Los grafos como funciones

0. Nociones generales.

Sean $\mathbb{A}$ y $\mathbb{B}$ dos conjuntos de elementos.

Se define $\mathbb{A}\times\mathbb{B}$ como el producto cartesiano de los elementos de $\mathbb{A}$ con los elementos $\mathbb{B}$, de forma tal que $\mathbb{A}\times\mathbb{B} = \{(x, y) \mid x \in \mathbb{A} \wedge y \in \mathbb{B}\}$.

Se define $\mathcal{P}(\mathbb{A})$ como las partes o las potencias de $\mathbb{A}$ y es el conjunto que contiene todos los posibles subconjuntos de $\mathbb{A}$, de forma tal que $\mathcal{P}(\mathbb{A}) = \{\mathbb{A}' \mid \mathbb{A}' \subseteq \mathbb{A}\}$.

Se define ${}_{\mathbb{A}}\mathcal{R}_{\mathbb{B}}$ como una relación entre los conjuntos $\mathbb{A}$ y $\mathbb{B}$, de forma tal que ${}_{\mathbb{A}}\mathcal{R}_{\mathbb{B}} \subseteq \mathbb{A}\times\mathbb{B}$.

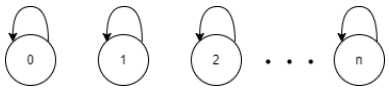
1. Grafos.

Sea $\mathbb{V}$ un conjunto de elementos que llamaremos vértices.

Se define $\mathbb{A}$ como un conjunto de parejas de vértices a las cuales llamaremos aristas, de forma tal que $\mathbb{A} = {}_{\mathbb{V}}\mathcal{R}_{\mathbb{V}}$.

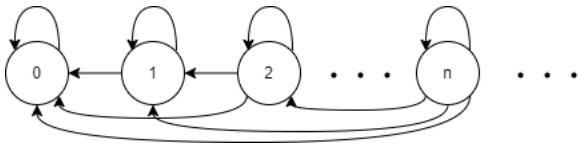
Sea ϑ un grafo, éste se define como una función que relaciona un conjunto de vértices de $\mathbb{V}$, con elementos del propio conjunto, de forma tal que $\vartheta : \mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})$.

`Por ejemplo.` Si queremos definir un grafo que está formado por puros lazos, podríamos definirlo de la siguiente forma:



$\vartheta_{id} : \mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})$
$\vartheta_{id}(v) = \{v\}$

`Otro ejemplo.` Si queremos definir un grafo que está compuesto por las aristas que relacionan a cada vértice consigo mismo y sus predecesores, podríamos hacerlo de la siguiente forma:

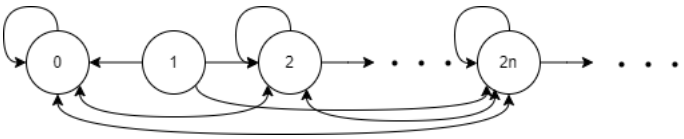


$\vartheta_{leq} : \mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})$
$\vartheta_{leq}(v) = \{\ x \mid x \in \mathbb{V} \ \wedge \ x \leq v\}$

`Otro ejemplo.` Si queremos definir un grafo que está compuesto por las aristas que relacionan a cada vértice con su sucesor, podríamos hacerlo de la siguiente forma:

$\vartheta_s : \mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})$
$\vartheta_s(v) = \{v + 1\}$

`Último ejemplo.` Si queremos definir un grafo que está compuesto por las aristas que relacionan a cada vértice con todos vértices pares, podríamos hacerlo de la siguiente forma:



$\vartheta_{even} : \mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})$
$\vartheta_{even}(v) = \{\ x \mid x \in \mathbb{V}\ \wedge\ x = \dot{2}\}$

2. Representación.

Aprovechando el la sencillez de representación y manipulación de funciones en el modelo funcional, utilizaremos un lenguaje de este modelo para la representación de grafos como funciones, elegiremos en este caso `Haskell` como lenguaje de implementación.

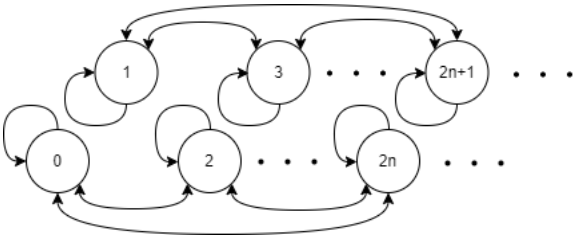
```
--Grafos
type N = Integer --consideraremos solo a los naturales.
type Conj a = [a]
type V = N
type G = V -> Conj V

--Ejemplos mencionados anteriormente
gid :: G
gid v = [v]

--Necesaria para obtener los vértices del grafo
vertices :: G -> Conj V
vertices g = [0..]

gleq :: G
gleq v = [x | x <- vertices gleq,  x <= v]

gs :: G
gs v = [v+1]

geven :: G
geven v = [x | x <- vertices geven,  even x]
```

3. Grafos no regulares.

Como podemos observar en los ejemplos y en el código `Haskell`, podemos representar grafos infinitos, donde a cada vértice le asignamos una expresión particular (lo cual aparentemente obliga a los mismos a tener un comportamiento regular). Para esto hace falta recordar el concepto de `funciones partidas (o por partes)`, las cuales son funciones que no están definidas por una sola ecuación, sino que lo está por dos o mas, donde cada ecuación es válida para algún intervalo del dominio.

Debido a que nuestro grafos actualmente se representan con funciones, podemos hacer uso de este concepto y definirnos grafos de la siguiente manera:

`Ejemplo.` Si queremos definir un grafo donde los partes se apunten entre si y los impares entre si.



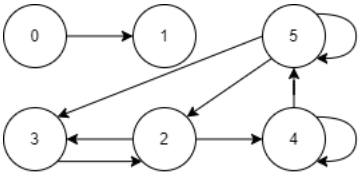
$\vartheta_{eo} : \mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})$

$$\vartheta_{eo}(v) = \begin{cases} \{e \mid e \in V \ \wedge \ e = \dot{2}\} & si\ v = \dot{2} \\ \{o \mid o \in V \ \wedge \ o \neq \dot{2}\} & si\ no \end{cases}$$

```
geo :: G
geo v
    | even v = [e | e <- vertices geo,  even v]
    | otherwise = [x | x <- vertices geo,  odd x]
```

4. Grafos finitos.

Hasta ahora solo vimos representaciones de grafos infinitos, pero recordemos que si bien en `Haskell` estamos definiendo a los vértices de un grafo como el conjunto de todos los naturales, en realidad la definición dada de grafos admite como dominio cualquier conjunto.
Intentemos representar al siguiente grafo:



Empecemos por aclarar que notaremos $\perp$ para señalar la ausencia de un valor. Y otro detalle es que al momento de declarar funciones definidas para un subconjunto de elementos incluido estricto en el dominio, o bien podemos acotar el dominio a ese subconjunto para que la función sea total o bien podemos declararla como una función parcial.

$\vartheta_{fin} : \mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})$

$$\vartheta_{fin}(v) = \begin{cases} \{1\} & si\ v = 0 \\ \{\} & si\ v = 1 \\ \{3,4\} & si\ v = 2 \\ \{2\} & si\ v = 3 \\ \{4,5\} & si\ v = 4 \\ \{2,3,5\} & si\ v = 5 \\ \perp & si\ v > 5 \end{cases}$$

Ahora ¿cómo proseguimos en `Haskell`? Pues como es de esperarse, es sencillo hacerlo en haskell también.

```
gfin :: G
gfin 0 = [1]
gfin 1 = []
```

```
gfin 2 = [3,4]
gfin 3 = [2]
gfin 4 = [4,5]
gfin 5 = [2,3,5]
gfin _ = undefined
```

5. Grafos finitos y acotados. Para representar grafos finitos donde sea deseable conocer una cota para evitar acceder a vértices que no estén definidos en el grafo, podríamos definir los grafos de una manera un tanto diferente.

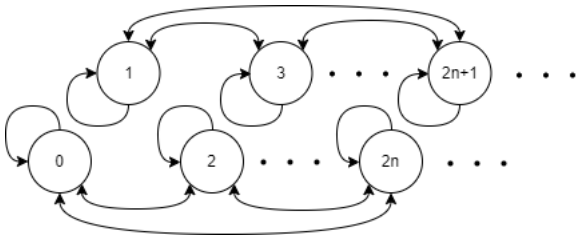
Sea $\hat{\vartheta}$ un grafo, éste se define como una pareja que dado un grafo ϑ y un natural n, se consideran solo los primeros n vértices de ϑ, de forma tal que: $\hat{\vartheta} : (\mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})) \times \mathbb{N}$

Estos grafos se pueden definir con las funciones que ya conocíamos, y un natural arbitrario:

$\hat{\vartheta} : (\mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})) \times \mathbb{N}$
$\hat{\vartheta} = (\vartheta, n) = \begin{cases} f(v) = \vartheta\ v & si\ v < n \\ \bot & si\ no \end{cases}$

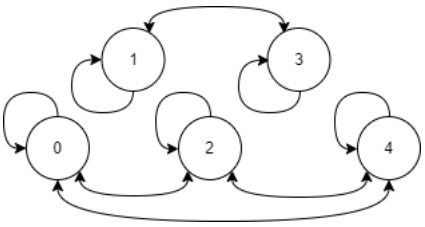
Con esta representación para los grafos finitos y acotados, podemos seguir definiendo los grafos como hasta ahora, pero limitar el dominio para poder representar cosas finitas.

`Ejemplo.` Recordemos el grafo que conectaba los pares entre si, y los impares entre si:



$\vartheta_{eo} : \mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})$
$\vartheta_{eo}(v) = \begin{cases} \{e \mid e \in V\ \wedge\ e = \dot{2}\} & si\ v = \dot{2} \\ \{o \mid o \in V\ \wedge\ o \neq \dot{2}\} & si\ no \end{cases}$

ahora podremos definir el grafo que cumpla estas características pero solo esté definido para los primeros 5 vértices de la siguiente forma:



$\hat{\vartheta}_{eo5} : (\mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})) \times \mathbb{N}$
$\hat{\vartheta}_{eo5} = (\vartheta eo, 5) = \begin{cases} f(v) = \{e \mid e \in V\ \wedge\ e = \dot{2}\} & si\ v = \dot{2} \wedge v < 5 \\ f'(v) = \{o \mid o \in V\ \wedge\ o \neq \dot{2}\} & si\ v \neq \dot{2} \wedge v < 5 \end{cases}$

Es momento de implementar en `Haskell` esta nueva representación de grafos.

```
type Gn = (G,N) -> G
```

```
-- donde recordemos que: type G = V -> Conj V

gn :: Gn
gn (g,n) = \v -> case v < n of {
    True -> g v;
    False -> undefined
}

geo :: G
geo v
    | even v = [e | e <- vertices geo,  even v]
    | otherwise = [x | x <- vertices geo,  odd x]

geo5 :: G
geo5 = gn (geo, 5)
```

6. Grafos ponderados.

Podemos entender la noción de un grafo ponderado, como aquella en donde las aristas del mismo poseen un costo, este puede tomar valores positivos o negativos, por simplificar asumiremos naturales, para la implementación de algoritmos como 'hallar el árbol de cubrimiento mínimo', o 'el camino mas barato entre un par de vértices'. Y para ello es cómodo y usual no contemplar costos negativos para evitar ciclos, ni tener que colocar restricciones como no poder repetir aristas, cuando eventualmente pueda ser necesario.

La pregunta que da lugar a la discusión nos surgirá ahora es ¿Cómo quéremos representar los grafos ponderados?, usualmente este costo se almacena en las aristas, y nuestros grafos son funciones que dado un vértice obtenemos los vértices adyacentes al mismo, entonces una primera aproximación sería, que en lugar de retornarme un conjunto de vértices adyacentes, que nos retorne un conjunto de parejas, conformadas con el vértice adyacente, y el cósto de llegar al mismo. De esta forma podríamos incluso fácilmente representar los multigrafos, permitiendo que el conjunto tenga parejas cuyos vértices adyacentes sean el mismo, pero el costo sea diferente.

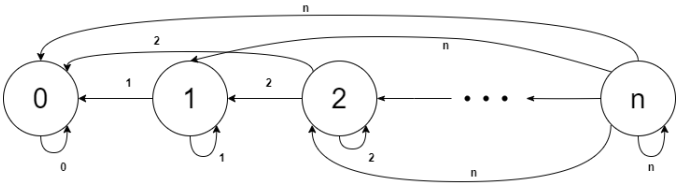
Pero como en esta representación de grafos lo que estamos intentando es fomentar/priorizar la *abstracción* por sobre las *definiciones estructurales*, propongo que los grafos ponderados (así como lo hicimos con los grafos finitos/acotados) como una pareja conformada por el `grafo tradicional(`ϑ`)` y una `función de costos`, que dada una arista, me retorne el costo de dicha arista en ϑ.
Representándose de la siguiente manera, $\overset{\mapsto}{\vartheta} : (\mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})) \times (\mathbb{A} \nrightarrow \mathbb{N})$.

Y pudiendo definir grafos tales como:

$\overset{k}{\underset{\vartheta_{leq}}{\mapsto}} : (\mathbb{V} \nrightarrow \mathcal{P}(\mathbb{V})) \times (\mathbb{A} \nrightarrow \mathbb{N})$
$\overset{k}{\underset{\vartheta_{leq}}{\mapsto}} = (\vartheta_{leq}, k) \ / \ k(o,d) = o$

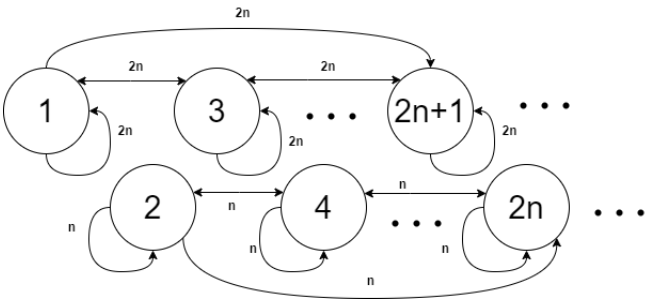
Como podemos observar y al igual que en los grafos finitos/acotados, podemos definir los ponderados utilizando los grafos que ya definíamos originalmente solo, que adicionalmente (en este caso ϑ_{leq}) agregamos una función `k` a esta pareja `grafo-función de costo`, donde esta función `k` está definida como dada una arista (una pareja vértice origen, vértice destino), nos retorna el origen, por lo que este grafo, es uno que para cada arista del grafo, su costo es igual al 'nombre' del vértice de origen de la arista, por lo que se podría

representar gráficamente de la siguiente forma:



Por supuesto también podríamos definirnos grafos cuyas aristas tengan costo según ciertas condiciones como podría ser:

$\overset{f_n}{\mapsto}$
$\vartheta_{\vartheta_{eo}} : (\mathbb{V} \not\to \mathcal{P}(\mathbb{V})) \times (\mathbb{A} \not\to \mathbb{N})$
$\overset{f_n}{\mapsto}$
$\vartheta_{\vartheta_{eo}} = (\vartheta_{eo}, f_n) \ / \ f_n(o,d) = \begin{cases} n & si\ o = \dot{2} \\ 2n & si\ o \neq \dot{2} \end{cases}$



Como podemos notar este grafo es el que ya habiamos definido con anteroridad, donde los vértices pares se conectan con todos los pares, y los impares con todos los impares (ϑ_{eo}). Ahora agregamos una función de costo, conviertiendolo en un grafo ponderado, donde las aristas entre los vértices impares cuestan el doble, que las aristas entre los vértices pares.

7. Definición de funciones. Luego de todas estas definiciones, uno podría preguntarse como podríamos programar funciones que involucren grafos, y eventualmente implementar los algoritmos más conocidos.

Procederémos a enlistar un compendio de funciones de grafos, que nos ayudarán a empezar a reconocer nociones sobre las funciones que involucran a los mismos, y de hecho algunas podían servirnos como auxiliares a utilizar mas adelante.

- `existeArista :: A -> G -> Bool`.
- `tieneLazoV :: V -> G -> Bool`.
- `esAislado :: V -> G -> Bool`.
- `existeSimetrica :: A -> G -> Bool`.
- `gradoEntrada :: V -> G -> Bool`.
- `gradoSalida :: V -> G -> Bool`.
- `gradoV :: V -> G -> Bool`.
- `vertices :: G -> [V]`.
- `aristas :: G -> [A]`.

- `grado :: G -> Bool.`
- `listaDeAdyacencia :: G -> [(V,[V])].`
- `matrizDeAdyacencia :: G -> [[Bool]].`
- `parVA :: G -> ([V],[A]).`
- `agregarA :: A -> G -> G.`
- `sacarA :: A -> G -> G.`
- `esSubgrafo :: G -> G -> Bool.`
- `sonComplementarios :: G -> G -> Bool.`
- `complementario :: G -> G.`
- `esCamino :: [V] -> G -> [A].`
- `clausuraSimetrica :: G -> G.`
- `clausuraTransitiva :: G -> G.`
- `esCiclico :: G -> Bool.`
- `esConexo :: G -> Bool.`
- `ordenTopologico :: G -> [V].`
- `dfs :: G -> [V].`
- `bfs :: G -> [V].`
- `dijkstra :: V -> V -> GP -> Int.`
- `dijkstra' :: V -> GP -> [(V,Int)].`
- `dijkstra'' :: V -> GP -> [(V,V)].`
- `dijkstra''' :: V -> GP -> [(V,V,C)].`
- `kruskal :: G -> [V].`