Aplicaciones de búsqueda por anchura

Introducción

En la entrada anterior hablamos de búsqueda por anchura. Vimos en qué consiste y cómo se implementa. Lo que haremos ahora es ver cómo podemos utilizar la idea de buscar por anchura para resolver algunos problemas en teoría de gráficas. Daremos aplicaciones en la determinación de caminos cortos de un vértice a otro, en la determinación de los componentes conexos de una gráfica y en un problema aplicado de juntar los votos de una elección en una central electoral.

Distancia y caminos cortos

Tomemos una gráfica conexa G y dos vértices u y v. Supongamos que queremos encontrar la distancia de v a u de manera algorítmica. Una forma de resolver esto es hacer una búsqueda por anchura que comience en v. Es sencillo ver que si T es el árbol por anchura asociado, entonces la distancia buscada es la cantidad de veces que debemos aplicar la función `padre` a u para llegar a v.

Esta idea nos lleva a la siguiente implementación.

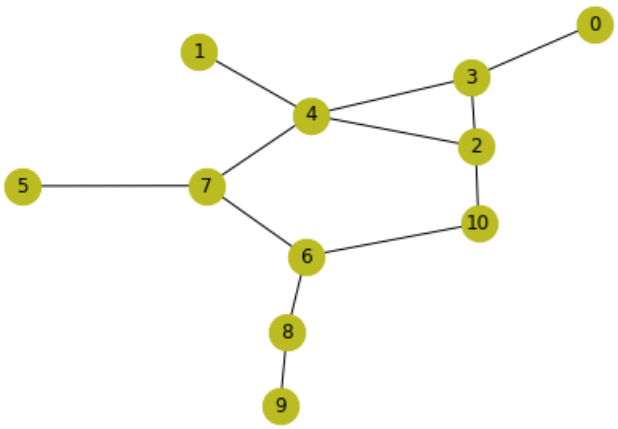
```
import networkx as nx

def distancia(G,v,u):
    if v==u:
        return 0
    encontrados={v}
    distancias={v:0}
    procesados=set()
    en_proceso=[v]
    while en_proceso:
        w=en_proceso[0]
        for z in G.neighbors(w):
            if (z not in procesados) and (z not in encontrados):
                encontrados.add(z)
                en_proceso.append(z)
                distancias[z]=distancias[w]+1
                if z==u:
                    return(distancias[z])
        en_proceso.remove(w)
        procesados.add(w)
    return "No están conectados, la gráfica no es conexa."

G = nx.Graph()
G.add_edges_from([(0,3), (1,4), (7,4), (10,2), (2,3), (2,4), (3,4), (6,7), (5,7), (6,8), (8,9), (6,10)])
nx.draw_kamada_kawai(G,with_labels=True, node_color='#bbbb22',node_size=500)

print(distancia(G,1,10))
print(distancia(G,5,0))
print(distancia(G,0,4))
```

```
3
4
2
```



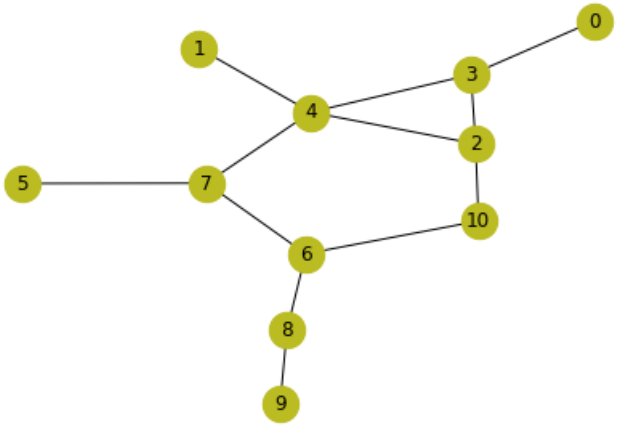
Si en el algoritmo anterior llevamos registro de los padres de cada vértice, entonces podemos no sólo determinar la distancia de \(u\) a \(v\), sino además encontrar un camino más corto entre ellos.

```
## Este calcula explícitamente uno de los caminos más cortos de v a u
def camino_corto(G,v,u):
    if v==u:
        return [u]
    encontrados={v}
    procesados=set()
    padres={v:None}
    en_proceso=[v]
    while en_proceso:
        w=en_proceso[0]
        for z in G.neighbors(w):
            if (z not in procesados) and (z not in encontrados):
                encontrados.add(z)
                en_proceso.append(z)
                padres[z]=w
                if z==u:
                    camino=[z]
                    padre=z
                    while padre!=v:
                        padre=padres[padre]
                        camino=[padre]+camino
                    return camino
        en_proceso.remove(w)
        procesados.add(w)
    return "No están conectados"

nx.draw_kamada_kawai(G,with_labels=True, node_color='#bbbb22',node_size=500)

print(camino_corto(G,1,10))
print(camino_corto(G,5,0))
print(camino_corto(G,0,4))
```

```
[1, 4, 2, 10]
[5, 7, 4, 3, 0]
[0, 3, 4]
```



Componentes conexas

Hasta ahora nuestras búsquedas por anchura sólo nos han permitido explorar por completo gráficas conexas. En realidad esto no es un problema. Si tras terminar una búsqueda por anchura aún no hemos procesado a todos los vértices de la gráfica, entonces podemos iniciar una nueva búsqueda por anchura en alguno de los vértices no visitados.

Cada que iniciemos una búsqueda por anchura en un vértice \(v\), la búsqueda encontrará a todos los vértices en su componente conexa.

Estas ideas nos pueden ayudar a resolver problemas relacionados con las componentes conexas de una gráfica, por ejemplo:

- Decidir si una gráfica es conexa o no.
- Contar el número de componentes conexas.
- Enlistar las componentes conexas de una gráfica.

- Encontrar la componente conexa con la mayor/menor cantidad de vértices/aristas.

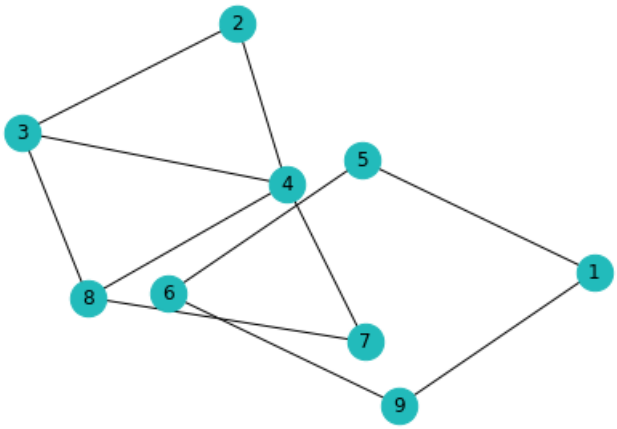
Comencemos viendo cómo adaptar las ideas de búsqueda por anchura para contar las componentes conexas de una gráfica. Para ello, basta llevar una cuenta de cuántas búsquedas por anchura tuvimos que hacer. La implementación se muestra a continuación.

```
def cont_cc(G):
    # Estas variables se definen globalmente, para todas las búsquedas.
    encontrados=set()
    procesados=set()
    en_proceso=[]
    num_cc=0
    for v0 in G.nodes():
        # Hacemos lo siguiente para no hacer búsquedas por anchura de más
        if v0 in procesados:
            continue
        else:
            num_cc+=1 # Esto es lo que va llevando la cuenta.
            en_proceso=[v0]
            encontrados.add(v0)
            while en_proceso:
                v=en_proceso[0]
                for w in G.neighbors(v):
                    if (w not in procesados) and (w not in encontrados):
                        encontrados.add(w)
                        en_proceso.append(w)
                en_proceso.remove(v)
                procesados.add(v)
    return(num_cc)

G = nx.Graph()
G.add_edges_from([(1,5),(2,4),(3,8),(5,6),(1,9),(4,7),(2,3),(7,8),(4,8),(3,4),(6,9)])
nx.draw_kamada_kawai(G,with_labels=True, node_color='#22bbbb',node_size=500)

print("Hay {} componentes conexas".format(cont_cc(G)))
```

```
Hay 2 componentes conexas
```



Una vez más, un ligero cambio nos puede ayudar a tener mucha más información. Además de saber cuántas componentes conexas hay, también podemos llevar un registro de cuáles son.

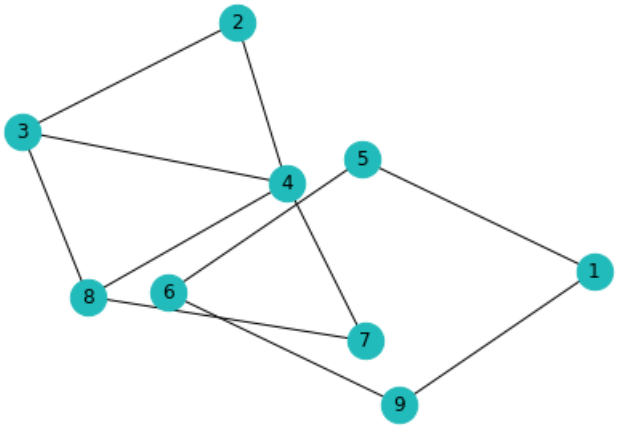
```
def enlistar_cc(G):
    encontrados=set()
    procesados=set()
    en_proceso=[]
    lista_cc=[] # Aquí iremos almacenando componentes conexas.
    for v0 in G.nodes():
        if v0 in procesados:
            continue
        else:
            nueva_cc={v0}
            en_proceso=[v0]
            encontrados.add(v0)
            while en_proceso:
                v=en_proceso[0]
                for w in G.neighbors(v):
                    if (w not in procesados) and (w not in encontrados):
                        nueva_cc.add(w)
                        encontrados.add(w)
                        en_proceso.append(w)
                en_proceso.remove(v)
                procesados.add(v)
            lista_cc.append(nueva_cc)
    return(lista_cc)

G = nx.Graph()
```

```
G.add_edges_from([(1,5),(2,4),(3,8),(5,6),(1,9),(4,7),(2,3),(7,8),(4,8),(3,4),(6,9)])
nx.draw_kamada_kawai(G,with_labels=True, node_color='#22bbbb',node_size=500)

print("Las compontentes conexas son:")
for j in enlistar_cc(G):
    print(j)
```

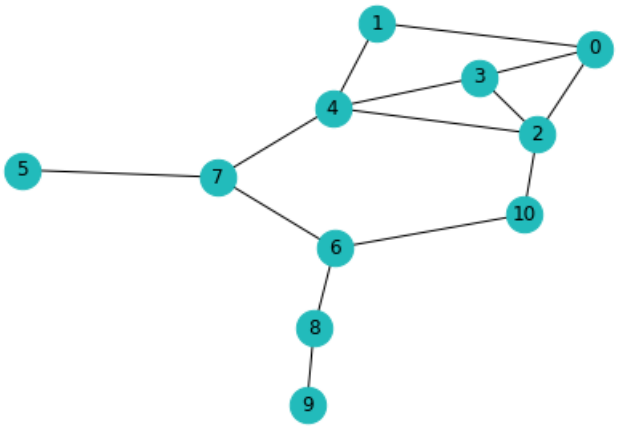
```
Las compontentes conexas son:
{1, 5, 9, 6}
{2, 3, 4, 7, 8}
```



Llevando votos a una central electoral

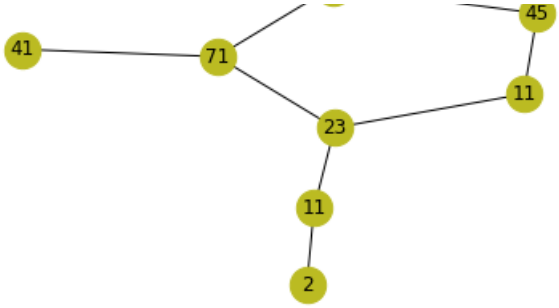
Veamos un problema más interesante, motivado por una aplicación. Supongamos que acaba de terminar una jornada electoral. Hubo votaciones en varias casillas electorales conectadas por caminos de acuerdo a la siguiente gráfica \(G\).

```
G = nx.Graph()
G.add_nodes_from([(0,{'votos':58}),(1,{'votos':66}),(2,{'votos':45}),(3,{'votos':36}),(4,{'votos':16}),(5,{'votos':41}),(6,{'votc
G.add_edges_from([(0,1), (0,2), (0,3), (1,4), (7,4), (10,2), (2,3), (2,4), (3,4), (6,7), (5,7), (6,8), (8,9), (6,10)])
nx.draw_kamada_kawai(G,with_labels=True, node_color='#22bbbb',node_size=500)
KKL = nx.kamada_kawai_layout(G)
```



El vértice \(4\) está designado como la **central electoral**. A ella tienen que llegar todos los votos. La cantidad de votos en cada casilla se muestra en la siguiente figura.

```
picture=nx.draw_kamada_kawai(G,labels=dict(G.nodes(data='votos')), node_color='#bbbb22',node_size=500)
```



En cada vértice hay una moto que puede usarse, o no. Cada moto tarda una hora en recorrer una arista y gasta 2 litros de gasolina en hacerlo. Una moto puede recoger tantos votos como quiera, de tantas casillas como quiera. Se vale pasar votos de una moto a otra.

- ¿Cuál es la mínima cantidad de tiempo que se necesita para llevar todos los votos a la central electoral?
- Para esa cantidad de tiempo mínima, ¿cuál es la mínima cantidad de gasolina que se tienen que usar?
- En una repartición óptima, ¿cuántos votos debe esperar recibir la central electoral de las motos que lleguen de las casillas vecinas \(1,2,3,7\)?

Una manera de hacer la recolección es que se use una sola moto que pase por todas las casillas. Esto cumple con el objetivo de llevar todos los votos, pero probablemente tarde mucho tiempo y use mucha gasolina por pasar varias veces por aristas.

Otra manera de hacer la recolección es que se usen todas las motos y que cada una lleve sus votos a la central electoral. Esto seguro es óptimo en tiempo, pero usa mucha gasolina: si una moto saldrá de la casilla \(9\), entonces podría recoger también los votos de las casillas \(8\), \(6\) y \(7\). Se tendría que avisar que esa moto lleva \(2+11+23+71=107\) votos. Se ahorraría toda la gasolina de las motos de las casillas \(8,6,7\).

¿Cómo le podemos hacer para tardarnos lo menos posible con la mínima cantidad de motos? Usaremos búsqueda por anchura de la siguiente manera.

- Hacemos una búsqueda por anchura en la gráfica.
- Usaremos únicamente las motos de los vértices que no hayan descubierto nuevos vértices.
- Le diremos a cada moto que lleve los votos de un vértice a su padre. Ya que a un vértice lleguen todos los votos de sus hijos, apuntamos cuántos deben ser. Los ponemos en una misma moto para mandarlos a su padre, y así sucesivamente.

La menor cantidad en horas es por lo menos la distancia de la casilla más lejana a la central electoral. Es decir, las \(4\) horas que se tarda en ir de la casilla \(9\) a la casilla \(4\). El algoritmo anterior termina en menos de \(4\) horas pues se puede mostrar inductivamente que a la hora \(j\) ya sólo hay votos a distancia \(4-j\).

El algoritmo usa la menor cantidad de gasolina pues cada arista del árbol por anchura la recorre exactamente una moto, exactamente una vez. Eso es lo mínimo necesario, pues los árboes tienen la mínima cantidad de aristas necesarias para conectar una gráfica. Cualquier algoritmo que use menos gasolina tendrá una desconexión y por lo tanto no llevará todos los votos a la central electoral.

Veamos cómo queda la implementación.

```
def casillas(G,v0):
    n=G.number_of_nodes()
    encontrados={v0}
    en_proceso=[v0]
    procesados=set()
    padres={v0:None}
    distancias={v0:0}
    finales={j: True for j in G.nodes()}
    encontrados.add(v0)
    while en_proceso:
        v=en_proceso[0]
        for w in G.neighbors(v):
            if (w not in procesados) and (w not in encontrados):
                encontrados.add(w)
                en_proceso.append(w)
                padres[w]=v
                finales[w]=False
                distancias[w]=distancias[v]+1
        en_proceso.remove(v)
```

```
            procesados.add(v)

    # Vamos a ver cuántos votos debe llevar cada moto.
    por_llevar={} # Cantidad de votos que lleva la moto que sale de la casilla.
    for distancia in range(n-1,0,-1):
        for vertice in G.nodes():
            if distancias[vertice]==distancia:
                por_llevar[vertice]=por_llevar.get(vertice,0)+dict(G.nodes(data='votos'))[vertice]
                por_llevar[padres[vertice]]=por_llevar.get(padres[vertice],0)+por_llevar[vertice]

    # Vamos a preparar una digráfica de cuántos votos enviar de cada vértice a cada padre.
    scheme=nx.DiGraph()
    for par in padres.items():
        if par[1]!=None:
            scheme.add_edge(par[0],par[1],llevar=por_llevar[par[0]])
    return(scheme,distancias)
```

Estamos listos para ejecutar el algoritmo. Lo que hace es darnos una gráfica dirigida en donde dice cuántos votos debemos enviar de cada vértice. También, nos dice a qué distancia está cada vértice de la central electoral.

El código a continuación recupera esta salida para hacer una representación gráfica que podamos entender mejor. A la izquierda está la gráfica original de número de votos por casilla. A la derecha se indica en cada vértice a qué hora hay que mandar vértices. La cantidad y nodo destino se indican mediante la flecha.

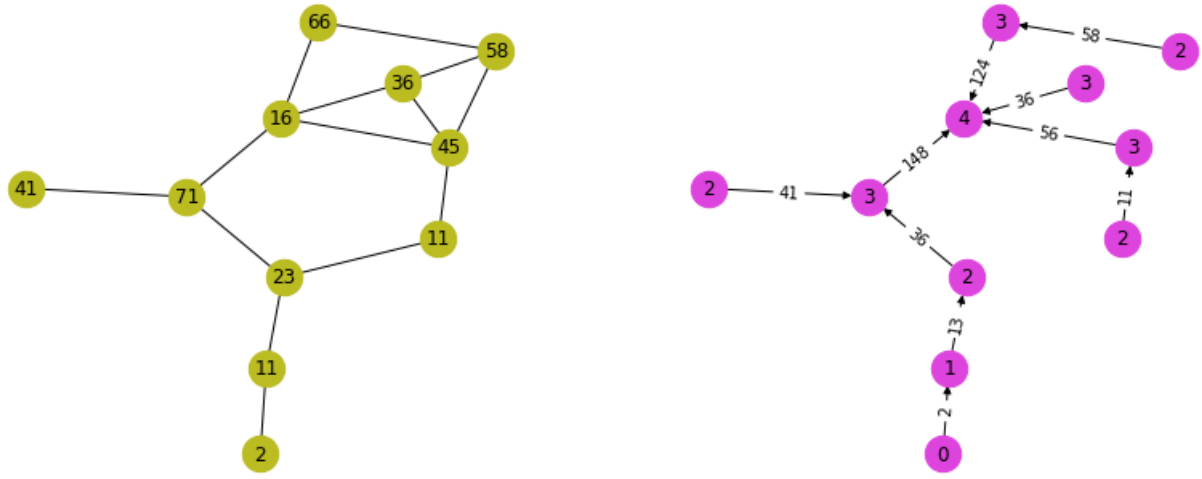
```
import matplotlib.pyplot as plt

reparticion,distancias=casillas(G,4)

d_max=max(distancias.values())
horas={j:d_max-distancias[j] for j in distancias}
LE={(j[0],j[1]):j[2] for j in reparticion.edges(data='llevar')}

fig, (ax1, ax2) = plt.subplots(1, 2)
fig.set_size_inches(14,6)

nx.draw_kamada_kawai(G,ax=ax1,labels=dict(G.nodes(data='votos')), node_color='#bbbb22',node_size=500)
nx.draw(reparticion,ax=ax2,pos=KKL,labels=horas, node_color='#dd44dd',node_size=500)
labels=nx.draw_networkx_edge_labels(reparticion,ax=ax2,pos=KKL,edge_labels=LE)
```



Así, el tiempo mínimo total es de \(4\) horas y la mínima cantidad de gasolina que se puede utilizar son \(20\) litros.

\(\square\)

Tarea moral

Los siguientes problemas te ayudarán a practicar lo visto en esta entrada. Para resolverlos, necesitarás usar herramientas matemáticas, computacionales o ambas.

1. Haz algoritmos de búsqueda por anchura que resuelvan cada uno de los siguientes problemas:
    - Para un vértice \(v\), encontrar el vértice \(u\) más lejano.
    - Para vértices \(u\) y \(v\), encontrar todos los caminos de longitud mínima de \(u\) a \(v\).
    - Para vértices \(u\), \(v\) y \(w\), determinar cuáles dos de ellos están más cerca entre sí.

    Asegúrate de que tus algoritmos no están realizando exploraciones o cómputos de más. Argumenta por qué tus algoritmos son

Asegurate de que tus algoritmos no esten realizando exploraciones o computos de mas. Argumenta por que tus algoritmos son correctos.

2. Haz un algoritmo que encuentre cuántas componentes conexas con tres elementos tiene una gráfica. Asegúrate de no estar haciendo cuentas de más haciendo que se corte la búsqueda en componentes conexas que ya son muy grandes.
3. Diseña un algoritmo que enliste todas las parejas de vértices a distancia 2 de una gráfica. Realiza un análisis de correctitud y un análisis asintótico de complejidad.
4. Considera el problema de la jornada electoral pero en una gráfica G conexa de n vértices. Demuestra con formalidad que la mínima cantidad de horas que se necesitan es igual a la distancia del vértice más lejano al centro electoral y que la mínima cantidad de gasolina para hacer este tiempo es iguala a $2(n-1)$.
5. Diseña un algoritmo de búsqueda por anchura que determine si una gráfica es bipartita o no.

Anterior
Búsqueda por anchura

Siguiente
Búsqueda por profundidad

Por Leonardo Ignacio Martínez Sandoval