccionario contendrá en su interior los vértices como llaves asociados a una variable booleana como valor para indicar si ya fueron visitados los nodos y continuar con el recorrido.

APUNTE NO. 24 Miércoles 5 de mayo

Árboles de expansión.

Si se tiene un grafo, suponiendo que sea conexo, nos gustaría poder extraer un árbol a partir de ese grafo. ¿Qué quiere decir esto? Tener todos los vértices y un subconjunto de las aristas en otro grafo, particularmente en un árbol: un grafo conexo acíclico. Poder encontrar un árbol a partir de un grafo es útil para ciertas cosas. Por ejemplo: hacer llegar algo a los *n* nodos sin que se repitan; o si tenemos un chip: los vértices son los pines del chip y queremos conectar todos con todos para que sea eléctricamente adyacentes, pero se desea utilizar el mínimo número de cables; o mandar un mensaje a *n* computadoras, conectar *n* ciudades sin ciclos. Incluso una red adhoc, que se hace con teléfonos celulares y no hay una conexión física entre ellos; si todos estamos conectados por bluetooth, ahora los mensajes tienen que saltar de teléfono a teléfono y no usando la red de telmex, así que quiero saber cómo ruteo y eso se puede lograr con un árbol de expansión. Encontrar el árbol de un grafo [la columna vertebral] es un problema interesante.

Ahora vamos a complicar un poco todo, imaginemos que el grafo es ponderado, ¿qué nos gustaría hacer aquí? Existen muchos árboles que se pueden extraer de un árbol, pero nos gustaría encontrar aquel árbol que tenga el peso mínimo, es decir, que la suma del valor asociado a las aristas en este árbol sea la menor cantidad. El problema que queremos atender es que dado un grafo conexo ponderado, queremos encontrar el **árbol de expansión mínima**. Se llama árbol de expansión mínima porque es un árbol [tiene una raíz], están todos los nodos del grafo en él y la suma de los costos de todas las aristas es el mínimo valor posible [empezando por un nodo].

Tanto DFS Y BFS generan un árbol de expansión [diferentes], pero no son de expansión mínima porque no se usa ningún criterio de visitar primero las aristas más baratas. Entonces ¿cómo le vamos a hacer para lograr el objetivo? con ayuda del Algoritmo de Prim. La lógica del algoritmo es que para ir conectando los nodos se va a tomar la arista mínima de todas las aristas del grafo que se está construyendo con las del grafo remanente o que falta.

Algoritmo de Prim.

Podemos explicar este algoritmo de una manera didáctica. Pensemos que existe una línea imaginaria que divide al grafo existente con el árbol de expansión mínima que vamos a desarrollar. Se tienen que tomar todos los vértices del grafo, entonces el algoritmo empieza con todos los vértices del lado del grafo y termina cuando todos los vértices están del lado del árbol de expansión mínima. Entonces, ¿cómo funciona?

Pasos:

1. Se toma un vértice (el que sea) como el primero y se pone del lado del árbol de expansión mínima. Ese vértice está conectado con el grafo por varias aristas. En este caso, necesariamente escojo la arista de menor peso y pasamos este otro vértice al lado del árbol de expansión mínima.
2. Este nuevo vértice, junto con el primero, están conectado a todo el resto del grafo por un montón de aristas. De entre todas estas, escojo la más chica, y pasamos ese nuevo vértice del lado del árbol de expansión mínima.
3. Continuamos con este proceso. En cada paso pasamos un vértice al árbol que se está construyendo, siempre escogiendo la arista de menor peso que conecte el grafo y el árbol que se está creando.

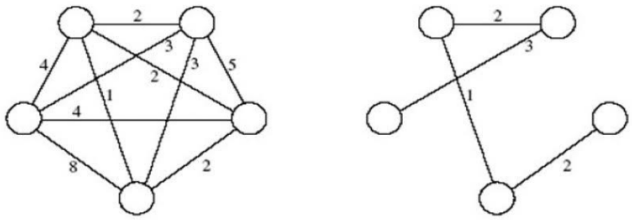
Notemos que siempre va a haber al menos una arista entre los grafos (dada nuestra suposición de que el grafo es conexo) por lo que incluso si “subimos” un vértice que ya no tiene aristas, seguirán habiendo más aristas conectadas con otros (a menos que ya hayamos terminado). Otra cosa que se debe tener clara es que si un vértice está conectado con otros dos que tienen diferente ponderación, solamente se va a quedar la unión de la arista con menor valor, no las dos aristas, pues esto último implicaría un ciclo en el grafo [árbol].

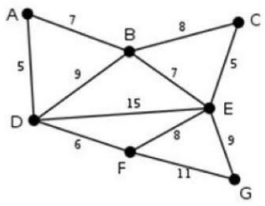
El *output* del algoritmo es un subconjunto de las aristas, porque sabemos que se tienen que considerar todos los vértices, pero solo se tomarán ciertas aristas. Ahora veremos el pseudocódigo.

Ejemplo del algoritmo de Prim.

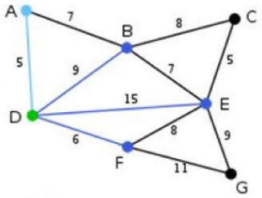
Árboles de expansión: Algoritmo de Prim (I)

- Un **árbol de expansión** de un grafo no dirigido G=(V,A) y conexo, es un subgrafo G'=(V,A') no dirigido, conexo y sin ciclos. Importante: contiene todos los vértices de G.
- El **algoritmo de Prim** intenta encontrar un árbol de expansión de un grafo, cuyas aristas sumen el peso mínimo.

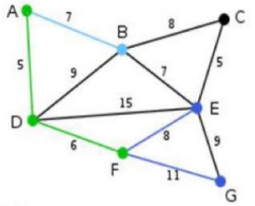




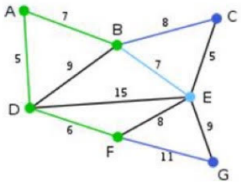
D será el punto de partida.



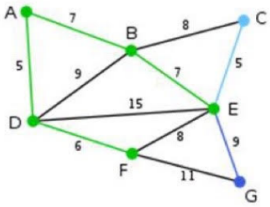
El segundo vértice es el más cercano a **D**: **A** está a 5 de distancia, **B** a 9, **E** a 15 y **F** a 6. De estos, 5 es el valor más pequeño, así que marcamos la arista **DA**



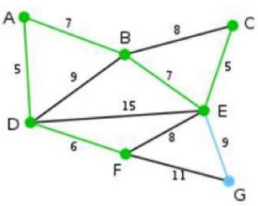
El vértice **B**, que está a una distancia de 7 de **A**, es el siguiente marcado.



Aquí hay que elegir entre **C**, **E** y **G**. **C** está a 8 de distancia de **B**, **E** está a 7 de distancia de **B**, y **G** está a 11 de distancia de **F**. **E** está más cerca, entonces marcamos el vértice **E** y la arista **EB**.



Sólo quedan disponibles **C** y **G**. **C** está a 5 de distancia de **E**, y **G** a 9 de distancia de **E**. Se elige **C**, y se marca con el arco **EC.**



G es el único vértice pendiente, y está más cerca de **E** que de **F**, así que se agrega **EG** al árbol. Todos los vértices están ya marcados, el árbol de expansión mínimo se muestra en verde. En este caso con un peso de 39.

Pseudocódigo Algoritmo de Prim.

$for\ each\ u \in V[C]$:
    $key\ [u] \leftarrow \infty$
    $\pi\ [u] \leftarrow null$
$key\ [r] \leftarrow 0$
$Q \leftarrow V[C]$
$while\ Q \neq \{\}$:
    $u \leftarrow ExtractMin(Q)$
    $for\ each\ v \in Adj[u]$:
        $if\ v \in Q\ and\ w(u,v) < key[v]$:
            $h[v] \leftarrow u$
            $key[v] \leftarrow w(u,v)$

Lo podemos leer de la siguiente manera:

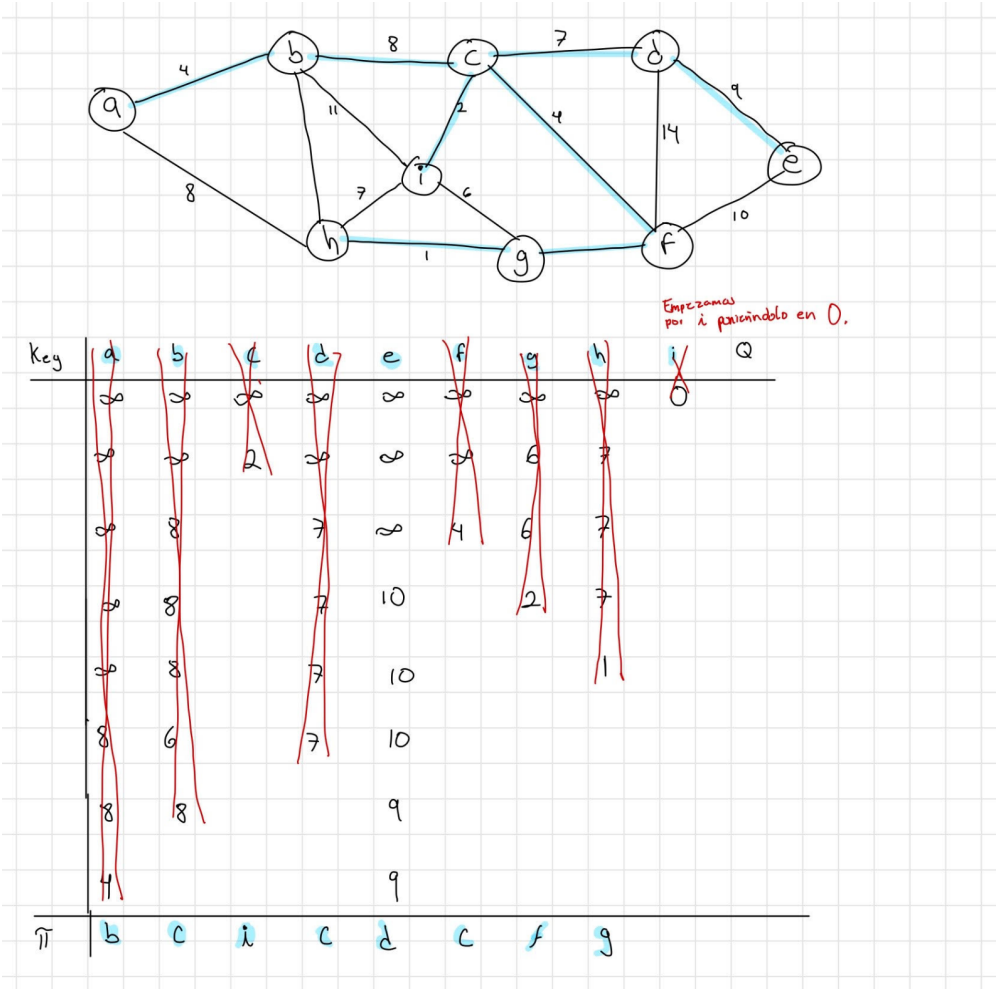
- Para cada vértice que está en el grafo, se hace una inicialización en la cual el valor de las **llaves** [estas llaves van a permitir escoger el mínimo de los pesos de las aristas] van a ser infinito y el valor de **π** [que es donde se va a guardar "quién es papá de quién", es decir, las aristas resultantes] va a ser nulo.
- La llave de **r** va a ser cero y en **Q** [pensémoslo como una estructura de datos] se van a guardar todos los vértices del grafo. De esta Q o estructura de datos es como se van a ir sacando uno por uno los vértices.
- Mientras que Q no esté vacía [vamos a terminar cuando ya no se tengan más vértices que considerar], se extrae el vértice que tiene la arista con el peso de menor valor que cruza el corte y se pone en **u**.
- Para cada vértice **v** que está en la lista de adyacencia de **u**, es decir, para cada vértice que está conectado con el que acabamos de sacar, si **v** está en **Q** [esto quiere decir que todavía no está en el árbol de expansión mínima] y el peso de la arista del vértice **u** al vértice **v** es menor que lo que se tiene guardado en la llave de **v**, entonces el papá de **v** es **u** y el peso de **v** es el peso de la arista del vértice **u** al vértice **v**.

En cada ciclo del while se va a sacar a un vértice, entonces el while va a correr el número de vértices haya en la estructura. ¿Cuál es el que se saca? El que tenga el mínimo valor en la llave. Inicialmente, quien tiene el mínimo del valor de la llave es **r**: el vértice arbitrario por el que va a empezar el algoritmo, y este va a ser el primer **u**. Ahora, para cada vecino que tenga **u**, si el vecino no lo hemos

sacado y el peso de la arista que conecta **u** con el vecino es menor a la llave que tengo guardada en el vecino, entonces el papá del vecino es **u** [esta es la única arista que nos importa] y el peso va a estar dado por el de esta unión.

Veamos un ejemplo de este algoritmo con un grafo real:

- Se empezó el algoritmo con el vértice **i**, entonces la llave inicial de este vértice va a ser 0.
- El valor de las llaves está dado por los recuadros del diagrama.
- Poco a poco se van tachando todos los vértices que están en Q cuando se extrae el mínimo valor de las llaves [y ya no va a pertenecer a Q], por lo que el diagrama muestra el resultado final.
- La última fila del diagrama representan los papás de los vértices. Esta fila se va actualizando con cada iteración del while, por lo que el diagrama demuestra el estado final de los papás de los vértices.
- Al final solamente se tienen que conectar los vértices con el papá correspondiente.



Este diagrama fue brindado por el compañero de la clase que estaba desarrollando el apunte del día correspondiente.

Es relativamente sencillo llegar al resultado final, solamente hay que seguir el código y tener confianza en uno mismo.

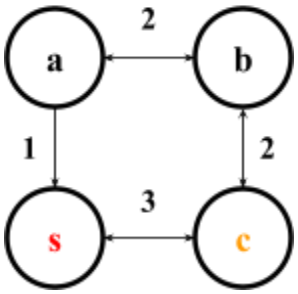
Notas sobre la implementación del algoritmo de Prim en Python.

- Va a ser necesario tener un diccionario para las aristas finales [las representadas en π]
- ¿Cómo vamos a extraer el mínimo valor de las llaves? Se pueden guardar el valor de las llaves en un min-heap para fácilmente extraer estos valores cuando sea necesario. ¿Por qué es inteligente utilizar esta estructura? La respuesta está en el apartado *Heaps*. Fácilmente podría decirse que un min-heap está hecho para esto.
- Para tener la representación de Q, podemos utilizar el diccionario de visitados.
- Se va a implementar en Python la biblioteca *heapdict*.

APUNTE NO. 25 *Lunes 10 de mayo*

¿El camino más corto en un grafo?

Suponiendo que tenemos un grafo ponderado, conexo y dirigido, ahora nos gustaría saber cuál es la ruta con el menor costo para llegar de un vértice a otro. Esto es lo que hace Waze, por ejemplo. Este algoritmo no es el mismo que el Algoritmo de Prim, aunque a primera vista parece ser que sí. La generación de un árbol de expansión mínima y la ruta más corta entre dos vértices son conceptos completamente diferentes. Lo podemos ejemplificar rápido en el siguiente diagrama:



El árbol de expansión mínima empezando por **s** está dado por: $s - a - b - c$. Pero la ruta más corta entre **s** y **c** está dado por: $s - c$. Encontrar el árbol de expansión mínima no nos está dando la ruta más corta entre un vértice particular y otro.

Una restricción que se tiene para este nuevo algoritmo es que no se pueden tener ciclos en donde la suma de los pesos del ciclo sea negativa porque cada vuelta va a resultar que es un camino más y más barato cada vez. ¿Cómo podemos comprobar eso? Quién sabe, pero es una buena idea pensar.

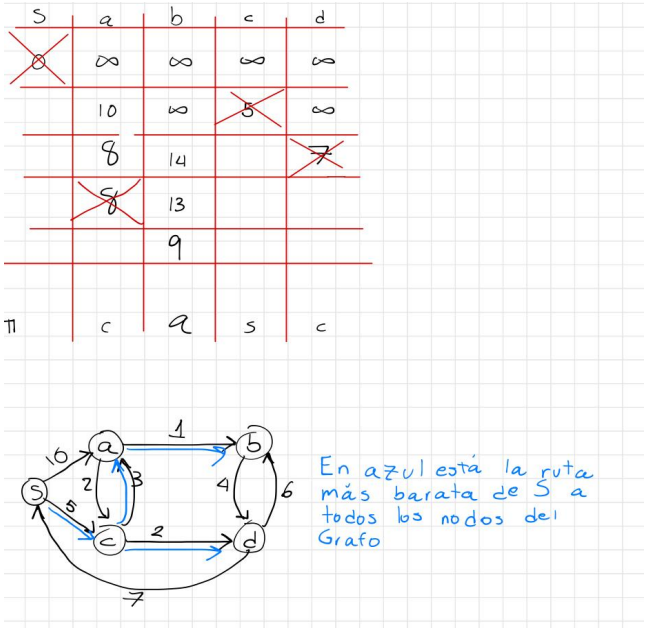
El objetivo es intuitivamente muy claro: **tenemos una fuente *s* de *source* y queremos llegar a todos los destinos de la manera más barata posible**. El problema que se tiene es el de acumular el costo de la ruta. El algoritmo tiene el nombre de uno de los más venerados y Santo Patrono **Dijkstra**.

Pseudocódigo Algoritmo de Dijkstra.

$for\ each\ u \in V[C]$:
    $key\ [u] \leftarrow \infty$
    $\pi\ [u] \leftarrow null$
$key\ [s] \leftarrow 0$
$Q \leftarrow V[C]$
$while\ Q \neq \{\}$:
    $u \leftarrow ExtractMin(Q)$
    $for\ each\ v \in Adj[u]$:
        $if\ v \in Q\ and\ key[u] + w(u,v) < key[v]$:
            $h[v] \leftarrow u$
            $key[v] \leftarrow key[u] + w(u,v)$

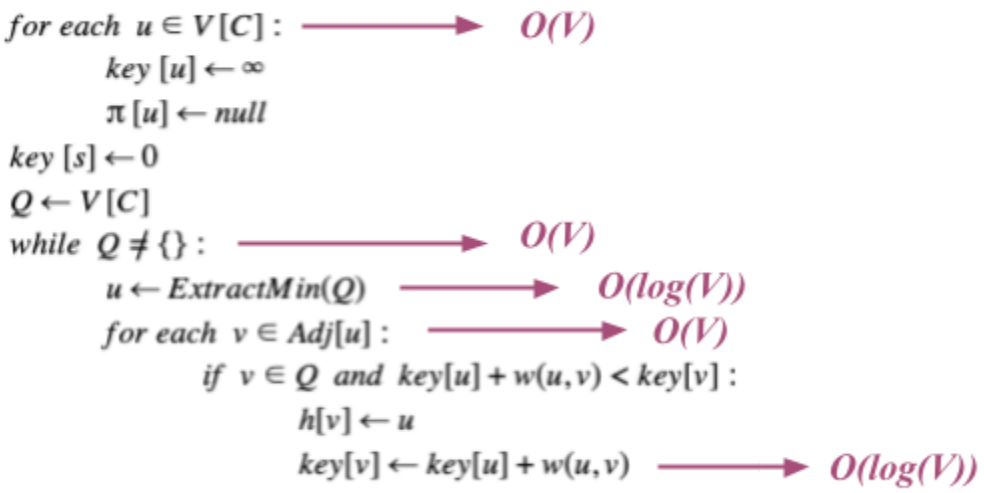
- **key[u]** va a tener el acumulado de de los valores de **w** que componen un camino.
- La única diferencia de este algoritmo con el de Prim es que para las rutas se tienen que acumular el costo a los lugares intermedios porque sí se tiene que pasar por ellos para llegar.

De esta manera, se puede ver un ejemplo de este algoritmo con un grafo real:



Este diagrama fue brindado por el compañero de la clase que estaba desarrollando el apunte del día correspondiente.

¿Cuánto se tarda el algoritmo de Dijkstra?



- Si pensamos que la función de ExtractMin la vamos a implementar con ayuda de un min-heap, entonces lo que se va a tardar esa instrucción es ***O(log(V))***.
- En la instrucción de *para cada vértice* ***v*** *adyacente a* ***u*** podemos decir que se tarde ***O(V)*** en su peor caso si suponemos que cada vértice está conectado con todos los vértices del grafo.
- De la misma manera, en la instrucción *key[v] ← key[u] + w(u,v)* se va a tardar ***O(log(V))*** debido a que estamos cambiando un valor de algún nodo del min-heap.

Por lo tanto, lo que se tarda el algoritmo de Dijkstra en su peor caso es:

$$O(V + Vlog(V) + V \cdot Vlog(V))$$

$$O(V + Vlog(V) + E \cdot log(V))$$

$$O(E \cdot log(V))$$

¿Qué es E? Es la cardinalidad de las aristas en el grafo, porque en el peor de los casos cada vértice está conectado con todos los restantes.