

## Cómo implementar eficientemente el algoritmo de Dijkstra?

La idea clave será utilizar un **priority queue** ó **heap**. Esta es una estructura de datos particular que interactúa muy bien con Dijkstra.

Obs: La interacción entre algoritmos y estructuras de datos es una idea clave para lograr programas eficientes. Para ello hay que tener familiaridad con varias estructuras de datos así que empezaremos por discutir las ideas generales sobre **heaps**.

Def: Un **heap** o **priority queue** es como un diccionario  $\{\text{Keys} : \text{Values}\}$  donde las Keys son elementos de un conjunto totalmente ordenado (por ejemplo los enteros). Un heap  $H$  tiene algunas operaciones **MUY EFICIENTES**:

(1)  $H.\text{INSERT}(k, v)$ : Introduce una nueva pareja  $(k, v)$  en  $H$ .

(2)  $H.\text{ExtractMin}()$ : Encuentra en  $H$  una pareja  $(k^*, v^*)$  donde  $k^*$  es el **MÍNIMO** de las keys de  $H$  en el orden total. Retorna  $(k^*, v^*)$  y lo elimina de  $H$ . (Si el mínimo se alcanza en varias parejas retornar y quitar cualquiera de ellas)

(3)  $H.\text{DELETE}(p)$  Dado un apuntador  $p$  a un objeto  $(k, v)$  en  $H$ , lo quita de  $H$ .

Teorema: Existe una manera <sup>H</sup> de representar los datos que permite realizar las operaciones (1) y (2) en tiempo  $O(\log(n))$  donde  $n$  es el número de parejas en  $H$

Al final explicaremos cómo, por ahora veámoslas...

EN PYTHON:

```
import heapq
H = []
heapq.heappush(H, (1, "definición"))
heapq.heappop(H)
```

INSERT

EXTRACT MIN de este key

Cuándo se usa un heap?

"Cuando queremos incluir objetos (en un orden cualquiera) y extraerlos en orden creciente (es decir, el más bajito primero)"

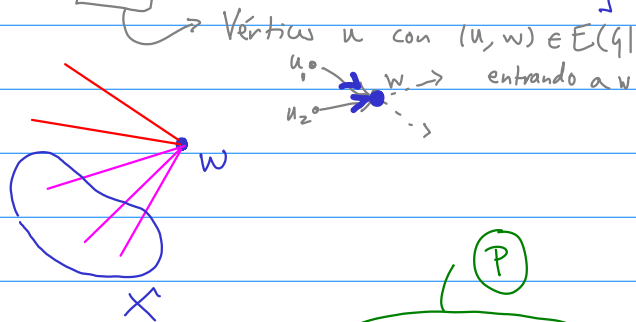
El ejemplo más sencillo es un "event\_handler" por ejemplo al programar un juego en el que registramos el instante <sup>key</sup> en que deben ocurrir una serie de eventos <sup>value</sup> y usamos el heap para saber cuál es el más próximo a ser procesado

EXTRACT - MIN.

Dijkstra con heap: [Usaremos la notación de Dijkstra de la clase anterior...]

Idea: Mantendremos un **heap** que recuerde los vértices de  $V(G) \setminus X$  donde

$$(*) \quad \left[ \begin{array}{l} \text{value}(w) = \text{"índice del vértice } w \in V(G)\text{"} \\ \text{key}(w) = \min_{u \in \text{In}(w) \cap X} \{ l(u, w) + \varphi(u) \} \end{array} \right]$$



$$w^* = \text{EXTRACT-MIN}(H),$$

$$\textcircled{P} \quad \left[ \begin{array}{l} X \leftarrow X \cup \{w^*\} \\ \varphi(w^*) = \text{key}(w^*) \end{array} \right]$$

Lema: Si los keys de  $H$  satisfacen la ecuación  $(*)$  entonces  $\textcircled{P}$  lleva al siguiente paso correcto de Dijkstra

Dem: Este paso puede parecer sorprendente pues Dijkstra busca el mínimo entre las aristas que cruzan, pero

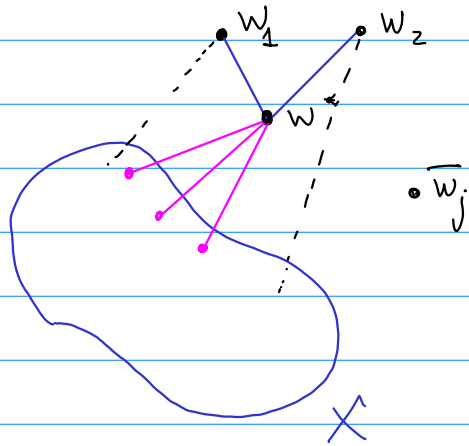
$$\min_{u, w: \substack{u \in X, w \notin X \\ (u, w) \in E(G)}} l(u, w) + \varphi(u)$$

$$\min_{w \in V \setminus X} \left[ \min_{\substack{u \in X \\ (u, w) \in E(G)}} l(u, w) + \varphi(u) \right]$$

$\text{key}(w)$   
demostrado el lema

El problema es que, después de incluir  $w^*$  en  $X$  la ecuación  $(*)$  puede fallar en algunos vértices **PORQUE  $X$  CAMBIÓ** y debemos

ajustar nuestro **heap** para que la ecuación (\*) se cumpla para el nuevo conjunto  $X$ .  
 Para ello debemos entender donde puede haber cambios:



Si  $\neg \exists (w^*, \bar{w}_j) \Rightarrow \text{key}(\bar{w}_j) \checkmark$  OK mejor resultado con aristas existentes

Si  $\exists (w^*, w_i) \Rightarrow \text{key}(w_i) = \min \{ \text{key}(w_i), \underbrace{\ell(w^*, w_i) + \varphi(w^*)}_{\text{nuevo candidato por el mínimo}} \}$

→  $Q \leftarrow Q \cup \{w^*\} \cap [X \setminus V(G)]$

Cuánto, llamados a las operaciones del Heap hacen?

EXTRACT-MIN — 1 vez cada vértice

DELETE e INSERT — máximo una vez por arista

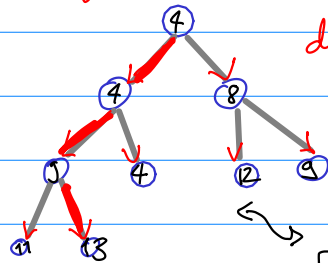
Teorema: [Dijkstra con heaps] Si  $G$  es un grafo cualquiera,  $s \in V(G)$  y  $\ell \geq 0$  es una colección de pesos positivos entonces todos los valores  $d_\ell(s, v)$ ,  $v \in V(G)$  pueden calcularse en  $\mathcal{O}(\ell(m+n) \log(n))$

Obs: Esta implementación es MUY eficiente y puede correr en grafos de millones de vértices en un laptop.

Cómo funciona un heap?

Def: Un heap es un árbol binario con números en los vértices  $key(v)$  orientados hacia abajo de tal manera que  $key(v)$  es no decreciente a lo largo de todo camino.

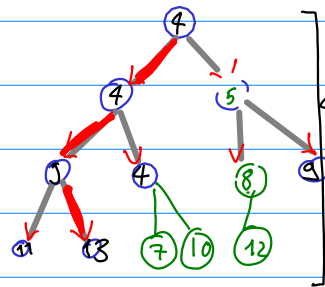
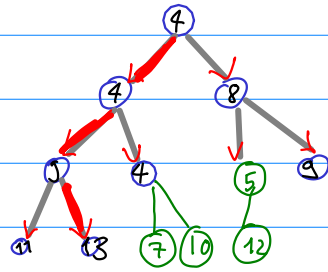
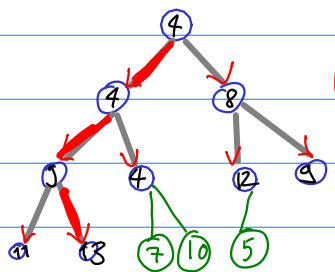
Heap Property



4, 4, 9, 13

Array de izquierda a derecha  
[4, 4, 8, 9, 4, 12, 9, 11, 13]

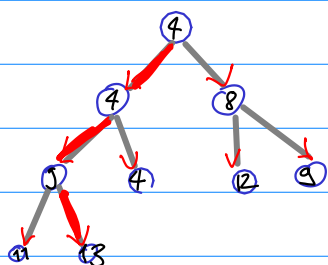
INSERT (7, H) (10, H) (15, H)  
↑ interesante.



tiene de nuevo el heap property.

Cuántos cambios tenemos que hacer para INSERT?  
máximo  $O(\log_2(n))$ , que es el número de niveles del árbol.

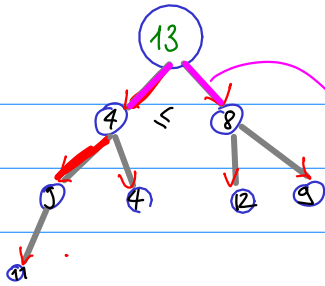
Para EXTRACT MIN Hacemos lo siguiente:



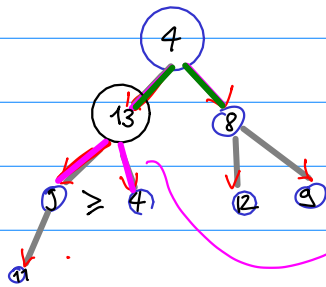
(1) Sacamos la raíz, que será el min y la reemplazamos por el último elemento así...

Raíz: 4

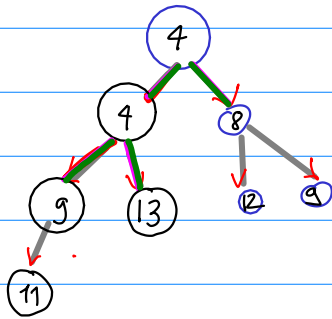
y ponemos el último elemento (13)



- ② Reemplazo el padre 13 por el menor de sus dos hijos



- ③ Reemplazo el padre 13 por el menor de sus dos hijos



Todo OK! y el número de operaciones es

$O(\log(n))$  porque en cada paso bajamos el elemento un nivel y hay  $\log(n)$  niveles.