Teoría de grafos - Algoritmos

November 18, 2022

Recapitulando

Definición Un grafo $G = (V, E)$ consiste de un conjunto V de vértices o nodos y un conjunto E de aristas tal que cada arista $e \in E$ se asocia con un par no ordenado de vértices. Normalmente, si e es una arista asociada a los nodos v y w, escribimos simplemente $e = (v, w)$ o $e = (w, v)$, o más simple aún, (v, w). En este contexto, es importante notar que (v, w) y (w, v) son exactamente la misma arista.

Recapitulando

Si $e = (v, w)$ decimos que la arista e es *incidente* sobre v y w, que v y w son incidentes en e y además diremos que los vértices v y w son *adyacentes*.

Ayudamemoria Siempre que hablamos de incidencia estamos hablando de una relación entre un vértice y una arista, y siempre que hablamos de una relación de adyacencia, estamos hablando de una relación entre dos vértices. Cuando dos vértices son adyacentes, podemos decir coloquialmente que son **vecinos**.

Recapitulando

Definición Un **grafo con pesos** $G = (V, E)$ consiste de un conjunto V de vértices o nodos y un conjunto E de aristas tal que cada arista $e \in E$ se asocia con un par no ordenado de vértices. Adicionalmente, a cada arista le asignamos un número, que conocemos como el **peso** de la arista.

Recapitulando

Definición El grafo completo de n vértices, denotado por K_n es el grafo con n vértices en la que hay una arista entre cada para de vértices distintos.

Recapitulando

Se dice que un grafo es *bipartito* si existen subconjuntos V_1 y V_2 de V tales que $V_1 \cap V_2 = \emptyset$, $V_1 \cup V_2 = V$ y cada arista $e \in E$ es incidente sobre un vértice de V_1 y sobre un vértice en V_2.
El grafo bipartito completo de m y n vértices, que notamos $K_{m,n}$ es el grafo donde el conjunto de vértices tiene una partición V_1 con m vértices y V_2 con n vértices y donde el conjunto de aristas consiste de **todas** las aristas de la forma (v_1, v_2) con $v_1 \in V_1$ y $v_2 \in V_2$.

Recapitulando

Un grafo conexo es un grafo del cual podemos ir de cualquier vértice a cualquier otro por un camino, es decir, si dado cualquier par de vértices v y w en G, siempre existe un camino de v a w.
Intuitivamente, un grafo conexo es "de una sola pieza", mientras que un grafo no conexo tiene "varias partes". Cada una de estas partes se llama **componente conexo** del grafo.

Recapitulando

Sea $G = (V, E)$ un grafo. $G' = (V', E')$ es un subgrafo de G si:

1. $V' \subseteq V$ y $E' \subseteq E$
2. Para toda arista $e' \in E'$, si e' incide en v' y w', entonces $v', w' \in V'$

Recapitulando

Un camino es una sucesión de vértices y aristas dentro de un grafo, que empieza y termina en vértices, tal que cada vértice es incidente en las aristas que le siguen y que le preceden en la secuencia.
Dos vértices están conectados o son accesibles si existe un camino que forma una trayectoria para llegar de uno al otro; en caso contrario, los vértices están desconectados o bien son inaccesible
Un ciclo es un camino de longitud diferente de cero que empieza y termina en el mismo vértice.
Nota Por lo general, estamos interesados en caminos y ciclos **simples** es decir, que no repiten aristas.
En un grafo sin pesos, la longitud de un camino es la cantidad de aristas por las que pasamos. En un grafo con pesos, la longitud del camino es la suma de los pesos de las aristas por las que pasamos.

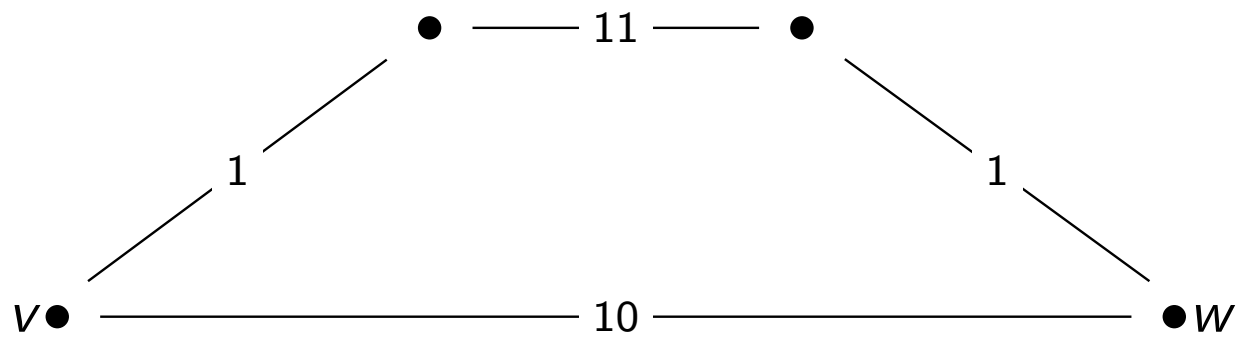
Recapitulando

El **grado de un vértice** v, que denotamos como $\delta(v)$ es el número de aristas que inciden en v.

Un algoritmo para la ruta mas corta

Un grafo con pesos es un grafo donde asignamos a cada arista un valor numérico.
En un grafo con pesos que la longitud de un camino es la suma de los pesos de las aristas que componen el camino.
Notaremos $w(i, j)$ el peso de la arista (i, j). En un grafo con pesos un problema muy común es encontrar el camino más corto entre dos vértices (es decir, un camino que tiene la longitud mínima entre todas los posibles caminos entre esos dos vértices.)

Probando greedy...



¿Cual es el camino mas corto entre v y w?
Un algoritmo greedy consistiría en comenzar en v y elegir en cada paso la arista de menor valor que nos "acerque" a w. Sin embargo, el algoritmo no es óptimo: Para el grafo de la figura elige un camino de longitud 12, cuando hay un camino de longitud 10 válido.

Edsger Dijkstra

Edsger W. Dijkstra (1930-2002) fue un programador holandés que ideo un algoritmo que resuelve el problema de la ruta más corta de forma óptima. Además, fue de los primeros en proponer la programación como una ciencia.

El algoritmo de Dijkstra

Este algoritmo encuentra la longitud de una ruta más corta del vértice a al vértice z en un grafo $G = (V, E)$ con pesos conexo. El peso de la arista (i, j) es $w(i, j) > 0$, y la **etiqueta** del vértice x es $L(x)$. Al terminar, $L(z)$ es la longitud de la ruta más corta de a a z.

El algoritmo de Dijkstra

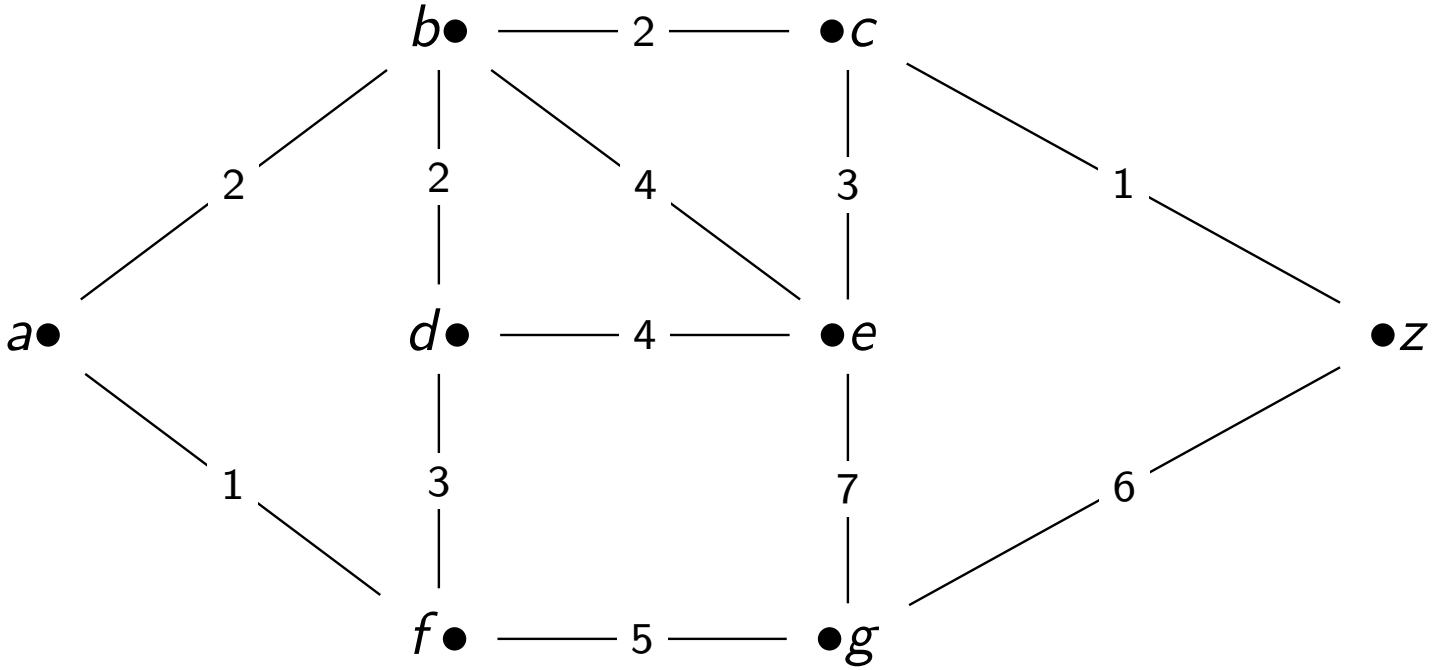
1. Primero, notemos que la ruta más corta del vértice a al vértice a, es la ruta de cero aristas, que tiene peso 0. Así, inicializamos $L(a) = 0$.
2. No sabemos aún el valor de una ruta más corta de a a los otros vértices, entonces para cada vértice $v \neq a$, inicializamos $L(v) = \infty$.
3. Inicializamos el conjunto T como el conjunto de todos los vértices, i.e. $T = V$.
4. Seleccionamos un vértice $v \in T$ tal que $L(v)$ sea mínimo.
5. Quitamos el vértice v del conjunto T: $T = T - v$
6. Para cada $w \in T$ adyacente a w, actualizamos su etiqueta: $L(w) = \text{MIN}\{L(w), L(v) + w(v, w)\}$
7. Si $z \in T$, repetimos desde el paso 4, si no, hemos terminado y $L(z)$ es el valor de la ruta mas corta entre a y z.

El algoritmo de Dijkstra

El algoritmo de Dijkstra para la ruta mas corta del vértice a al vértice z implica asignar etiquetas a los vértices. Sea $L(v)$ la etiqueta del vértice v. En cualquier punto, algunos vértices tienen etiquetas temporales y el resto son permanentes. Sea T el conjunto de vértices que tienen etiquetas temporales. Al ilustrar el algoritmo, normalmente remarcamos con un circulo los vértices que tienen etiquetas temporales. Si $L(v)$ es la etiqueta permanente del vértice v, entonces $L(v)$ es la longitud de una ruta más corta de a a v. Al inicio, todos los vértices tienen etiquetas temporales. Cada iteración del algoritmo cambia el estado de una etiqueta de temporal a permanente; entonces el algoritmo puede terminar cuando z recibe una etiqueta permanente.

El algoritmo Dijkstra - ejemplo

Utilizaremos el algoritmo de Dijkstra para encontrar el tamaño de la ruta más corta de a a z en el siguiente grafo:



Cuando corremos el algoritmo a mano, conviene utilizar una tabla para ir llevando como evolucionan los valores de las etiquetas L a medida que ocurren las iteraciones del algoritmo

Algoritmo de Dijkstra - ejemplo

Pondremos en la tabla una columna para llevar el número de iteración en el que nos encontramos y luego una columna por cada vértice. Un valor en la posición (i, j) en la tabla denotara el valor de la etiqueta L del vértice en la posición j, en la iteración i. Comenzamos escribiendo la primera fila, los valores con los que inicializamos L, para el vértice a, la etiqueta vale 0, y para todos los demás, infinito.

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞

Algoritmo de Dijkstra - ejemplo

Completamos la primera iteración del algoritmo, hay que elegir el vértice que tenga la etiqueta mínima y removerla del conjunto T. En la tabla consideraremos esto con una línea diagonal tachando ese casilla. En este paso, obviamente el valor menor es $L(a) == 0$. Luego actualizamos los valores de las etiquetas para los vértices adyacentes a a, que son los vértices b y f. Para el vértice b, el nuevo valor de la etiqueta es el menor entre la etiqueta que tenia, y lo que resultaría de sumar la etiqueta de a con el valor de la arista (a, b). Del mismo modo, procedemos con la etiqueta del vértice f.

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	???	∞	∞	∞	???	∞	∞

Algoritmo de Dijkstra - ejemplo

Completamos la primera iteración del algoritmo, hay que elegir el vértice que tenga la etiqueta mínima y removerla del conjunto T. En la tabla consideraremos esto con una línea diagonal tachando ese casilla. En este paso, obviamente el valor menor es $L(a) == 0$. Luego actualizamos los valores de las etiquetas para los vértices adyacentes a a, que son los vértices b y f. Para el vértice b, el nuevo valor de la etiqueta es el menor entre la etiqueta que tenia, y lo que resultaría de sumar la etiqueta de a con el valor de la arista (a, b). Del mismo modo, procedemos con la etiqueta del vértice f.

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	$\min\{\infty, 0 + 2\}$	∞	∞	∞	???	∞	∞

Algoritmo de Dijkstra - ejemplo

Completamos la primera iteración del algoritmo, hay que elegir el vértice que tenga la etiqueta mínima y removerla del conjunto T. En la tabla consideraremos esto con una línea diagonal tachando ese casilla. En este paso, obviamente el valor menor es $L(a) == 0$. Luego actualizamos los valores de las etiquetas para los vértices adyacentes a a, que son los vértices b y f. Para el vértice b, el nuevo valor de la etiqueta es el menor entre la etiqueta que tenia, y lo que resultaría de sumar la etiqueta de a con el valor de la arista (a, b). Del mismo modo, procedemos con la etiqueta del vértice f.

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	2	∞	∞	∞	???	∞	∞

Algoritmo de Dijkstra - ejemplo

Completamos la primera iteración del algoritmo, hay que elegir el vértice que tenga la etiqueta mínima y removerla del conjunto T. En la tabla consideraremos esto con una línea diagonal tachando ese casilla. En este paso, obviamente el valor menor es $L(a) == 0$. Luego actualizamos los valores de las etiquetas para los vértices adyacentes a a, que son los vértices b y f. Para el vértice b, el nuevo valor de la etiqueta es el menor entre la etiqueta que tenia, y lo que resultaría de sumar la etiqueta de a con el valor de la arista (a, b). Del mismo modo, procedemos con la etiqueta del vértice f.

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	2	∞	∞	∞	$\min\{ \infty, 0+1 \}$	∞	∞

Algoritmo de Dijkstra - ejemplo

Completamos la primera iteración del algoritmo, hay que elegir el vértice que tenga la etiqueta mínima y removerla del conjunto T. En la tabla consideraremos esto con una línea diagonal tachando ese casilla. En este paso, obviamente el valor menor es $L(a) == 0$. Luego actualizamos los valores de las etiquetas para los vértices adyacentes a a, que son los vértices b y f. Para el vértice b, el nuevo valor de la etiqueta es el menor entre la etiqueta que tenia, y lo que resultaría de sumar la etiqueta de a con el valor de la arista (a, b). Del mismo modo, procedemos con la etiqueta del vértice f.

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	2	∞	∞	∞	1	∞	∞

Algoritmo de Dijkstra - ejemplo

En la segunda iteración, elegimos el vértice f. Actualizamos los valores de los vértices d y g (no actualizamos el nodo a porque ya esta "tachado").

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	2	∞	∞	∞	1	∞	∞
2	/	2	∞	???	∞	/	???	∞

Algoritmo de Dijkstra - ejemplo

En la segunda iteración, elegimos el vértice f. Actualizamos los valores de los vértices d y g (no actualizamos el nodo a porque ya esta "tachado").

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	2	∞	∞	∞	1	∞	∞
2	/	2	∞	$\min\{ \infty, 1 + 3 \}$	∞	/	???	∞

Algoritmo de Dijkstra - ejemplo

En la segunda iteración, elegimos el vértice f. Actualizamos los valores de los vértices d y g (no actualizamos el nodo a porque ya esta "tachado").

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	2	∞	∞	∞	1	∞	∞
2	/	2	∞	4	∞	/	???	∞

Algoritmo de Dijkstra - ejemplo

En la segunda iteración, elegimos el vértice f. Actualizamos los valores de los vértices d y g (no actualizamos el nodo a porque ya esta "tachado").

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	2	∞	∞	∞	1	∞	∞
2	/	2	∞	4	∞	/	$\min\{ 1 + 5 \}$	∞

Algoritmo de Dijkstra - ejemplo

En la segunda iteración, elegimos el vértice f. Actualizamos los valores de los vértices d y g (no actualizamos el nodo a porque ya esta "tachado").

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	2	∞	∞	∞	1	∞	∞
2	/	2	∞	4	∞	/	6	∞

El algoritmo Dijkstra - ejemplo

Continuamos realizando las iteraciones del algoritmo hasta que conseguimos tachar el vértice z.

it	$L(a)$	$L(b)$	$L(c)$	$L(d)$	$L(e)$	$L(f)$	$L(g)$	$L(z)$
	0	∞	∞	∞	∞	∞	∞	∞
1	/	2	∞	∞	∞	1	∞	∞
2	/	2	∞	4	∞	/	6	∞
3	/	/	4	4	6	/	6	∞
4	/	/	/	4	6	/	6	5
5	/	/	/	/	6	/	6	5
6	/	/	/	/	6	/	6	/

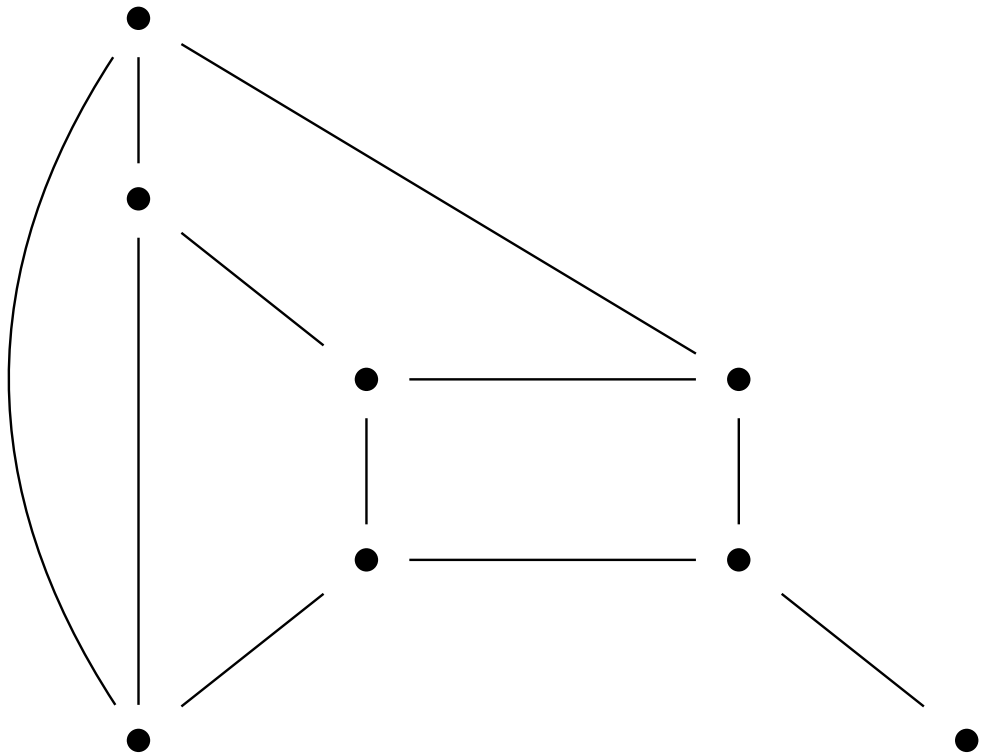
La longitud del camino más corto de a a z es 5.

Arboles de expansión

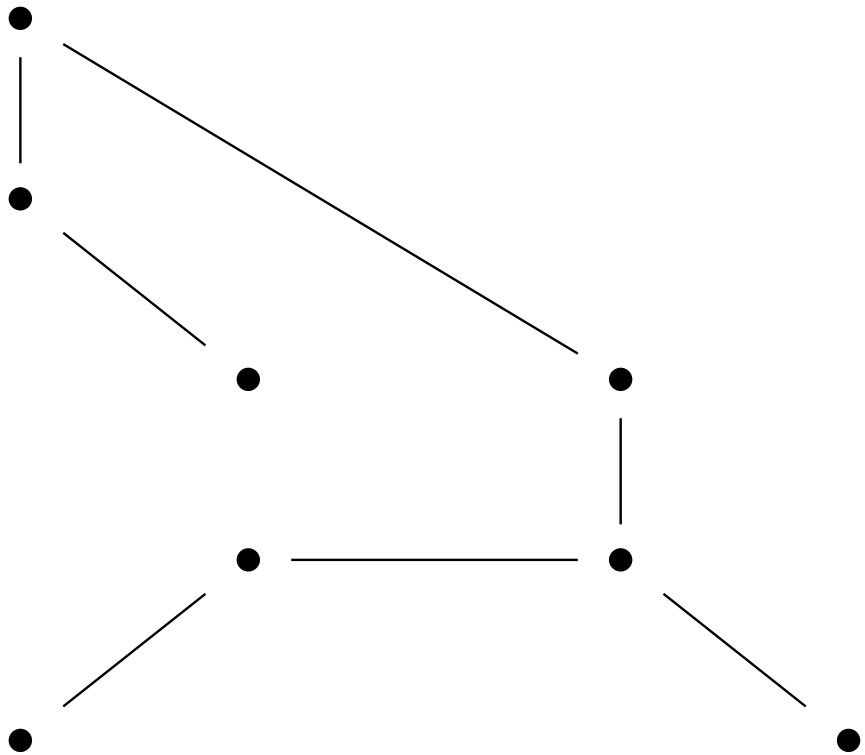
Se dice que un árbol T es un **árbol de expansión** de un grafo G al subgrafo tal que:

1. T es un árbol
2. T contiene todos los vértices de G.

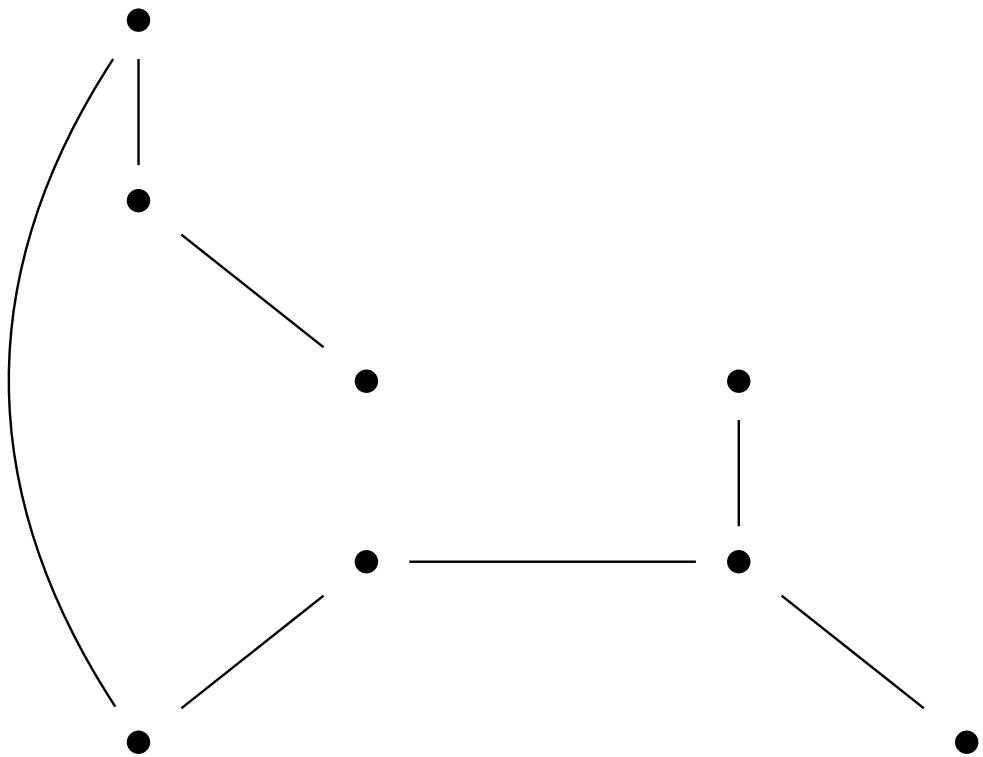
Por ejemplo, dado este grafo



Este es un arbol de expansion de ese grafo:



Nota El árbol de expansión de un grafo en general no es único, por ejemplo, acá mostramos otro árbol de expansión distinto del mismo grafo.



Propiedad

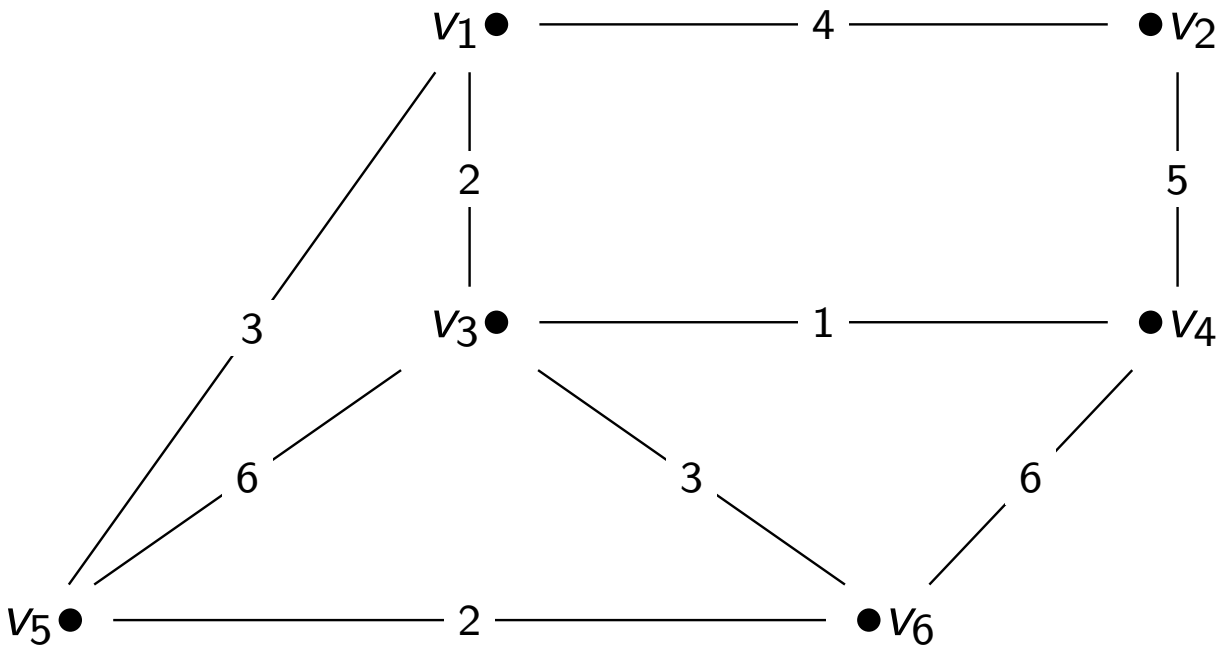
- Un árbol de expansión para un grafo existe si y solo si el grafo es conexo.

En efecto, todos los arboles pueden pensarse como grafos conexos sin ciclos. Si un grafo G tiene un árbol de expansión T, entonces entre dos vértices cualesquiera existe un camino en el árbol que los une (pues todos los arboles son conexos). Ahora bien, como T es subgrafo de G, necesariamente debe existir ese mismo camino en el grafo G.
El algoritmo de búsqueda en profundidad, normalmente llamado DFS por sus siglas en inglés *Depth First Search* permite encontrar el árbol de expansión de un grafo conexo. Dado un orden $v_1, v_2, ..., v_n$ de los vértices, el algoritmo procede como sigue:

1. Dado el grafo $G = (V, E)$, inicializamos el árbol $T = (V', E')$ con $V' = v_1$ y $E' = \emptyset$. La idea sera ir agregando aristas a E' a medida que lo necesitemos. Definimos además una variable $w = v_1$ que llevará el nodo donde estamos parados actualmente.
2. Mientras exista v tal que (w, v) es una arista que al agregarla a T no genera un ciclo, realizamos lo siguiente:
    1. Elegimos la arista (w, v_k) con k mínimo tal que al agregarla a T no genera un ciclo.
    2. Agregamos la arista (w, v_k) a E'.
    3. Agregamos v_k a V'
    4. Actualizamos $w = v_k$
3. Si $V' = V$ hemos terminado y T y T es un árbol de expansión del grafo G. Si $w = v_1$, el grafo es disconexo, y por lo tanto jamás podremos encontrar un árbol de expansión para el mismo. Si no se da ninguna de las dos situaciones, actualizamos el valor de w para que sea el padre de w en el árbol T, y repetimos desde el paso 2. Dar este paso hacia atrás nos obligará a explorar otros caminos.

Árboles de expansión mínima

El grado con pesos de la figura muestra seis ciudades y los costos de construir carreteras entre ellas. Se desea construir el sistema de carreteras de menor costo que conecte a las seis ciudades. La solución debe necesariamente ser un árbol de expansión ya que debe contener a todos los vértices y para ser de costo mínimo, seria redundante tener dos caminos entre ciudades. Entonces lo que necesitamos es el árbol de expansión del grafo que sea de peso mínimo.



Árboles de expansión mínima

Definición Sea G un grafo con pesos. Un árbol de expansión mínima de G es un árbol de expansión de G que tiene peso mínimo entre todos los posibles.

Nota El algoritmo DFS no asegura que el árbol encontrado sea de peso mínimo.

El algoritmo de Prim

El algoritmo de Prim permite encontrar un árbol de expansión mínimo para un grafo con pesos conexo de vértices $v_1, v_2, ...v_n$. Definimos $w(i, j)$ como el peso de la arista que une los vértices i, j si existe, o como ∞ si la misma no existe. Además, llevamos la cuenta en un diccionario *agregado* cuyas claves son los vértices y cuyas valores son *True* si el vértice fue agregado al árbol de expansión mínima, y *False* si aún no ha sido agregado. También iremos actualizando el diccionario E' de las aristas del árbol.

1. Inicializamos el diccionario *agregado*, seteando todos los vértices a *False* (es decir, ningún vértice ha sido agregado aún.)
2. Agregamos el primer vértice al árbol $agregado[v_1] = True$.
3. Inicializamos la lista de aristas que compondrán el árbol como un conjunto vacio: $E = \emptyset$.
4. Para cada i en el rango $1, ..., n-1$, agregamos la arista de peso mínimo que tiene un vértice que ya fue agregado, esto lo hacemos del siguiente modo:
    1. Definimos la variable temporal $min = \infty$.
    2. Para cada j en el rango $(1, ...n)$:
        1. Si $agregado[v_j] == True$, el vértice v_j ya esta en el árbol:
        2. Para cada k en el rango de $(1, ..., n)$:

            Si $agregado[v_k] == False$ y además $w(j,k) < min$, el vértice v_k será el *candidato* a ser agregado al árbol, y la arista (j,k) sera la arista candidata a agregar al árbol.
    3. Al finalizar el for, agregamos el vértice candidato al árbol actualizando el diccionario agregado, y además agregamos la arista candidata al conjunto de aristas.

Introducción a NetworkX

NetworkX es un paquete de Python para crear, manipular y estudiar la estructura de grafos complejos. Ya trae incluidos muchos algoritmos para grafos. Como ejemplo básico: acá vemos como crear un grafo:

```
 # casi todo el mundo importa networkx asi
import networkx as nx
G = nx.Graph()
```

El grafo G no contiene ningún nodo ni ninguna arista, veamos como agregarlas

Introducción a NetworkX

Podemos agregar nodos de a uno, por ejemplo:

```
G.add_node(1)
```

o de a varios a la vez

```
G.add_nodes_from([2, 3])
```

Bien! Ahora nuestro grafo tiene nodos, pero aún no tiene aristas, veamos como agregarlas:

Introducción a NetworkX

Podemos agregar aristas de a una, utilizando dos sintaxis distintas:

```
G.add_edge(1, 2)
# o bien
e = (2, 3)
G.add_edge(*e)
```

o de a varias a la vez

```
G.add_edges_from([(1, 2), (1, 3)])
```

Introducción a NetworkX

Podemos ver cuantos nodos y cuantas aristas tiene nuestro grafo utilizando los métodos asociados:

```
G.number_of_nodes() # 3
G.number_of_edges() # 3
```

También podemos obtener la lista completa de nodos y de aristas que contiene un grafo:

```
list(G.nodes)  # [1, 2, 3]
list(G.edges)  # [(1,2), (1, 3), (2, 3)]
```

Con el atributo `degree` podemos obtener un diccionario donde las claves son los vértices y los valores asociados son el grado de cada uno de los vértices:

```
dict(G.degree)  # { 1: 2, 2: 2, 3: 3 }
```

Introducción a NetworkX

Del mismo modo que agregamos nodos y aristas, podemos removerlos, utilizando las funciones apropiadas:

```
G.remove_node(2)
G.remove_nodes_from([1,3])
G.remove_edge(1, 3)
```

Introducción a NetworkX

Si queremos que nuestros grafos tengan peso en las aristas, utilizamos el parámetro especial `weight` al momento de agregarla

```
G.add_edge(1, 2, weight=4.7 )
```

Introducción a NetworkX

Una vez que tenemos el grafo listo, podemos empezar a trabajarlo. Por ejemplo, podemos pedirle a NetworkX que analice sus componentes conexas:

```
G = nx.Graph()
G.add_nodes_from([1, 2, 3])
G.add_edges_from([(1, 2), (1, 3)])
G.add_node("spam")        # adds node "spam"
len(list(nx.connected_components(G))) # 2
```

Introducción a NetworkX

Podemos también dibujar el grafo con la ayuda del paquete `matplotlib`

```
import matplotlib.pyplot as plt
G = nx.Graph()
G.add_nodes_from([1, 2, 3])
G.add_edges_from([(1, 2), (1, 3)])
G.add_node("spam")        # adds node "spam"
nx.draw(G, with_labels=True, font_weight='bold')
```

Si necesitamos dibujar grafos con peso, utilizamos la siguiente receta:

```
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, font_weight='bold')
edge_labels = dict([
    ((n1, n2), d['weight'])
    for n1, n2, d in G.edges(data=True)
    ])

nx.draw_networkx_edge_labels(G, pos=pos,
   edge_labels=edge_labels)
```

Podemos encontrar la longitud del camino mas corto entre dos vértices utilizando el método `shortest_path_length` y un camino de esa longitud (expresado como una secuencia de vértices) con el método `shortest_path`.

Nota Si en los métodos omitimos el parámetro `weight`, interpretará que todos los pesos son iguales a 1. Para que tome los pesos que asignamos a las aristas, debemos pasar explícitamente `weight="weight"`

```
G = nx.Graph()
G.add_nodes_from("abcdefghijz")
G.add_edge("a", "b", weight=4)
G.add_edge("b", "c", weight=1)
G.add_edge("c", "d", weight=6)
G.add_edge("b", "e", weight=6)
G.add_edge("b", "f", weight=4)
G.add_edge("c", "f", weight=3)
G.add_edge("d", "z", weight=1)
G.add_edge("a", "e", weight=1)
G.add_edge("f", "e", weight=6)
G.add_edge("f", "g", weight=5)
G.add_edge("g", "h", weight=1)
G.add_edge("a", "i", weight=6)
G.add_edge("e", "j", weight=8)
print(nx.shortest_path_length(
    G, source="a", target="z", weight="weight"
))
print(nx.shortest_path(
    G, source="a", target="z", weight="weight"
))
```

Podemos encontrar un árbol de expansión utilizando el algoritmo DFS mediante el método `dfs_tree`.

```
G = nx.Graph()
G.add_nodes_from("abcdef")
G.add_edge("a", "b", weight=4)
G.add_edge("b", "c", weight=1)
G.add_edge("c", "d", weight=6)
G.add_edge("b", "e", weight=6)
G.add_edge("b", "f", weight=4)
G.add_edge("c", "f", weight=3)
G.add_edge("d", "a", weight=1)
G.add_edge("a", "e", weight=1)
G.add_edge("f", "e", weight=6)
G.add_edge("f", "b", weight=5)
G.add_edge("c", "d", weight=1)
G.add_edge("a", "e", weight=6)
G.add_edge("e", "f", weight=8)
T = nx.dfs_tree(G, source='a')
nx.draw(T)
```

Podemos encontrar el árbol de expansión mínima utilizando el método `minimum_spanning_tree`

```
G = nx.Graph()
G.add_nodes_from("abcdef")
G.add_edge("a", "b", weight=4)
G.add_edge("b", "c", weight=1)
G.add_edge("c", "d", weight=6)
G.add_edge("b", "e", weight=6)
G.add_edge("b", "f", weight=4)
G.add_edge("c", "f", weight=3)
G.add_edge("d", "a", weight=1)
G.add_edge("a", "e", weight=1)
G.add_edge("f", "e", weight=6)
G.add_edge("f", "b", weight=5)
G.add_edge("c", "d", weight=1)
G.add_edge("a", "e", weight=6)
G.add_edge("e", "f", weight=8)
T = nx.minimum_spanning_tree(G)
# El peso del arbol T se puede consultar
# con el metodo size
print(T.size(weight="weight"))
```

Referencias

Tutorial oficial de NetworkX

https://networkx.org/documentation/stable/tutorial.html

Lectura recomendada

Johnsonbaugh

Matemáticas discretas. 6ta Edición.

Capítulos 8.5, 9.3 y 9.4

Grimaldi, R.

Matemáticas Discretas y Combinatoria.

Advertencia La terminología asociada a la teoría de grafos no se ha estandarizado aún. Al leer artículos y libros sobre grafos, es necesario verificar las definiciones que se emplean. Ante cualquier duda, consultar con los docentes de la cátedra.