DUC IN ALTUM

UNIVERSIDAD AUTONOMA DE CHILE

MÁS UNIVERSIDAD

ANÁLISIS DE ALGORITMOS

Mg. Marcos Contreras F.

SER PUNTUAL

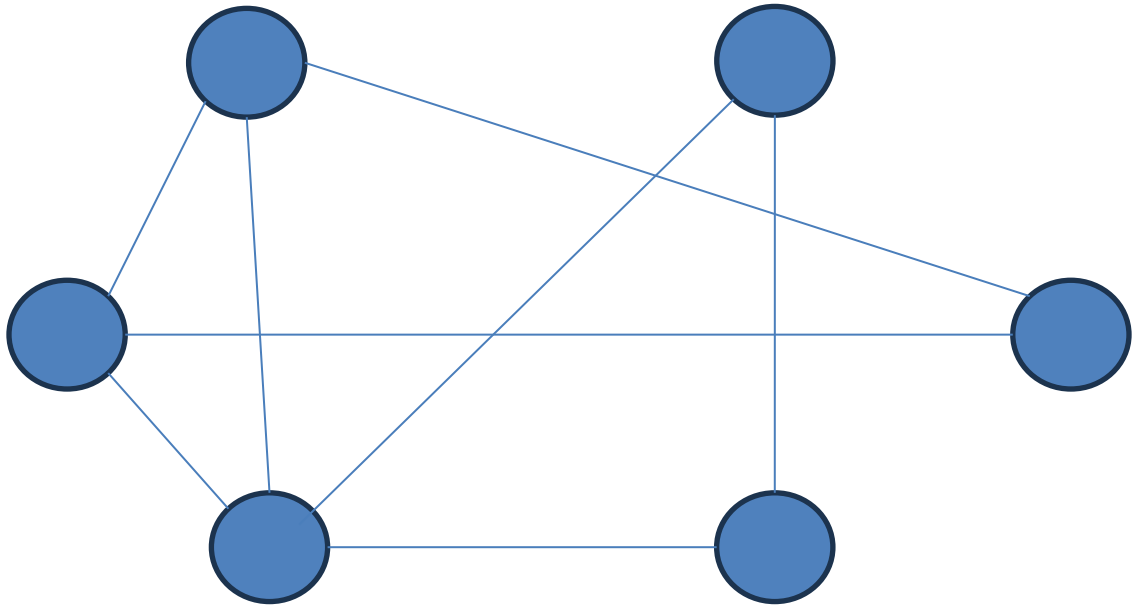
"Es valorar tu tiempo y el mío"

Teoría de Grafos

Un **grafo** es una estructura que consta de un conjunto de **vértices** (o nodos) y un conjunto de **aristas** (o enlaces) que conectan pares de vértices.

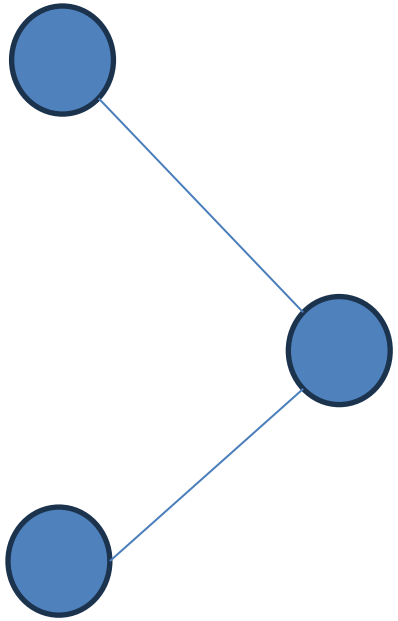
Formalmente, un grafo G se define como G = (V, E), donde:

- V es el conjunto de vértices.
- E es el conjunto de aristas.

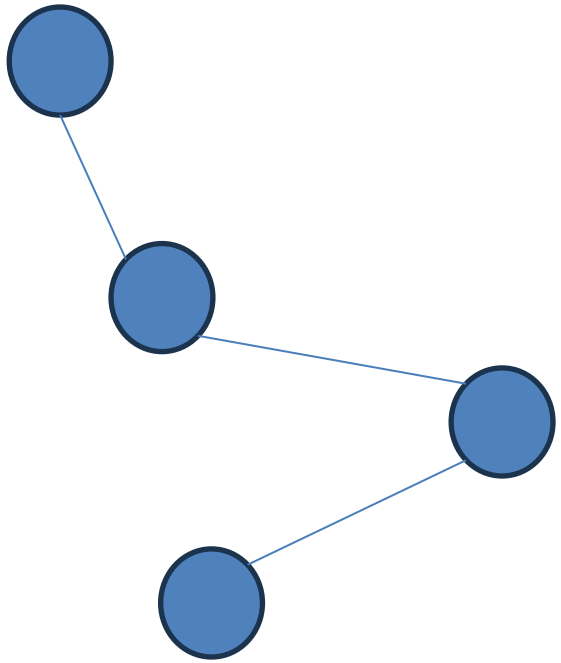


Un grafo está formado por un conjunto de vértices y otro de aristas que los conectan.

El grado de un vértice o nodo es el número de aristas o enlaces que están conectadas a él.

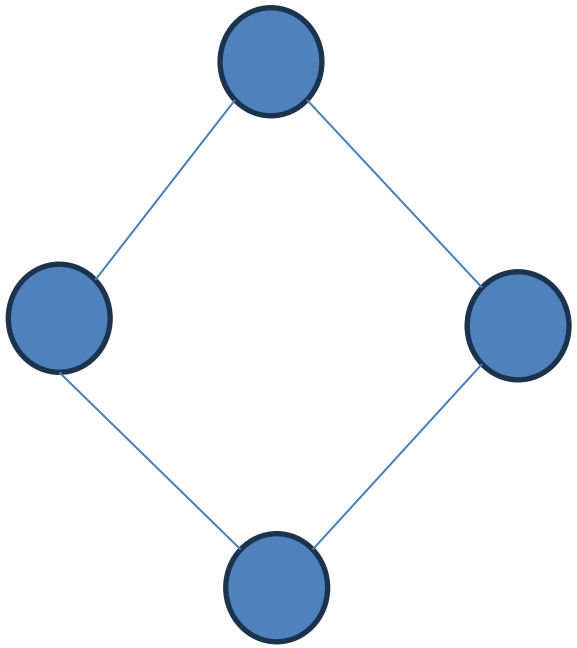


P3

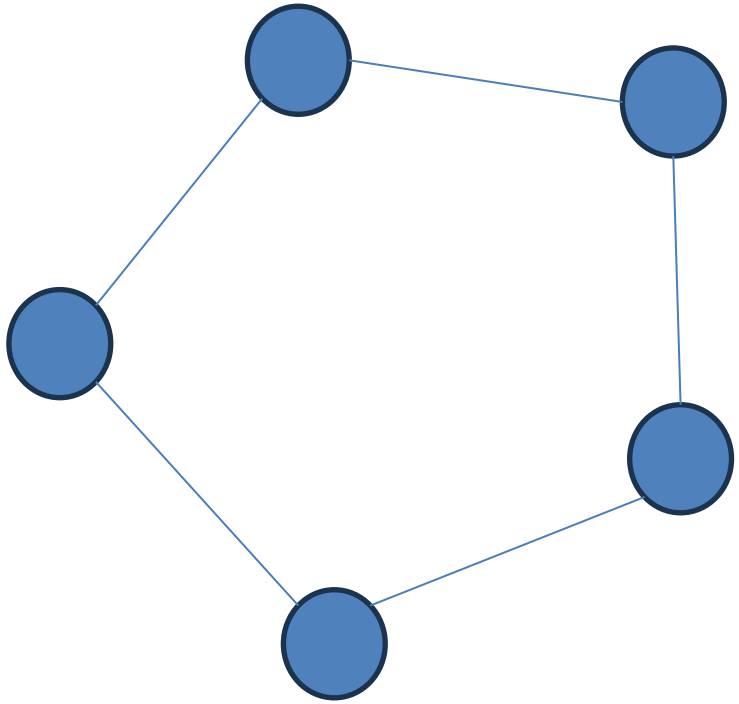


P4

Los Grafos camino: los vértices se conectan uno detrás de otro formando un camino sin conectar los extremos. Se usa la letra P de Path (Camino).



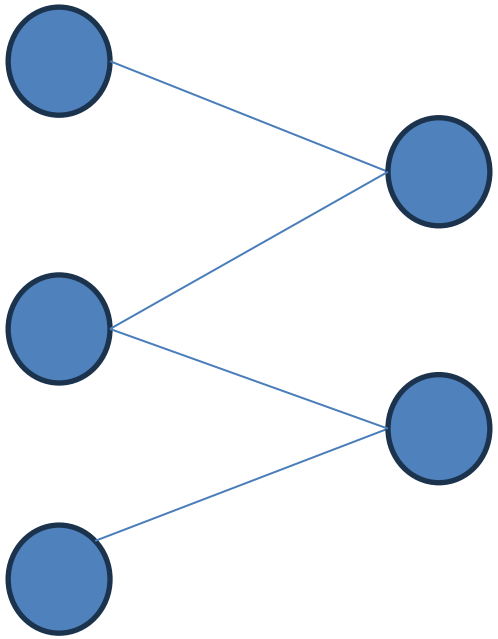
C4



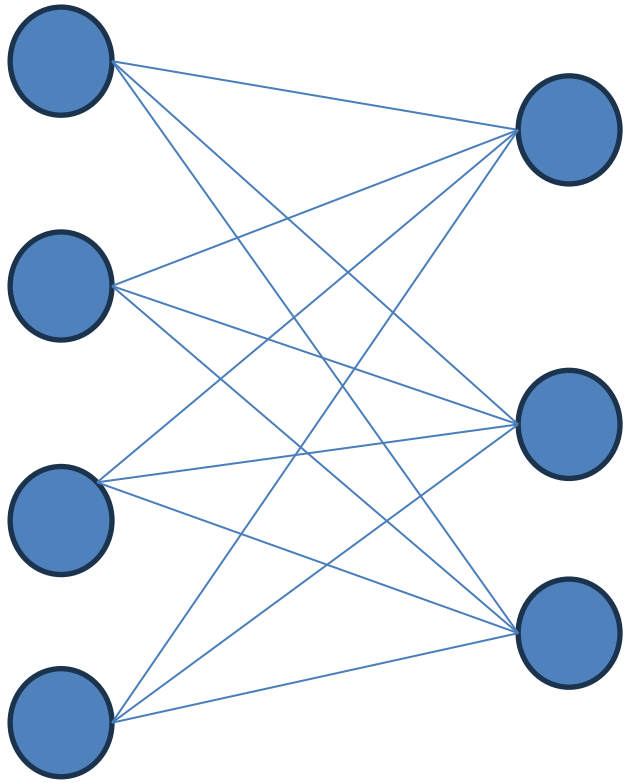
C5

Grafos ciclo: Cada vértice se conecta con otros dos formando un ciclo. Se usa la letra C de Cycle.

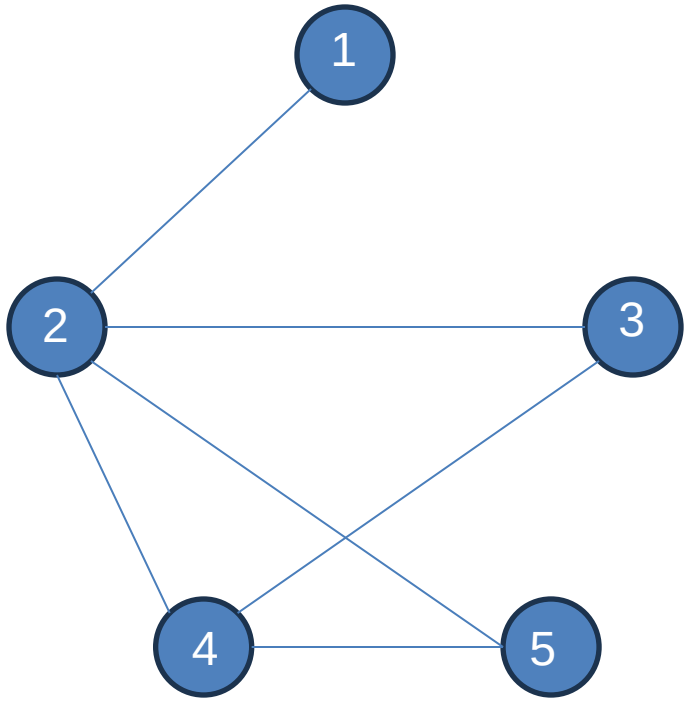
Grafos Bipartito y Bipartito Completo



K3,2



K4,3



La formal, nombrando los vértices y cada una de las aristas con llaves.

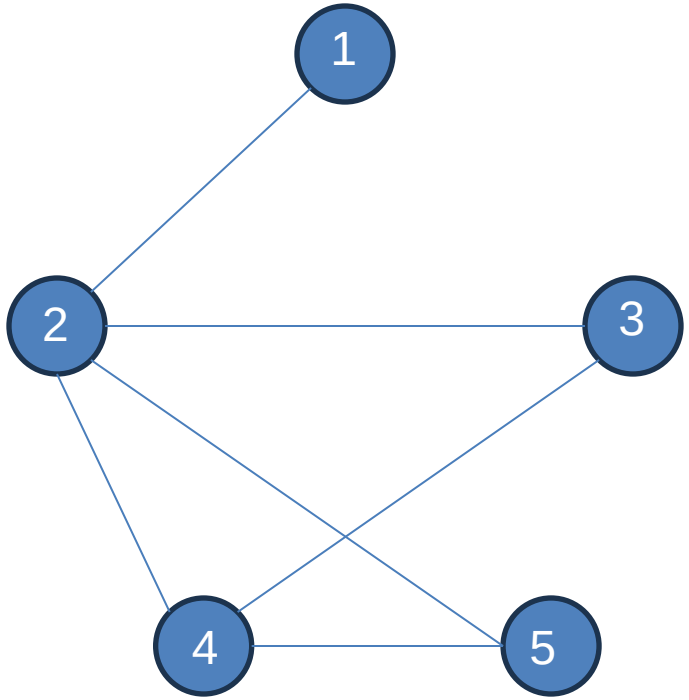
V = {1,2,3,4,5}

A = { {1,2}, {2,3} , {2,4} , {2,5} , {3,4} , {4,5} }

La definición formal de un grafo es la composición de dos conjuntos finitos: el conjunto de vértices y el conjunto de aristas.

Definición formal

- Un grafo G está formado por dos conjuntos finitos: N y A.
- N es el conjunto de vértices o nodos.
- A es el conjunto de aristas o arcos, que son las conexiones que relacionan los nodos.

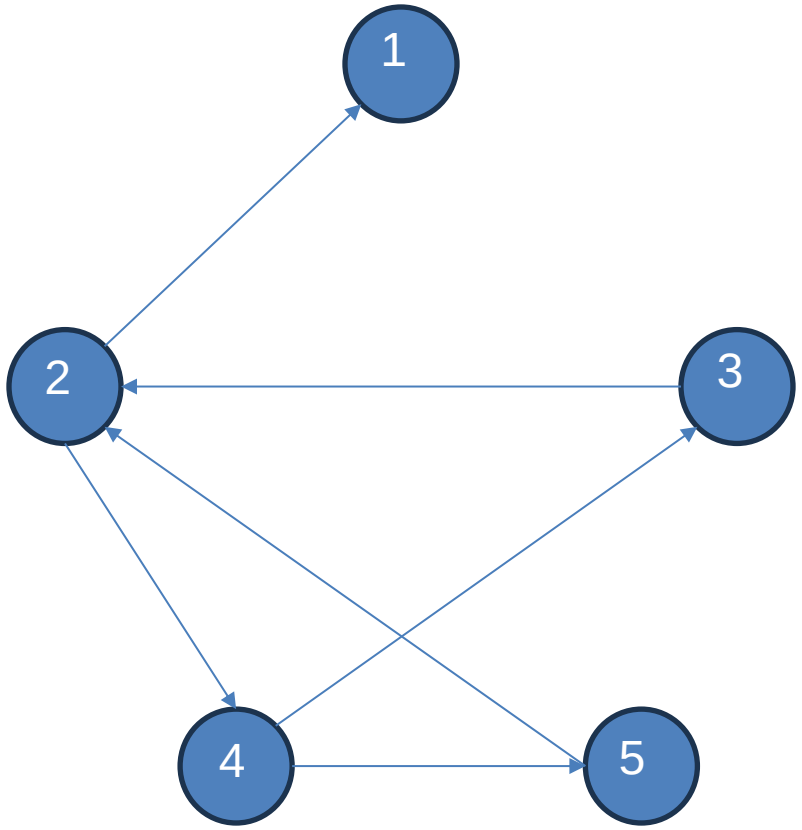


Lista de Adyacencias, una lista con las adyacencias de cada vértice.

[[2], [1,3,4,5], [2,4], [2,3,5], [2,4]]

MATRIZ DE ADYACENCIAS

	1	2	3	4	5
1	0	1	0	0	0
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0



Los grafos pueden ser dirigidos, en vez de aristas tendríamos flechas indicando la dirección, conocidas como arcos y esto afectaría en la forma de representarlo ya que ahora no es lo mismo {1,2} que {2,1}, para ello usamos un par ordenado, usando paréntesis en vez de llaves.

FORMAL

V = {1,2,3,4,5}

A = { (1,2), (2,3), (2,4), (2,5), (3,4), (4,5) }

MATRIZ DE ADYACENCIAS

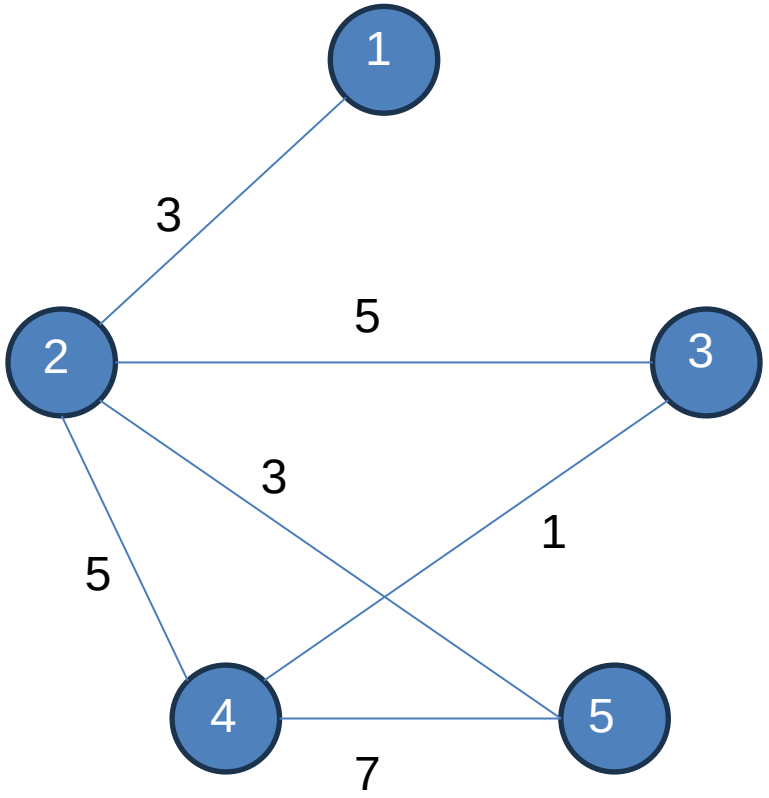
	1	2	3	4	5
1	0	0	0	0	0
2	1	0	0	1	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	1	0	0	0

Un grafo puede ser Ponderado, esto significa que podemos asociar un número a cada arista al que llamaremos peso y se suele usar la **w** de weight.

V = {1,2,3,4,5}

A = { {1,2}, {2,3} , {2,4} , {2,5} , {3,4} , {4,5} }

W = { {3} , {5}, {5}, {3}, {1}, {7} }



MATRIZ DE ADYACENCIAS

	1	2	3	4	5
1	0	3	0	0	0
2	3	0	5	5	3
3	0	5	0	1	0
4	0	5	1	0	7
5	0	3	0	7	0

Tipos de Grafos

- **Grafo dirigido**: Las aristas tienen una dirección.
- **Grafo no dirigido**: Las aristas no tienen dirección.
- **Grafo ponderado**: Las aristas tienen un peso o costo asociado.
- **Grafo no ponderado**: Las aristas no tienen peso.

¿Qué es un grafo dirigido?

Un **grafo dirigido** (también llamado **dígrafo**) es un tipo de grafo en el cual **cada arista tiene una dirección**. Es decir, las conexiones entre los vértices no son de ida y vuelta necesariamente.

G=(V,A)

donde:

- V es un conjunto de **vértices** o **nodos**.
- A es un conjunto de **aristas dirigidas** (también llamadas **arcos**), donde cada arco es un **par ordenado** (u,v), que indica una conexión **desde** el vértice u **hacia** el vértice v.

Ejemplo sencillo

Imagina que tienes tres vértices: A, B y C, y las siguientes conexiones:

- De A a B
- De B a C
- De A a C

Esto **no** implica que puedas ir de B a A o de C a A automáticamente, porque la dirección importa.

Grado de Entrada (In-degree)

Número de aristas que **entran** a un vértice.

Ejemplo:

- In-degree de C = 2 (viene desde A y B).

Grado de Salida (Out-degree)

Número de aristas que **salen** de un vértice.

Ejemplo:

- Out-degree de A = 2 (hacia B y C).

Aplicaciones de Grafos Dirigidos

- Redes sociales: Un usuario puede seguir a otro, pero no necesariamente ser seguido de vuelta (como en Twitter).
- Mapas de carreteras: Algunas calles son de sentido único.
- Tareas dependientes: Planificación de proyectos donde ciertas tareas deben completarse antes que otras (Diagramas de precedencia).

Ejercicio Rápido

Dado este conjunto de aristas:

- (A, B)
- (B, A)
- (B, C)
- (C, A)

1. ¿Cuál es el grado de entrada de A?
2. ¿Cuál es el grado de salida de B?

Respuesta:

1. Entrada a A: Desde B y desde C → **2**
2. Salida de B: Hacia A y hacia C → **2**

Matriz de Adyacencias

	A	B	C
A	0	1	0
B	1	0	1
C	1	0	0

Problema 1: Ruta óptima en un grafo dirigido

Supongamos el siguiente grafo:

- A → B (peso 2)
- A → C (peso 5)
- B → C (peso 1)
- B → D (peso 2)
- C → D (peso 3)

Pregunta: ¿Cuál es el camino más corto de A a D?

Matriz de Adyacencias

	A	B	C	D
A	0	2	5	0
B	0	0	1	2
C	0	0	0	3
D	0	0	0	0

Solución

1. **Inicialización:**
    1. **Distancia a A = 0**
    2. **Distancia a todos los demás = ∞**
2. **Desde A:**
    1. **A → B = 2 (mejorar distancia a B)**
    2. **A → C = 5 (mejorar distancia a C)**
3. **Desde B (distancia actual 2):**
    1. **B → C: 2 + 1 = 3 (mejorar distancia a C, antes era 5)**
    2. **B → D: 2 + 2 = 4 (actualizar distancia a D)**
4. **Desde C (distancia actual 3):**
    1. **C → D: 3 + 3 = 6 (pero ya tenemos un camino a D de peso 4, así que no actualizamos)**
5. **Resultado final:**
6. **Camino más corto de A a D es A → B → D, con distancia total 4.**

Problema 2: Detectar ciclo en un grafo dirigido

Dado el siguiente conjunto de aristas:

- A → B
- B → C
- C → A

¿Tiene ciclo?

Solución:
Si partimos de A:

- A lleva a B
- B lleva a C
- C regresa a A

Sí, tiene un ciclo (A → B → C → A)

Algoritmo de Dijkstra (para encontrar el camino más corto)

¿Qué hace?

Busca el **camino más corto** desde un vértice origen a todos los demás vértices en un grafo con **pesos no negativos**.

Pasos del Algoritmo:

1. **Inicializar**:
    1. Asignar distancia 0 al nodo de origen.
    2. Asignar distancia infinita (∞) a todos los otros nodos.
    3. Marcar todos los nodos como no visitados.
2. **Seleccionar** el nodo no visitado con **menor distancia** actual.
3. **Actualizar** la distancia de los nodos vecinos:
    1. Si el camino a través del nodo actual es más corto, actualizar.
4. **Marcar** el nodo actual como visitado.
5. **Repetir** hasta que todos los nodos estén visitados o alcanzados.

Ejemplo:

Grafo:

- A → B (peso 2)
- A → C (peso 4)
- B → C (peso 1)
- B → D (peso 7)
- C → D (peso 3)

Encuentra el camino más corto de A a D.

Nodo	Distancia Inicial
A	0
B	∞
C	∞
D	∞

Proceso:

1.Desde A:

1. A → B: distancia 2 (mejor que ∞)
2. A → C: distancia 4 (mejor que ∞)

2.Nodo con menor distancia: **B (2)**.

1. B → C: 2 + 1 = 3 (mejor que 4, actualizamos)
2. B → D: 2 + 7 = 9

3.Nodo con menor distancia: **C (3)**.

1. C → D: 3 + 3 = 6 (mejor que 9, actualizamos)

4.Nodo D (6): Llegamos.

Camino más corto: A → B → C → D

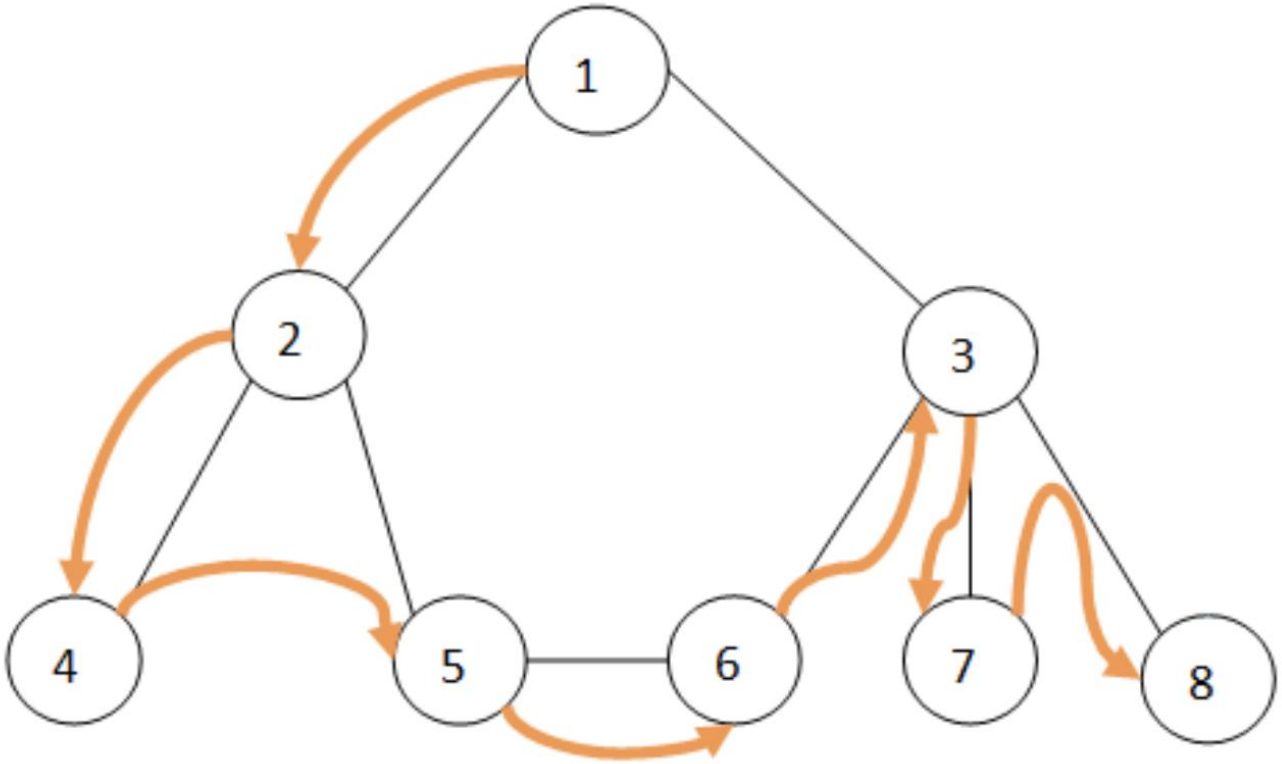
Costo total: 6

Búsqueda en Profundidad

Una búsqueda en profundidad (DFS) es un algoritmo de búsqueda para lo cual recorre los nodos de un grafo. Su funcionamiento consiste en ir expandiendo cada uno de los nodos que va localizando, de forma recurrente (desde el nodo padre hacia el nodo hijo).

Cuando ya no quedan más nodos que visitar en dicho camino, regresa al nodo predecesor, de modo que repite el mismo proceso con cada uno de los vecinos del nodo. Cabe resaltar que, si se encuentra el nodo antes de recorrer todos los nodos, concluye la búsqueda.

La búsqueda en profundidad se usa cuando queremos probar si una solución entre varias posibles cumple con ciertos requisitos; como sucede en el problema del camino que debe recorrer un caballo en un tablero de ajedrez para pasar por las 64 casillas del tablero.


1
2
3
4
5
6
7
8

Implementación de **DFS (Depth-First Search)** usando **recursión**, con estructuras para marcar nodos visitados, procesados y para registrar el **padre** de cada nodo en el recorrido.

- nodo_visitado: marca si un nodo ya fue recorrido.
- nodo_procesado: marca si se terminó completamente de explorar sus vecinos.
- nodo_padre: guarda quién fue el "padre" que lo descubrió en el recorrido DFS.
- procesar_nodo y procesar_lado: funciones que puedes adaptar según lo que necesites registrar, imprimir o calcular.

```python
def DFS(grafo, s, nodo_visitado, nodo_procesado, nodo_padre):
    nodo_visitado[s] = True
    procesar_nodo(s)

    for vecino in grafo[s]:
        if not nodo_visitado.get(vecino, False):
            nodo_padre[vecino] = s
            DFS(grafo, vecino, nodo_visitado, nodo_procesado, nodo_padre)
        elif not nodo_procesado.get(vecino, False):
            procesar_lado(s, vecino)

    nodo_procesado[s] = True

def procesar_nodo(nodo):
    print(f"Procesando nodo {nodo}")

def procesar_lado(origen, destino):
    print(f"Procesando lado de {origen} a {destino}")

# Grafo con vértices string (no dirigido)
grafo = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A'],
    'D': ['B']
}

# Inicialización con diccionarios
nodo_visitado = {}
nodo_procesado = {}
nodo_padre = {}

# Llamada inicial desde nodo 'A'
inicio = 'A'
DFS(grafo, inicio, nodo_visitado, nodo_procesado, nodo_padre)

print("\nÁrbol DFS (padres):")
for hijo, padre in nodo_padre.items():
    print(f"{hijo} ← {padre}")
```

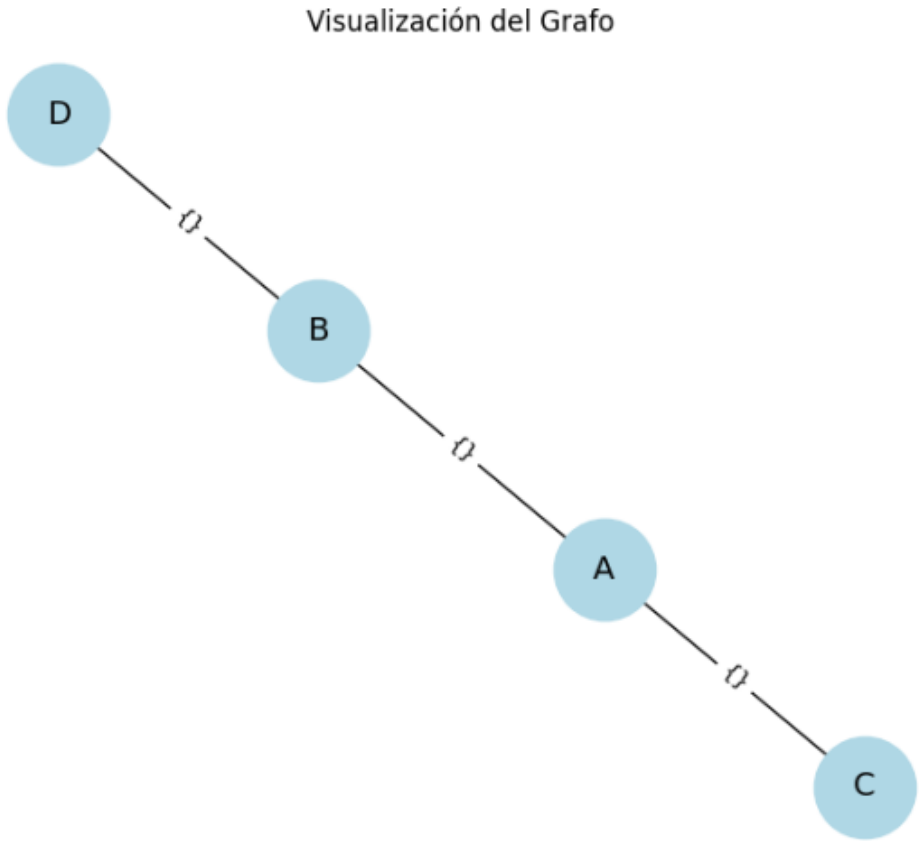
```
pip install networkx Matplotlib

import networkx as nx
import matplotlib.pyplot as plt

# Crear el grafo desde diccionario
G = nx.Graph()  # Usa nx.DiGraph() si es dirigido

for nodo, vecinos in grafo.items():
    for vecino in vecinos:
        G.add_edge(nodo, vecino)

# Dibujar el grafo
plt.figure(figsize=(6, 5))
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='lightblue',
node_size=2000, font_size=14)
nx.draw_networkx_edge_labels(G, pos)
plt.title("Visualización del Grafo")
plt.show()
```



Aplicaciones DFS

El algoritmo de búsqueda en profundidad tienes varias aplicaciones, entre las cuales tenemos las siguientes:

- Encontrar nodos conectados en un grafo
- Ordenamiento topológico en un grafo acíclico dirigido
- Encontrar puentes en un grafo de nodos
- Resolver puzles con una sola solución, como los laberintos
- Encontrar nodos fuertemente conectados

En un grafo acíclico dirigido, es decir un conjunto de nodos donde cada nodo tiene una sola dirección que no es consigo mismo, se puede realizar un ordenamiento topológico mediante el algoritmo DFS.

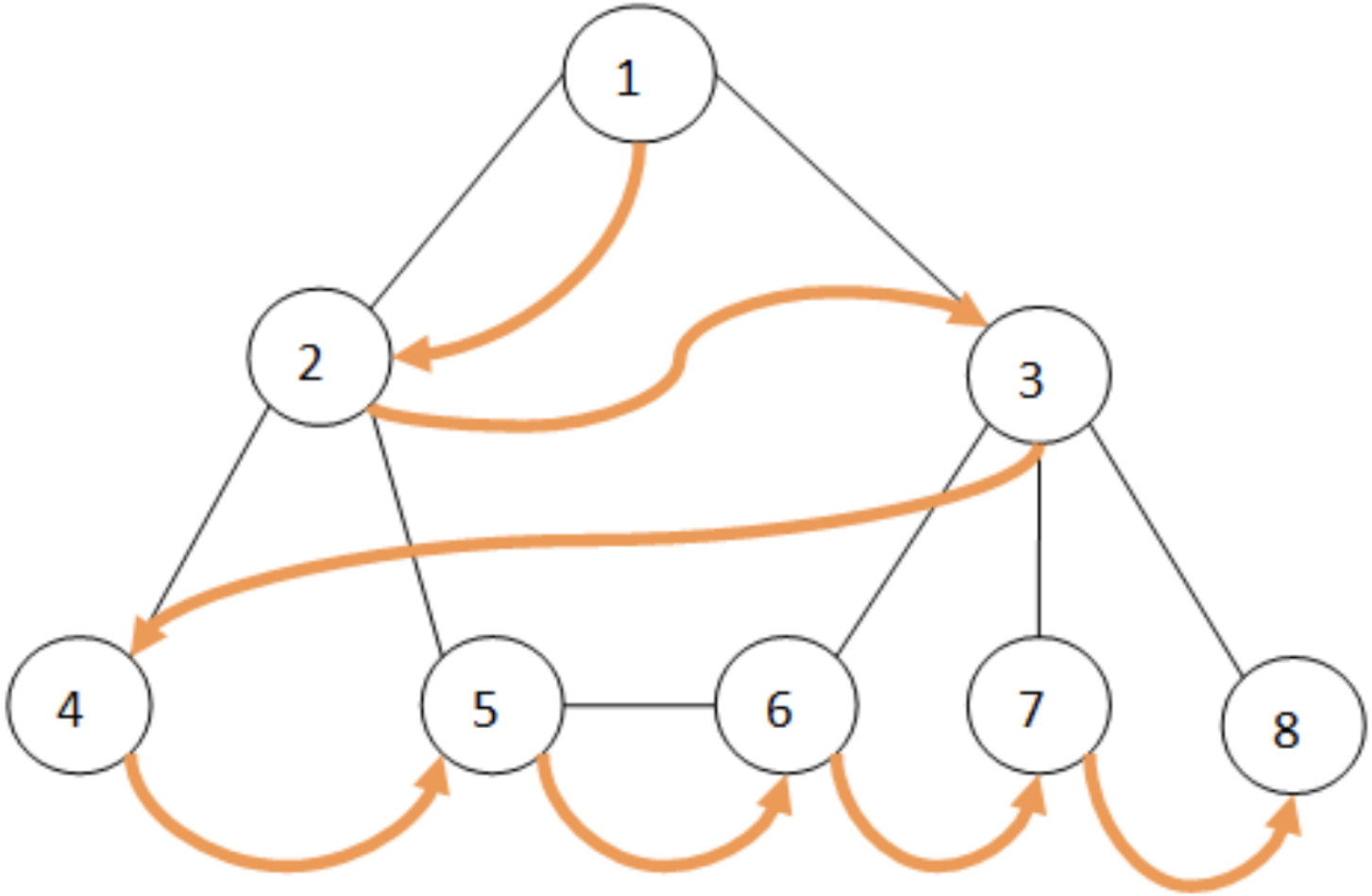
Por ejemplo, se puede aplicar para organizar actividades que tienen por lo menos alguna dependencia entre sí, a fin de organizar eficientemente la ejecución de una lista de actividades.

Búsqueda en Anchura

Una búsqueda en anchura (BFS) es un algoritmo de búsqueda para lo cual recorre los nodos de un grafo, comenzando en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo), para luego explorar todos los vecinos de este nodo.

A continuación, para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo. Cabe resaltar que, si se encuentra el nodo antes de recorrer todos los nodos, concluye la búsqueda.

La búsqueda por anchura se usa para aquellos algoritmos en donde resulta crítico elegir el mejor camino posible en cada momento del recorrido.


1
2
3
4
5
6
7
8

```
from collections import deque
import networkx as nx
import matplotlib.pyplot as plt

# ---------- Funciones de procesamiento ----------
def procesar_nodo(v):
    print(f"Procesando nodo {v}")

def procesar_lado(v1, v2):
    print(f"Procesando lado de {v1} a {v2}")

# ---------- Algoritmo BFS ----------
def BFS(grafo, s, nodo_visitado, nodo_procesado, nodo_padre):
    cola = deque()
    nodo_visitado[s] = True
    cola.append(s)

    while cola:
        v = cola.popleft()
        nodo_procesado[v] = True
        procesar_nodo(v)

        for vecino in grafo[v]:
            if not nodo_visitado.get(vecino, False):
                cola.append(vecino)
                nodo_visitado[vecino] = True
                nodo_padre[vecino] = v
            if not nodo_procesado.get(vecino, False):
                procesar_lado(v, vecino)
```

```
# ---------- Grafo de prueba ----------
grafo = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'E'],
    'D': ['B'],
    'E': ['C']
}

# ---------- Inicialización de estructuras ----------
nodo_visitado = {}
nodo_procesado = {}
nodo_padre = {}

# ---------- Ejecutar BFS ----------
BFS(grafo, 'A', nodo_visitado, nodo_procesado, nodo_padre)

print("\nPadres descubiertos (Árbol BFS):")
for hijo, padre in nodo_padre.items():
    print(f"{hijo} ← {padre}")

# ---------- Graficar ----------
# Grafo original (no dirigido)
G = nx.Graph()
for nodo, vecinos in grafo.items():
    for vecino in vecinos:
        G.add_edge(nodo, vecino)
```

```
# Árbol BFS como grafo dirigido
arbol_BFS = nx.DiGraph()
for hijo, padre in nodo_padre.items():
    arbol_BFS.add_edge(padre, hijo)

# Usamos la misma posición para ambos dibujos
pos = nx.spring_layout(G)

plt.figure(figsize=(12, 5))

# Grafo original
plt.subplot(1, 2, 1)
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=2000,
font_size=12)
plt.title("Grafo Original")

# Árbol BFS
plt.subplot(1, 2, 2)
nx.draw(arbol_BFS, pos, with_labels=True, node_color='lightgreen',
node_size=2000, font_size=12, arrows=True)
plt.title("Árbol BFS desde 'A'")

plt.tight_layout()
plt.show()
```

Usa la IA para el desarrollo de problemas, pero no **Abuses**.


consultas
dudas
dudas
dudas
consultas
consultas
dudas
dudas
consultas
dudas
dudas
dudas
dudas
dudas
consultas
dudas
consultas
dudas


THANK
YOU
GRACIAS
ARIGATO
SHUKURIA
DANKSCHEEN
BIYAN
SHUKRIA
TINGKI
YAQHANYELAY
TASHAKKUR ATU
SUKSAMA
EKHMET
MEHRBANI
GRAZIE
MAAKE
PALDIES
KOMAPSUMNIDA
GOZAIMASHITA
EFCHARISTO
JUSPAXAR
BOLZÏN
MERCI

¡¡¡Gracias por la asistencia… Éxito!!!