x

Algoritmos con grafos

May 19, 2023

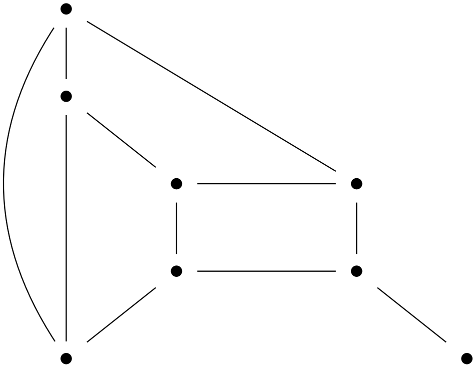
Contenido

Arboles de expansión

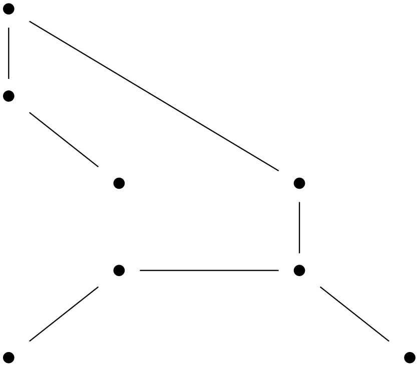
Se dice que un árbol T es un **árbol de expansión** de un grafo G al subgrafo tal que:

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

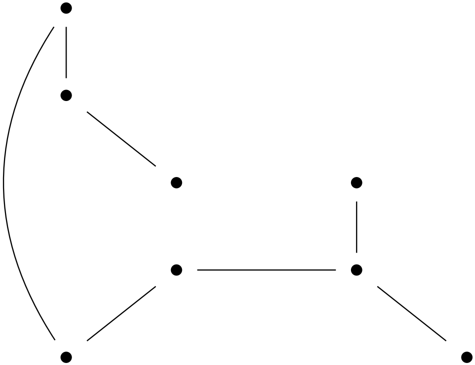
Por ejemplo, dado este grafo



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

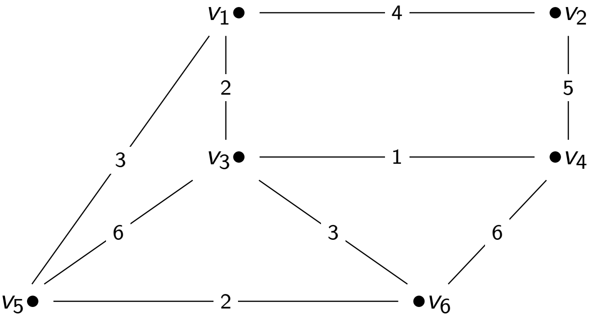
- Un árbol de expansión para un grafo existe si y solo si el grafo es conexo.

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.

Contenido

El grafo 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.

Contenido

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.

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()
```

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)
```

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])
```

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:

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

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

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
```

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)]
```

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 )
```

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')
```

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"`

```python
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.