

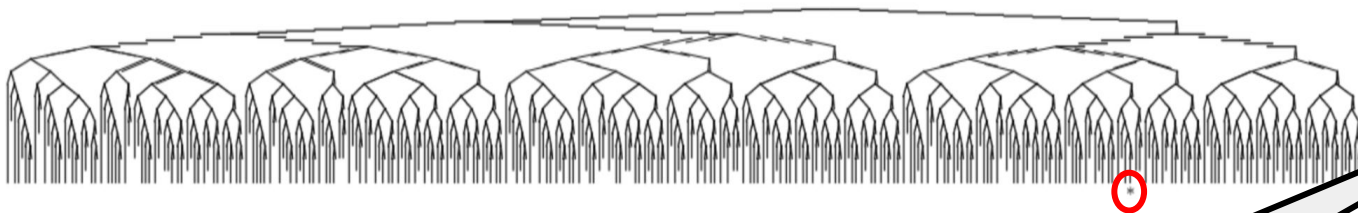
# Algoritmos de búsqueda



# Algoritmos de búsqueda

Los algoritmos de búsqueda sirven para **encontrar una o más soluciones** ante problemas que tienen **resultados o posibilidades finitas**.

Está pensado para aquellos problemas donde hay una cantidad **ENORME de posibles estados** siguientes. (Esto representa un problema para casi cualquier sistema de inferencia)

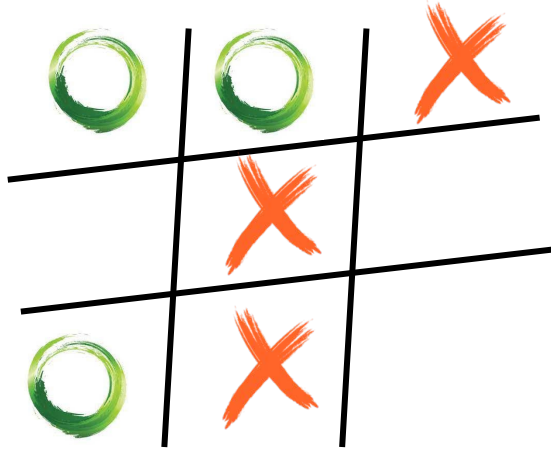


*"Hay 14,000,605 finales,  
y solo en este ganamos"*

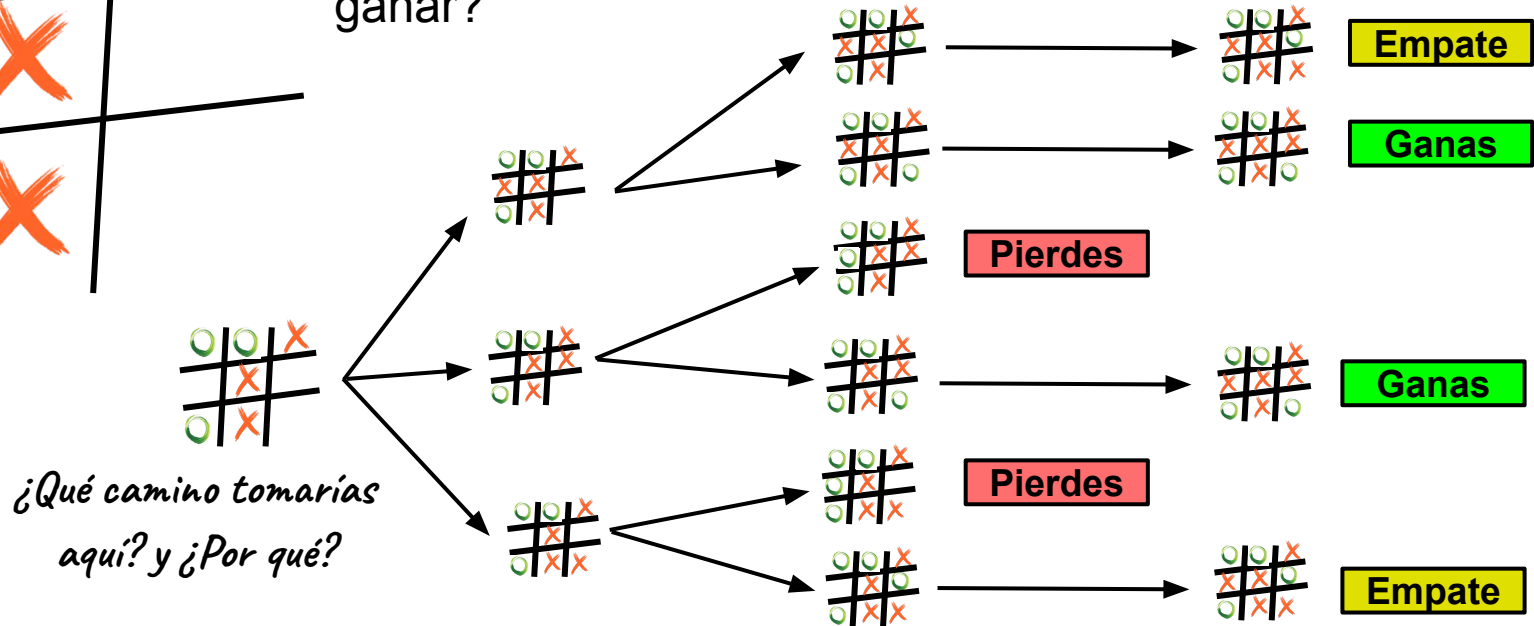


# Un ejemplo simple

Imagina que tienes el siguiente juego de gato:



Tú eres el jugador que está poniendo las x, a continuación, es tu turno. Sabiendo esto, ¿En cuántos escenarios puedes ganar?



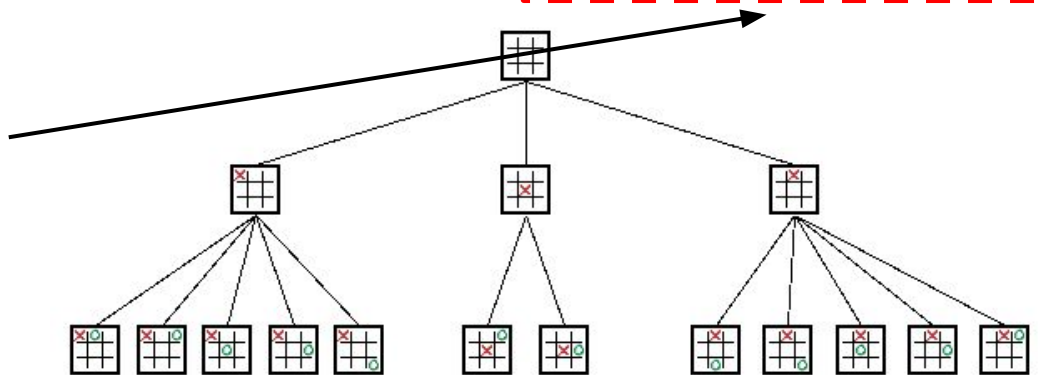
# Un ejemplo simple

En realidad, los humanos también analizamos esas posibilidades, pero **es muy complicado analizar todas** las existentes a futuro.

En el caso del gato, sin importar quién empiece, existen 9 tiradas posibles inicialmente, después 8, después 7, etc. Lo cual se puede representar como:

$$\text{Movimientos}_{\text{Totales}} = 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 9! = 362,880 \text{ Jugadas}$$

*Tóricamente, podrían haber 362,880 jugadas en el juego de gato, sin embargo, este número es menor porque en muchas de ellas el juego se gana y no se continúa con las jugadas posteriores*



# Algoritmos de búsqueda

En el gato existen 255,168 posibles jugadas para llegar a algún ganador. (Sigue siendo una cantidad muy grande de posibilidades). ¿Imaginas cómo sería para el caso del Ajedrez?

Comencemos por entender que, para analizar estos problemas, **siempre contamos con estados (s) y acciones (a)**, cada acción debería de **llevarnos a un nuevo estado s**, representado por ejemplo:

$$s' = a_1(s)$$

$$s'' = a_2(s)$$

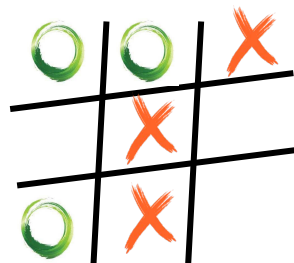
*La aplicación recursiva de todas las acciones posibles a todos los estados, comenzando con el estado inicial, describe todo el árbol de búsqueda*

# Definiciones

**Estado:** Descripción de dónde se encuentra el agente de búsqueda en un punto específico del tiempo.

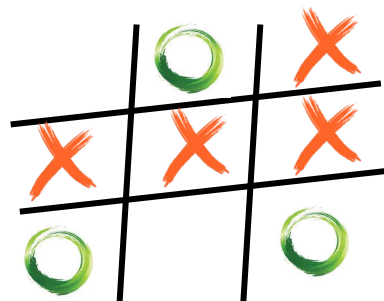
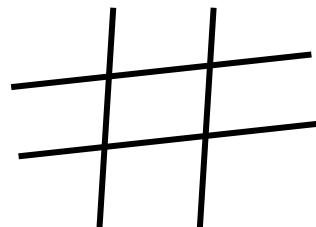
**Estado inicial:** Es el primer estado o la primer posibilidad con las que se puede abrir un problema de búsqueda.

**Estado Objetivo:** Es el estado deseado al que queremos llegar, puede haber uno o muchos, y cuando se alcanza, se termina la búsqueda.



*Este es un estado de ejemplo*

*En el gato, este es el estado inicial*

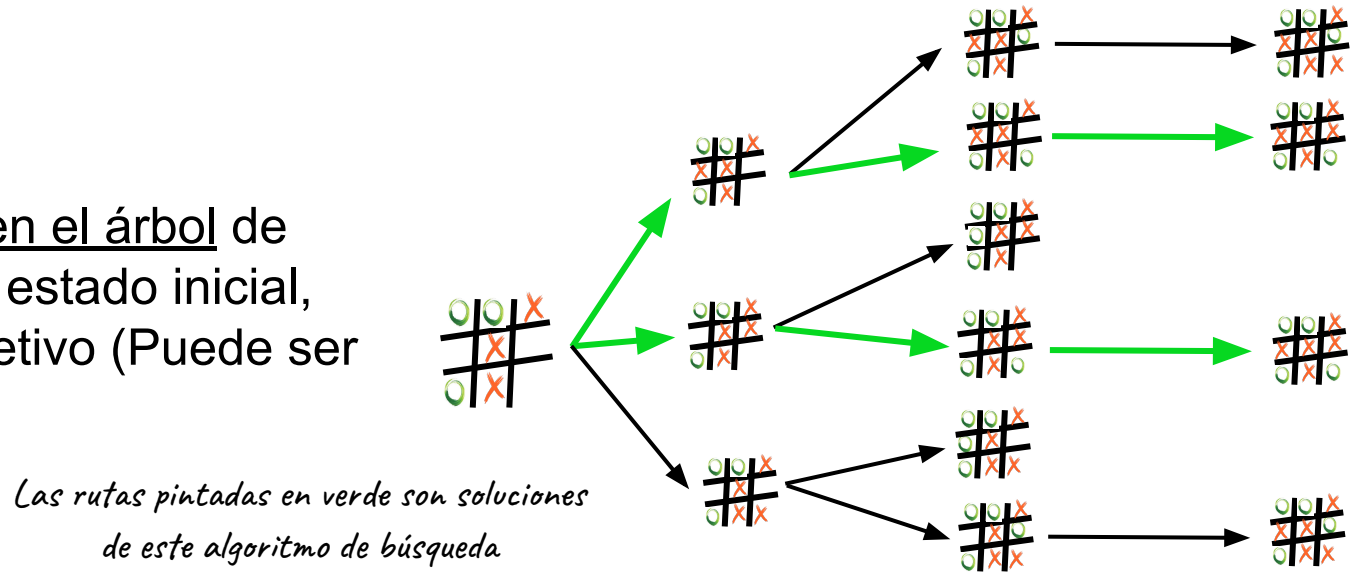
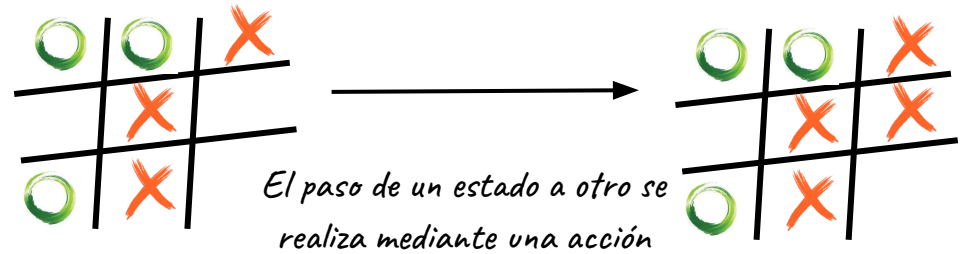


*Este es un estado objetivo porque aquí, el jugador X ya ganó la partida*

# Definiciones

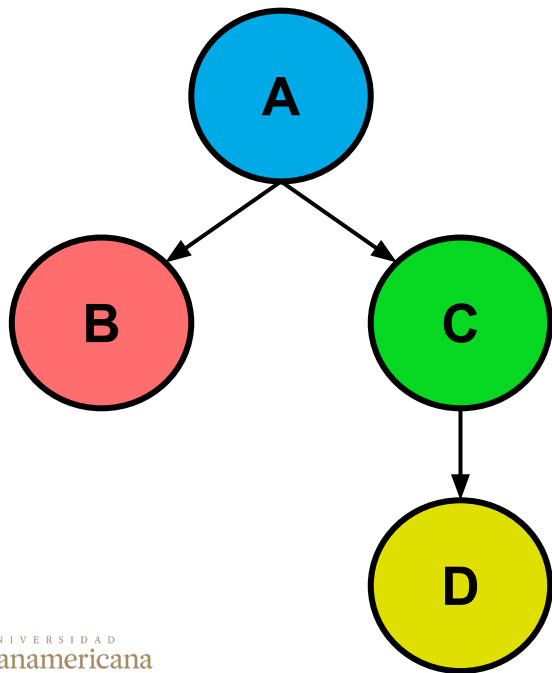
**Acción:** Todo lo que modifique un estado para pasar a uno nuevo.

**Solución:** La ruta en el árbol de búsqueda desde el estado inicial, hasta el estado objetivo (Puede ser más de una).



# Uso de Grafos

Los problemas de búsqueda se pueden representar usualmente por medio de **Grafos**. Los grafos **representan los estados**, y las **transiciones** que éstos pueden hacer a estados siguientes, por ejemplo:



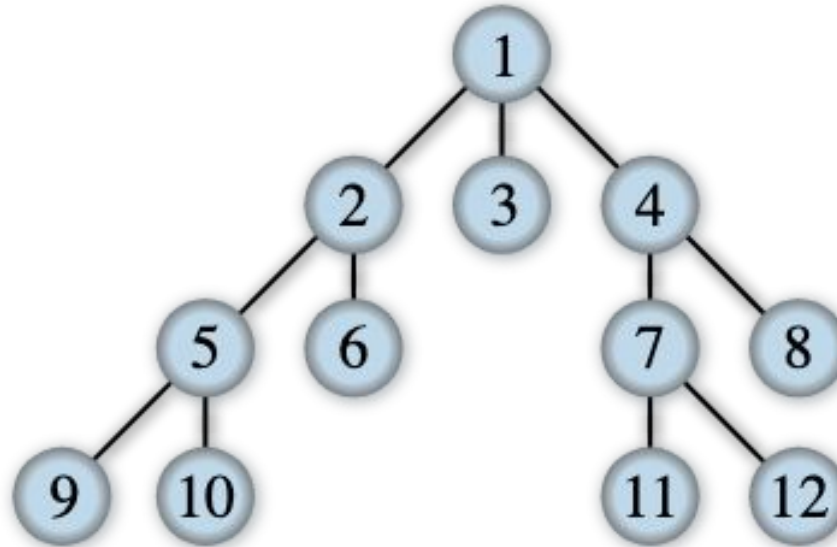
*En el grafo, cada estado puede representar un punto exacto en una partida de Ajedrez, una ubicación exacta en un mapa, o cualquier conjunto de características que representan una secuencia.*

En este ejemplo, si suponemos que el estado A es nuestro estado inicial, y D es nuestro estado objetivo, la única secuencia lógica para llegar hasta allá es:

**Solución:**  $A \rightarrow C \rightarrow D$

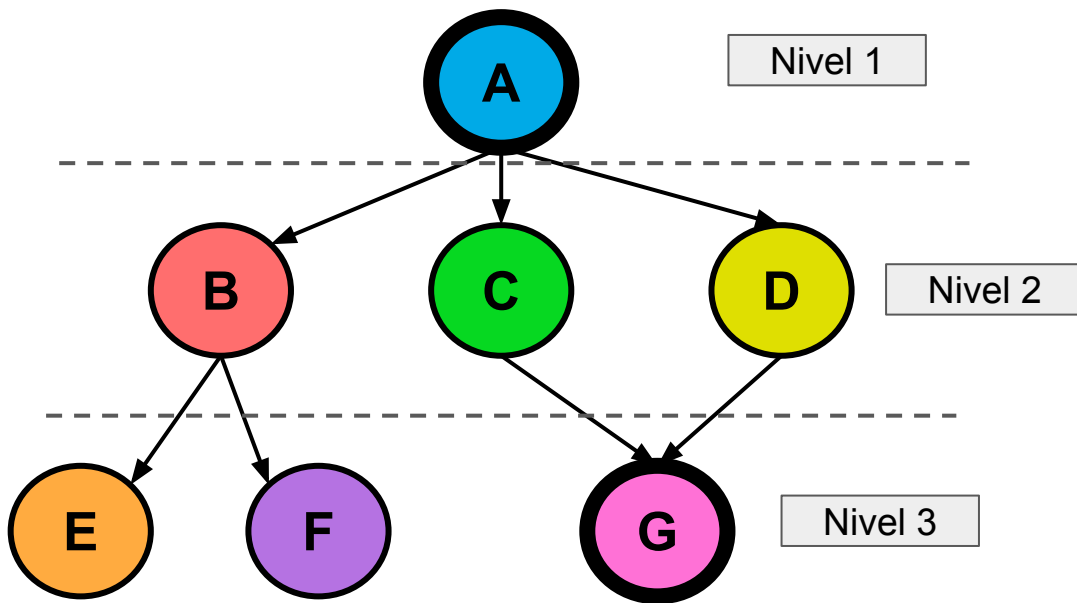


# Búsqueda en Amplitud



# Algoritmo de búsqueda en Amplitud

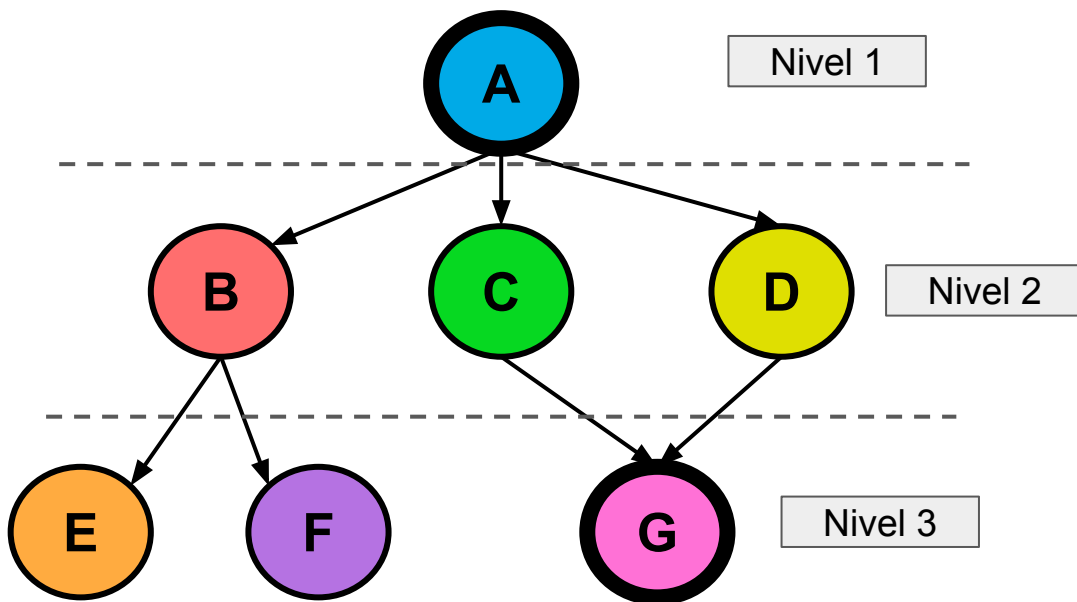
Existen *diferentes algoritmos de búsqueda* para atacar al mismo tipo de problemas, uno de ellos, es la búsqueda en amplitud, el cual consiste en buscar el estado objetivo *por niveles*:



En este ejemplo, la búsqueda en amplitud, intentará recorrer de izquierda a derecha, cada estado desde el nivel 1, hasta el 3, buscando alguna ruta que lleve del estado inicial A, al estado final G.

# Algoritmo de búsqueda en Amplitud

La secuencia, sería la siguiente:



## Nodo A:

Caminos: [A, B], [A, C], [A, D]

(Ninguno es el estado final)

## Nodo B:

Caminos: [A, B, E], [A, B, F]

(Ninguno es el estado final)

## Nodo C:

Caminos: [A, C, G]

(Esta es una solución)

## Nodo D:

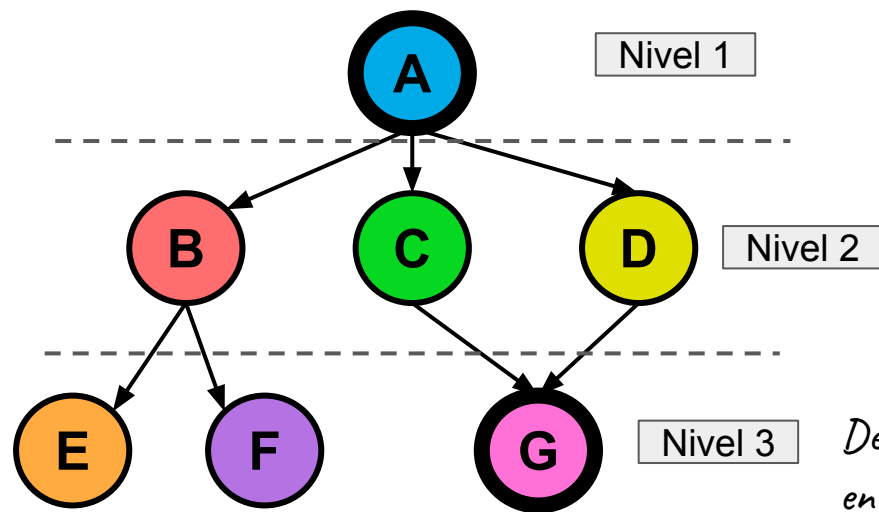
Caminos: [A, D, G]

(Esta es una solución)

*Después de la búsqueda, encontramos 2 soluciones después de evaluar 4 nodos.*

# Programación de Búsqueda en Amplitud

La búsqueda en amplitud, también conocida como “BFS” (Breadth-First Search), tiene el siguiente pseudocódigo:

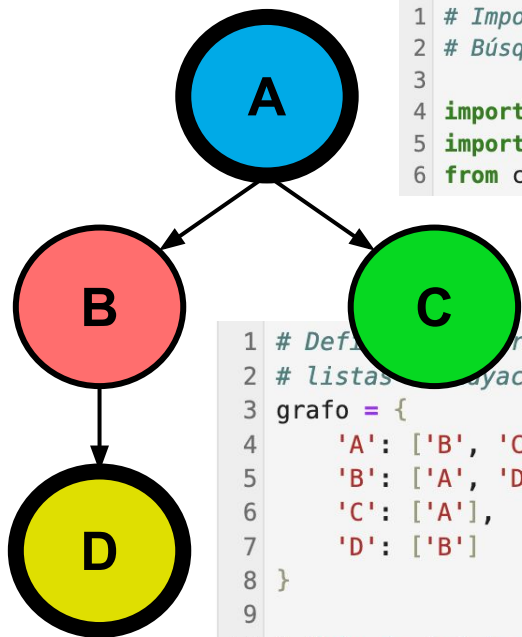


```
BREADTHFIRSTSEARCH(NodeList, Goal)
NewNodes = ∅
For all Node ∈ NodeList
    If GoalReached(Node, Goal)
        Return("Solution found", Node)
    NewNodes = Append(NewNodes, Successors(Node))
If NewNodes ≠ ∅
    Return(BREADTH-FIRST-SEARCH(NewNodes, Goal))
Else
    Return("No solution")
```

*De manera general, se compara si en el estado actual ya se está en el nodo objetivo del grafo, de lo contrario, se analiza un nuevo nodo (siguiente nivel o a la derecha), y se va guardando el camino hasta ese nodo, hasta llegar a la solución final.*

# Ejemplo de búsqueda en amplitud

Resolvamos el siguiente ejemplo, utilizando python:



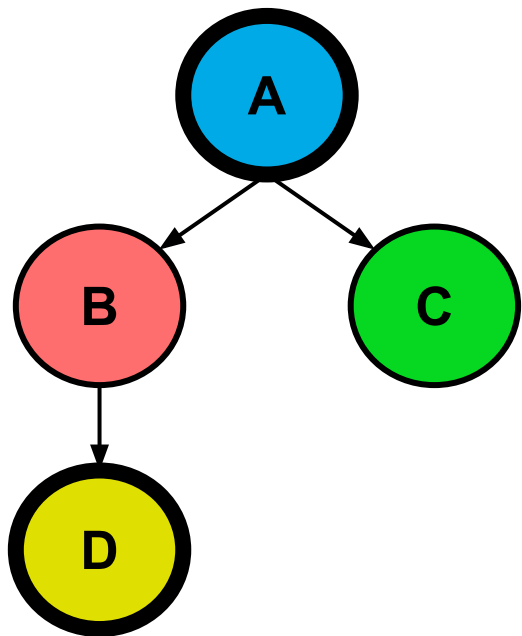
```
1 # Importamos las librerías para trat
2 # Búsqueda en amplitud
3
4 import networkx as nx
5 import matplotlib.pyplot as plt
6 from collections import deque
```

```
1 # Definimos el grafo como un diccionario de
2 # listas de adyacencia
3 grafo = {
4     'A': ['B', 'C'],
5     'B': ['A', 'D'],
6     'C': ['A'],
7     'D': ['B']
8 }
9
10 # Nodo de inicio y nodo objetivo
11 nodo_inicio = 'A'
12 nodo_objetivo = 'D'
```

```
1 # Definimos nuestro algoritmo de búsqueda:
2
3 def busqueda_amplitud(grafo, inicio, objetivo):
4     # El método set crea un conjunto de datos SIMILAR a una lista pero que
5     # NO puede tener elementos duplicados
6     visitados = set()
7     # (Doubly Ended Queue) Es un tipo de lista que permite agregar y eliminar
8     # elementos ya sea a la izquierda o a la derecha de la misma
9     cola = deque([(inicio, [inicio])])
10
11     i = 1
12     while cola:
13         # Aquí removemos el primer elemento del set (El de la izquierda)
14         # y este será nuestro nodo actual, el resto, el camino (Path restante)
15         nodo_actual, camino = cola.popleft()
16         print("\nIteración:", i)
17         print("Nodo actual:", nodo_actual)
18
19         if nodo_actual == objetivo:
20             print("Camino encontrado:", "->".join(camino))
21             return camino
22
23         if nodo_actual not in visitados:
24             visitados.add(nodo_actual)
25             print("Nodos visitados", visitados)
26             for vecino in grafo[nodo_actual]:
27                 if vecino not in visitados:
28                     nueva_ruta = camino + [vecino]
29                     print("Nueva ruta:", nueva_ruta)
30                     cola.append((vecino, nueva_ruta))
31
32             i += 1
33     print("No se encontró un camino al nodo objetivo.")
34     return None
```

# Ejemplo de búsqueda en amplitud

La solución debería de ser:



```
1 # Realizamos la búsqueda en amplitud
2 camino = busqueda_amplitud(grafo, nodo_inicio, nodo_objetivo)
```

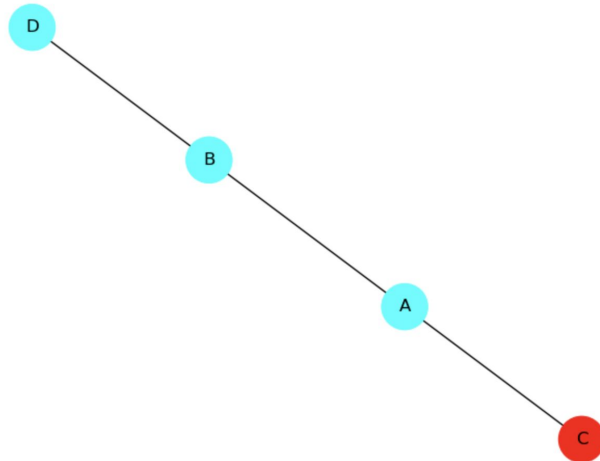
Iteración: 1  
Nodo actual: A  
Nodos visitados {'A'}  
Nueva ruta: ['A', 'B']  
Nueva ruta: ['A', 'C']

Iteración: 2  
Nodo actual: B  
Nodos visitados {'B', 'A'}  
Nueva ruta: ['A', 'B', 'D']

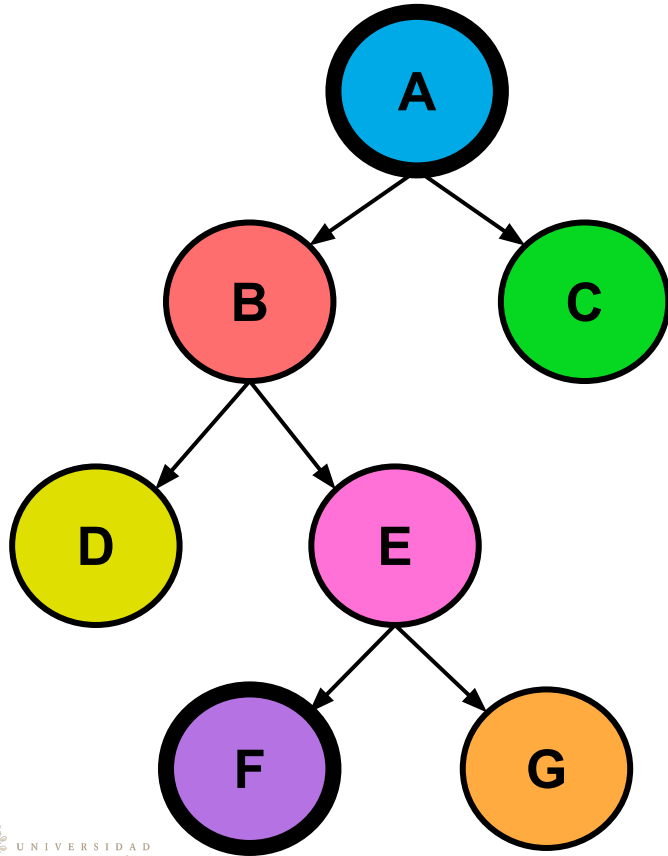
Iteración: 3  
Nodo actual: C  
Nodos visitados {'B', 'C', 'A'}

Iteración: 4  
Nodo actual: D  
Camino encontrado: A->B->D

```
1 # Creamos el gráfico
2 G = nx.Graph(grafo)
3
4 # Coloreamos los nodos según si están en el camino o no
5 colores = ['blue' if nodo in camino else 'red' for nodo in G.nodes()]
6
7 # Dibujamos el grafo
8 pos = nx.spring_layout(G)
9 nx.draw(G, pos, with_labels=True, node_color=colores, node_size=1000)
10 plt.title("Búsqueda en Amplitud")
11 plt.show()
```



# Ejemplo 2



Iteración: 1  
Nodo actual: A  
Nodos visitados {'A'}  
Nueva ruta: ['A', 'B']  
Nueva ruta: ['A', 'C']

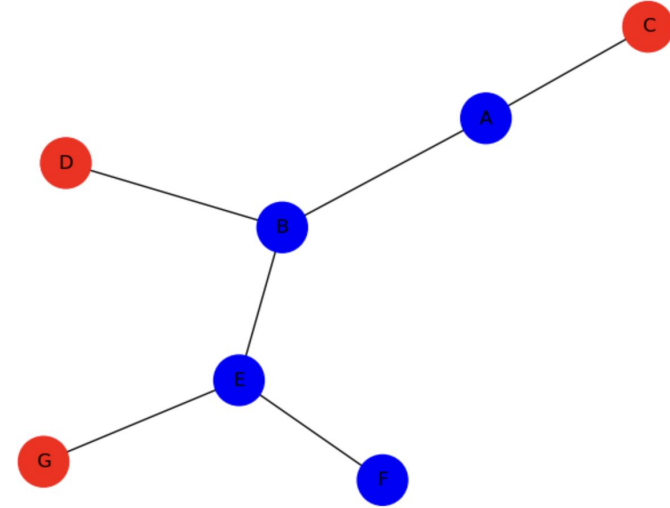
Iteración: 2  
Nodo actual: B  
Nodos visitados {'A', 'B'}  
Nueva ruta: ['A', 'B', 'D']  
Nueva ruta: ['A', 'B', 'E']

Iteración: 3  
Nodo actual: C  
Nodos visitados {'A', 'B', 'C'}

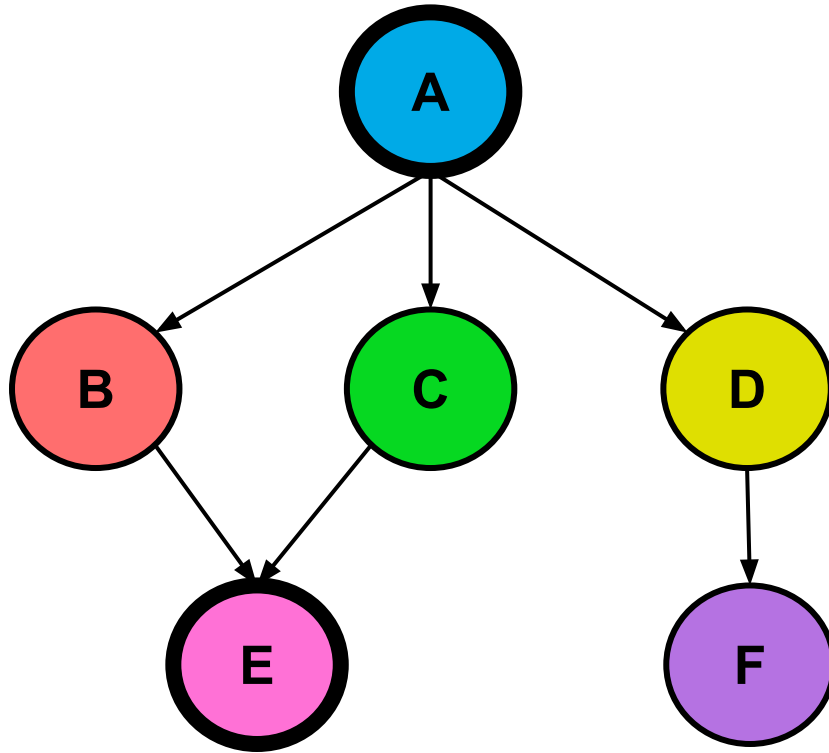
Iteración: 4  
Nodo actual: D  
Nodos visitados {'A', 'B', 'D', 'C'}

Iteración: 5  
Nodo actual: E  
Nodos visitados {'A', 'B', 'D', 'C', 'E'}  
Nueva ruta: ['A', 'B', 'E', 'F']  
Nueva ruta: ['A', 'B', 'E', 'G']

Iteración: 6  
Nodo actual: F  
Camino encontrado: A->B->E->F



# Ejemplo 3



Iteración: 1  
Nodo actual: A  
Nodos visitados {'A'}  
Nueva ruta: ['A', 'B']  
Nueva ruta: ['A', 'C']  
Nueva ruta: ['A', 'D']

Iteración: 2  
Nodo actual: B  
Nodos visitados {'A', 'B'}  
Nueva ruta: ['A', 'B', 'E']

Iteración: 3  
Nodo actual: C  
Nodos visitados {'A', 'B', 'C'}  
Nueva ruta: ['A', 'C', 'E']

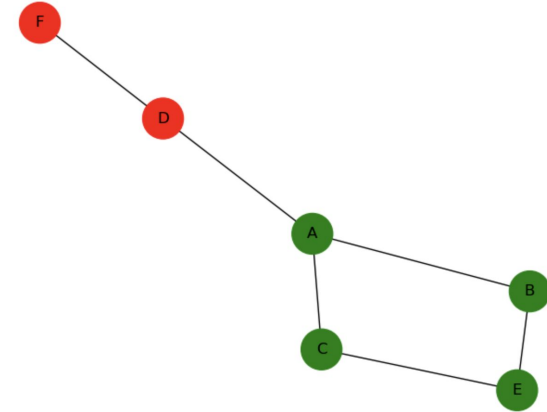
Iteración: 4  
Nodo actual: D  
Nodos visitados {'A', 'D', 'B', 'C'}  
Nueva ruta: ['A', 'D', 'F']

Iteración: 5  
Nodo actual: E  
Nodos visitados {'A', 'E', 'B', 'C', 'D'}

Iteración: 6  
Nodo actual: E

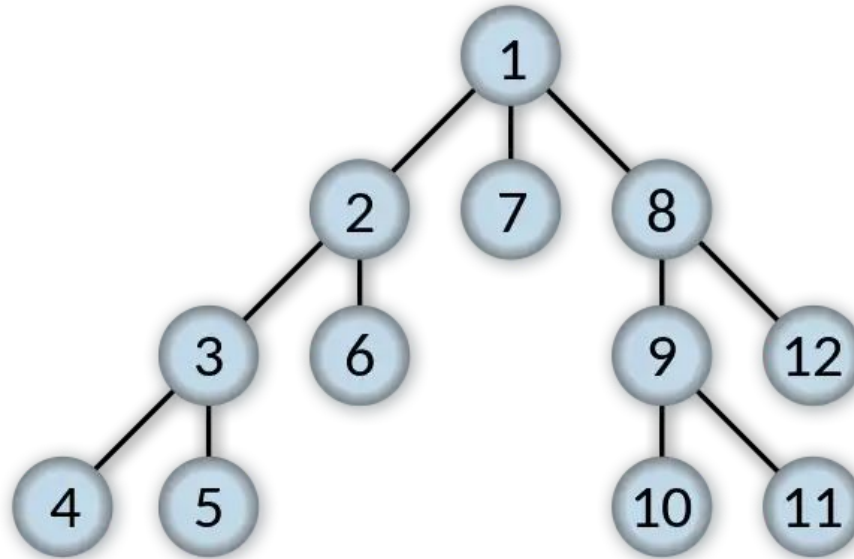
Iteración: 7  
Nodo actual: F  
Nodos visitados {'A', 'E', 'B', 'F', 'C', 'D'}

Camino 1: A → B → E  
Camino 2: A → C → E



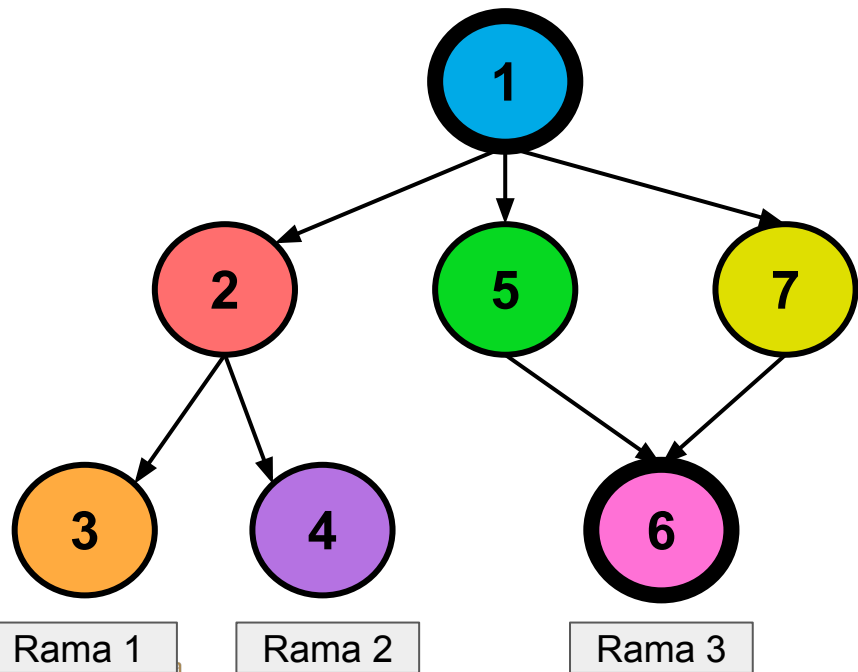


# Búsqueda en Profundidad



# Algoritmo de búsqueda en profundidad

A diferencia de la búsqueda en amplitud, la búsqueda en profundidad, **NO se realizar por niveles sino por ramas**, hasta llegar al último nodo, o lo que se conoce como **hojas del árbol**:

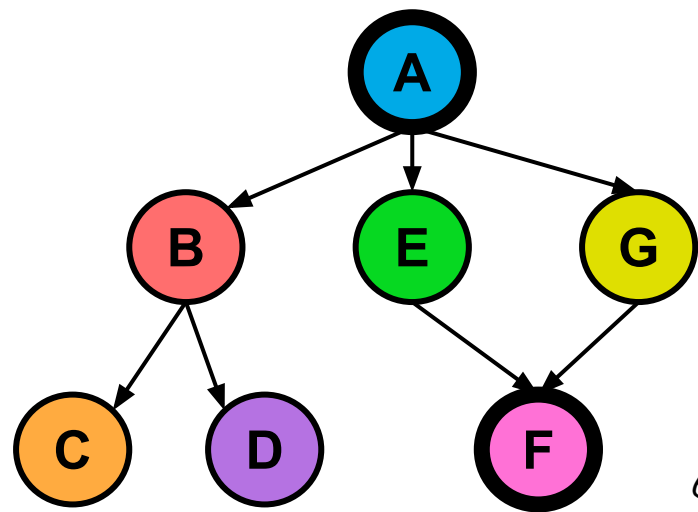


Si comparamos la secuencia con la búsqueda en amplitud, en este algoritmo llegaremos más rápido al nodo naranja (3), pero tardaremos mucho más en llegar al nodo amarillo (7).

Un **algoritmo será mejor que otro dependiendo del problema que se esté atacando**.

# Programación de Búsqueda en Profundidad

La búsqueda en profundidad, también conocida como “DFS” (Depth-First Search), tiene el siguiente pseudocódigo:



Rama 1

Rama 2

Rama 3

```
DEPTHFIRSTSEARCH(Node, Goal)
```

```
If GoalReached(Node, Goal) Return("Solution found")
```

```
NewNodes = Successors(Node)
```

```
While NewNodes  $\neq \emptyset$ 
```

```
    Result = DEPTH-FIRST-SEARCH(First(NewNodes), Goal)
```

```
    If Result = "Solution found" Return("Solution found")
```

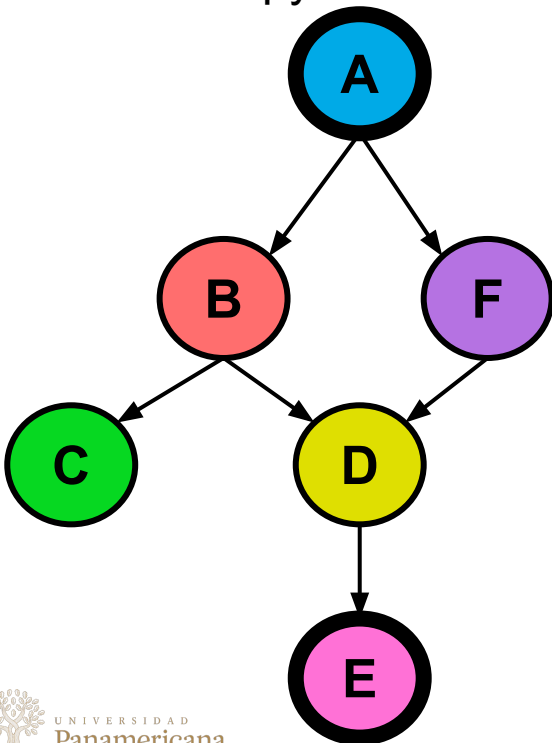
```
    NewNodes = Rest(NewNodes)
```

```
Return("No solution")
```

*En general, con este algoritmo buscaremos todos los nodos vecinos por los que no hayamos pasado aún, y seguiremos hasta llegar a la hoja, o nodo final de la rama actual en el grafo, antes de continuar con la siguiente*

# Ejemplo de búsqueda en profundidad

Resolvamos el siguiente ejemplo, utilizando python:



```
# Importamos las librerías para trabajar con DFS(Depth-First Search)
# Búsqueda en profundidad
```

```
import networkx as nx
import matplotlib.pyplot as plt
```

```
# Definimos el grafo como un diccionario
grafo = {
    'A': ['B', 'F'],
    'B': ['A', 'C', 'D'],
    'C': ['B'],
    'D': ['B', 'E', 'F'],
    'E': ['D'],
    'F': ['A', 'D']
}
```

```
def dfs(grafo, inicio, objetivo, visitados=None, camino=None, i=0):
    i += 1
    print("\nIteración:", i)

    # Si entramos a este método por primera vez
    if visitados is None:
        visitados = set()
    if camino is None:
        camino = []

    camino.append(inicio)
    visitados.add(inicio)

    print("Nodo actual:", inicio)
    print("Nodos visitados", visitados)
    print("Camino Actual:", camino)

    if inicio == objetivo:
        print("Camino encontrado:", ' -> '.join(camino))
        return camino

    for vecino in grafo[inicio]:
        if vecino not in visitados:
            nuevo_camino = dfs(grafo, vecino, objetivo, visitados, camino.copy(), i)
            if nuevo_camino:
                return nuevo_camino

    return None
```

```
# Nodo de inicio y nodo objetivo
```

```
nodo_inicio = 'A'
nodo_objetivo = 'F'
```

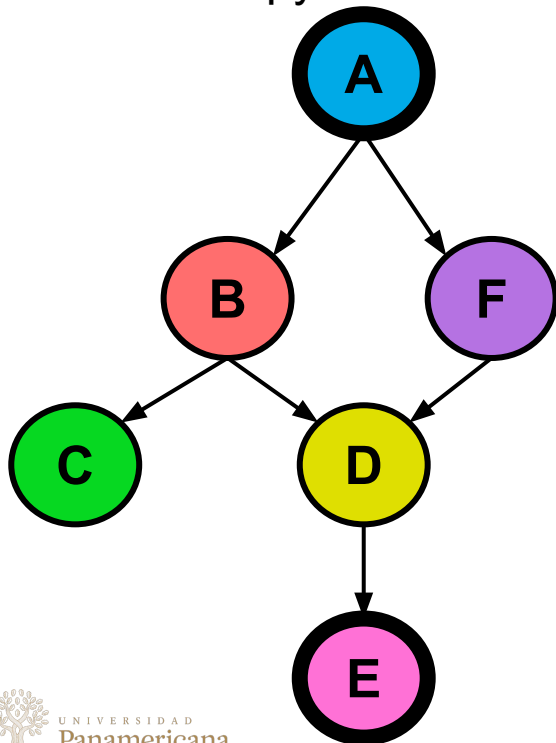
```
print("Recorrido DFS:")
```

```
camino_encontrado = dfs(grafo, nodo_inicio, nodo_objetivo)
```

```
if camino_encontrado:
    print("Camino encontrado:", ' -> '.join(camino_encontrado))
else:
    print("No se encontró un camino al nodo objetivo.")
```

# Ejemplo de búsqueda en profundidad

Resolvamos el siguiente ejemplo, utilizando python:



Recorrido DFS:

Iteración: 1

Nodo actual: A

Nodos visitados {'A'}

Camino Actual: ['A']

Iteración: 2

Nodo actual: B

Nodos visitados {'A', 'B'}

Camino Actual: ['A', 'B']

Iteración: 3

Nodo actual: C

Nodos visitados {'A', 'B', 'C'}

Camino Actual: ['A', 'B', 'C']

Iteración: 3

Nodo actual: D

Nodos visitados {'A', 'B', 'D', 'C'}

Camino Actual: ['A', 'B', 'D']

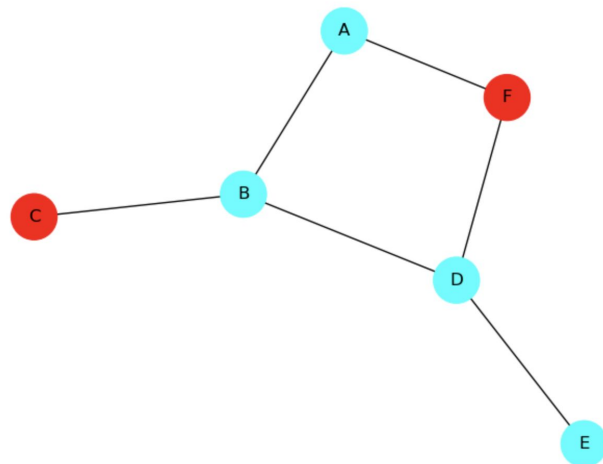
Iteración: 4

Nodo actual: E

Nodos visitados {'B', 'A', 'E', 'C', 'D'}

Camino Actual: ['A', 'B', 'D', 'E']

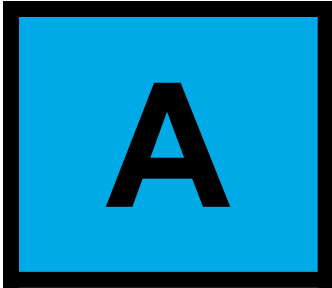
Camino encontrado: A -> B -> D -> E



# Uso de algoritmos para mapeo de rutas

Los algoritmos de búsqueda en amplitud y en profundidad, son muy útiles para el **mapeo de rutas**, lo cual puede servir para **encontrar caminos en tableros**, rutas de trenes, vuelos de conexión, y varias aplicaciones más.

Comencemos por algo básico, imagina que tienes una casilla única:



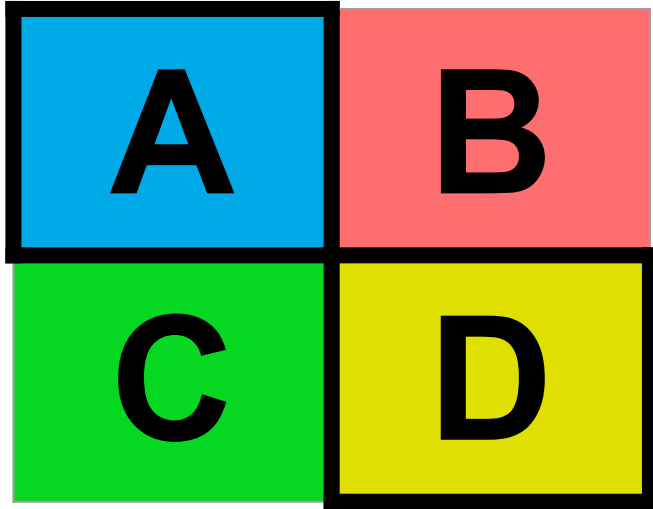
*En este ejemplo, partimos de nuestra coordenada ÚNICA "A", y queremos llegar a "A", entonces solo hay una única solución, que es:*

*A (Ningún cambio es necesario)*

*(En este caso, no es necesario movernos a ningún lado porque YA estamos en la coordenada deseada)*

# Uso de algoritmos para mapeo de rutas

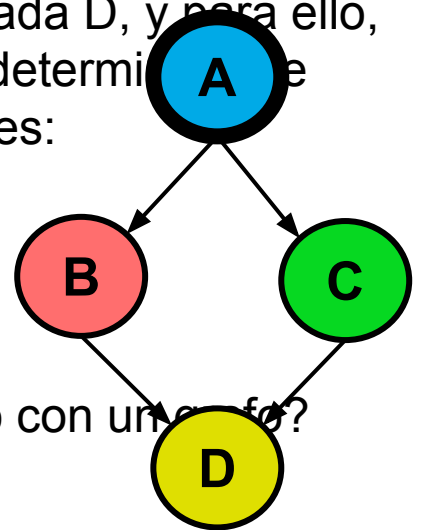
Ahora, algo un poco más complicado, imaginemos que tenemos un tablero o un mapa que contiene coordenadas de 2x2:



En este nuevo ejemplo, queremos llegar de la coordenada A a la coordenada D, y para ello, como humanos, podemos determinar que existen dos caminos posibles:

*A -> B -> D*

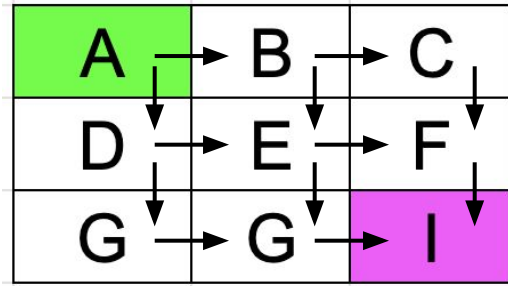
*A -> C -> D*



¿Cómo representarías esto con un código?  
¿Y cómo lo programarías?

# Ejercicio:

Realizar la programación correspondiente para encontrar la ruta más cercana desde la coordenada “A”, hasta la coordenada “I”, suponiendo que solo te puedes mover a la izquierda, derecha, arriba o abajo, pero no en diagonal:



*Ejemplo de  
solución*

A	B	C
D	E	F
G	G	I

$A \rightarrow B \rightarrow E \rightarrow F \rightarrow I$

## Actividad:

- Crear la relación de los nodos, y programarla
- Mostrar el grafo correspondiente que represente la unión de todos los nodos
- Usar el algoritmo de búsqueda en profundidad y en amplitud para encontrar todos los caminos hasta el nodo “I” (Imprimir todos)
- Imprimir otra vez aquellos que sean los caminos mas cortos del nodo “A” al “I”
- Imprimir cuál algoritmo tomó una cantidad menor de búsquedas para resolverse



## Ejercicio 2:

Usar los algoritmos de búsqueda en amplitud y en profundidad para llegar de la casilla A a la H, y encontrar TODAS las rutas, para el siguiente tablero con obstáculos.

A	B	C		D
E	F		G	H
I	J		K	L
M	N	Ñ		O
P		Q	R	S

*Ejemplo de*



UNIVERS *solución*  
Panamericana

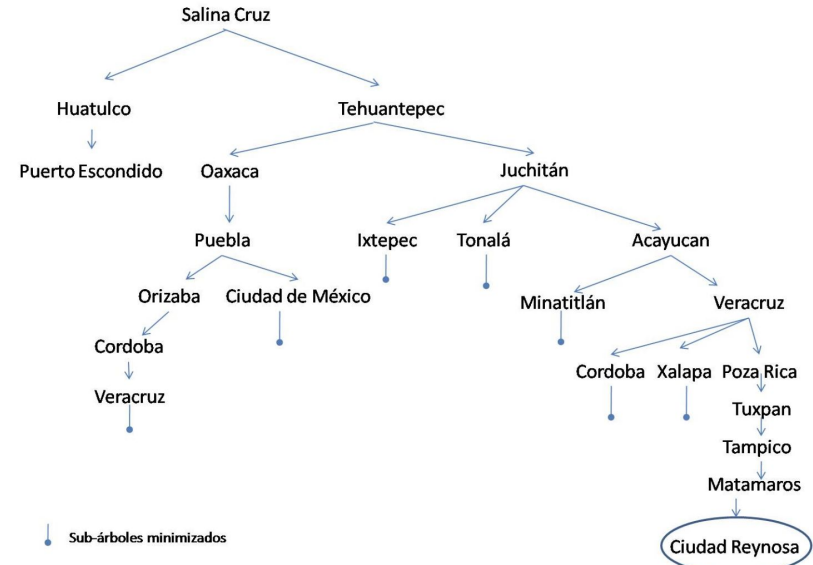
1				
2				12
3	4			11
	5	6		10
		7	8	9

### Actividad:

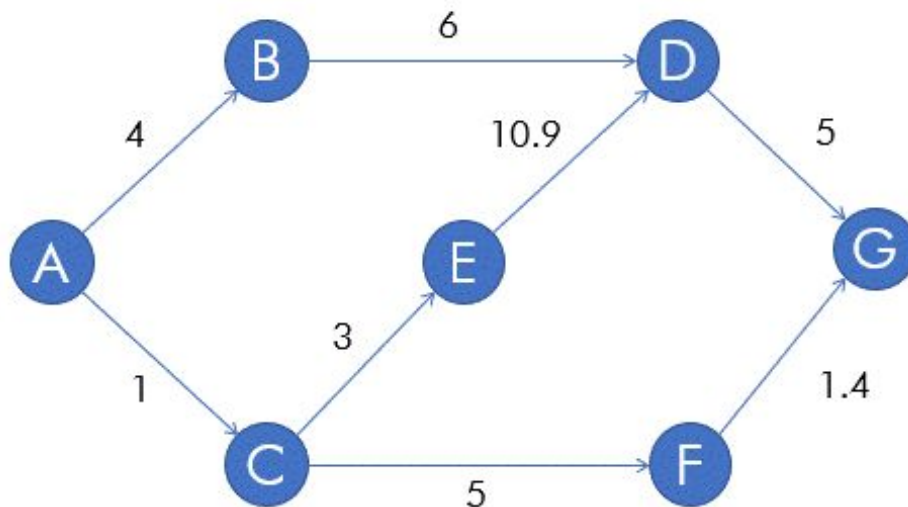
- Crear la relación de los nodos, y programarla
- Mostrar el grafo correspondiente que represente la unión de todos los nodos
- Usar el algoritmo de búsqueda en profundidad y en amplitud para encontrar todos los caminos hasta el nodo "I" (Imprimir todos)
- Imprimir otra vez aquellos que sean los caminos mas cortos del nodo "A" al "I"
- Imprimir cuál algoritmo tomó una cantidad menor de búsquedas para resolverse

# Ejercicio de mapeo de rutas

Resolvamos el siguiente ejemplo, utilizando búsqueda en profundidad (Y el alumno agregará la búsqueda en amplitud:

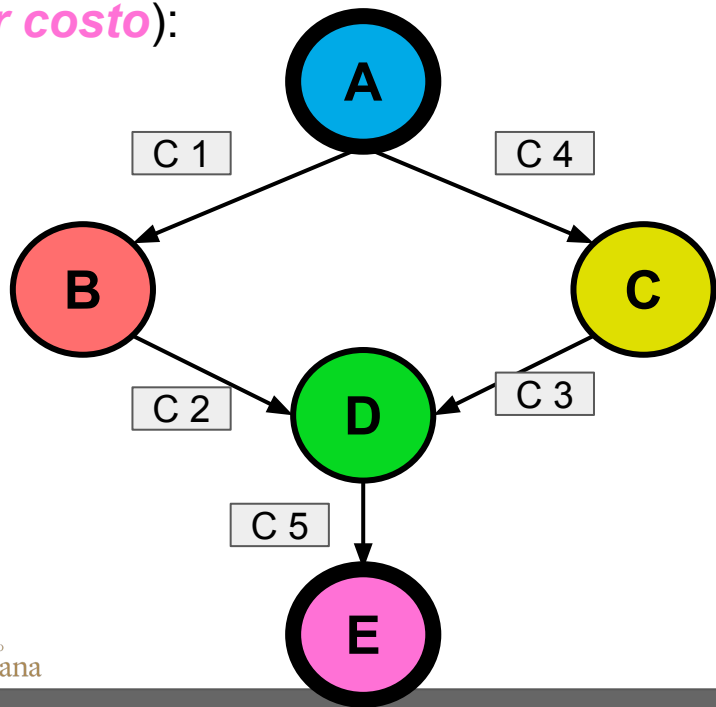


# Búsqueda de costo uniforme



# Algoritmo de búsqueda de costo uniforme

El Algoritmo de búsqueda de costo uniforme, contempla problemas donde se sabe que **hay un costo o esfuerzo diferente** (tiempo, dinero, distancia, movimientos, etc) para llegar de un estado o nodo a otro. Y busca mapear la mejor ruta (**con el menor costo**):



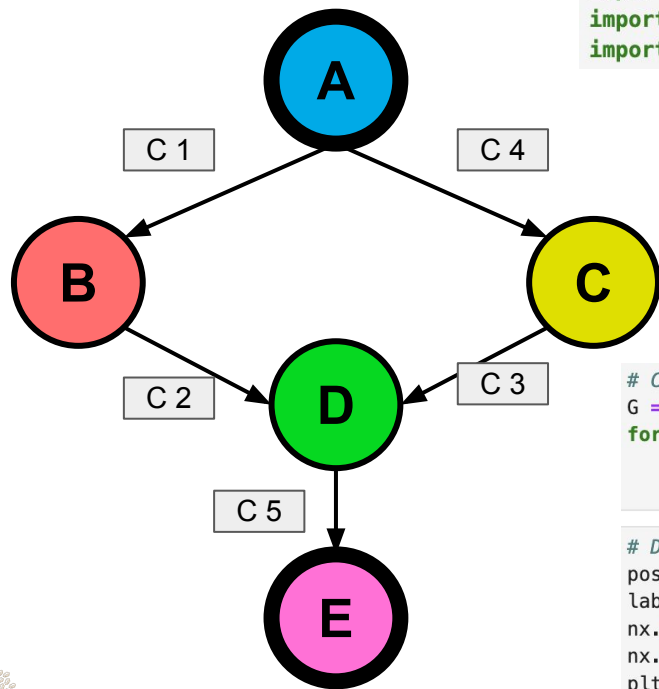
*En este ejemplo, la mejor ruta es la:*

*A -> B -> D -> E*

*La cual tiene un costo de 8  
(1 + 2 + 5)*

# Ejemplo de búsqueda de costo uniforme

Resolvamos el siguiente ejemplo, utilizando python:



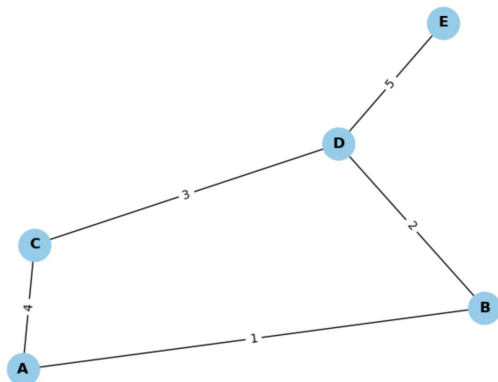
```
# Importamos librerías
import heapq
import networkx as nx
import matplotlib.pyplot as plt
```

```
# Grafo representado como un diccionario de diccionarios
grafo = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'D': 2},
    'C': {'A': 4, 'D': 3},
    'D': {'B': 2, 'C': 3, 'E': 5},
    'E': {'D': 5}
}
```

```
# Crear un grafo de NetworkX y agregar nodos y aristas
G = nx.Graph()
for nodo, vecinos in grafo.items():
    for vecino, peso in vecinos.items():
        G.add_edge(nodo, vecino, weight=peso)
```

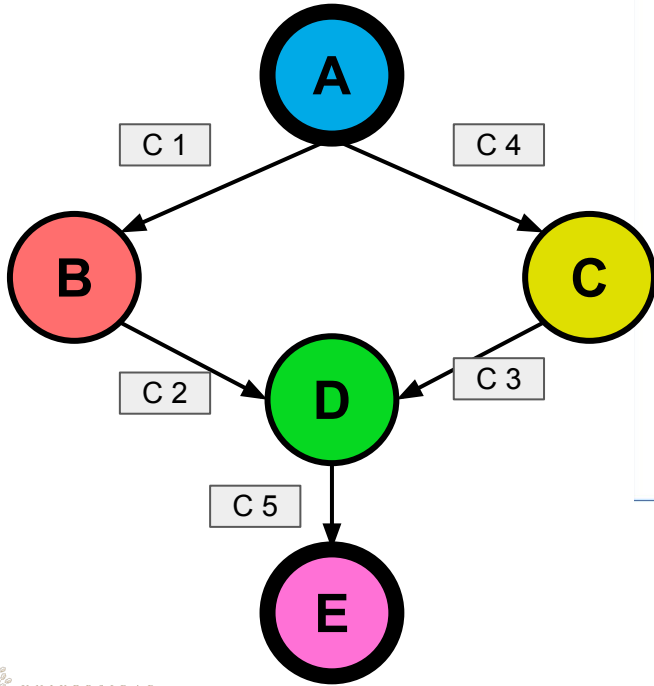
```
# Dibujar el grafo
pos = nx.spring_layout(G) # Creamos la red de nodos
labels = nx.get_edge_attributes(G, 'weight') # Asignamos los pesos (valores) que se verán en la red
nx.draw(G, pos, with_labels=True, node_size=700, node_color='skyblue', font_size=12, font_weight='bold')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.title('Grafo de ejemplo')
plt.show()
```

Grafo de ejemplo



# Ejemplo de búsqueda de costo uniforme

Resolvamos el siguiente ejemplo, utilizando python:



```
# Algoritmo de costo uniforme
def costo_uniforme(inicio, objetivo, grafo):
    cola_prioridad = []
    heapq.heappush(cola_prioridad, (0, inicio, [inicio])) # Un heap es una estructura de datos
                                                         # basada en un árbol binario
                                                         # heapq.heappush(heap, item):
                                                         # Agrega un nuevo elemento al montón.

    visitados = set()

    while cola_prioridad:
        costo_actual, nodo_actual, camino = heapq.heappop(cola_prioridad)
        # heapq.heappop(heap):

        # Calcular el costo mínimo y el camino usando búsqueda de costo uniforme
        inicio = 'A'
        objetivo = 'E'
        costo_minimo, camino_minimo = costo_uniforme(inicio, objetivo, grafo)

        # Imprimir el costo mínimo y el camino
        if costo_minimo != float('inf'):
            print(f"El costo mínimo desde {inicio} hasta {objetivo} es: {costo_minimo}")
            print(f"El camino mínimo es: {' -> '.join(camino_minimo)}")
        else:
            print(f"No se encontró un camino desde {inicio} hasta {objetivo}")

    print("-----")
    print("Nodos visitados", visitados)
    print("Nodo Actual:", nodo_actual)
    print("Costo Actual:", costo_actual)
    print("Camino:", camino)

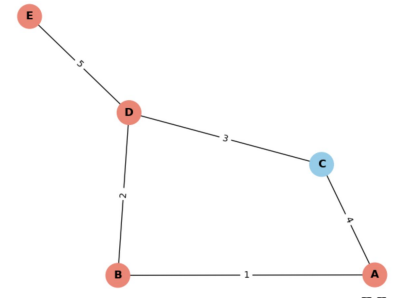
    if nodo_actual == objetivo:
        return costo_actual, camino

    if nodo_actual not in visitados:
        visitados.add(nodo_actual)
        for vecino, costo in grafo[nodo_actual].items():
            if vecino not in visitados: # Visitamos solo los vecinos NO visitados
                nuevo_costo = costo_actual + costo
                nuevo_camino = camino + [vecino]
                heapq.heappush(cola_prioridad, (nuevo_costo, vecino, nuevo_camino))

    print("Cola:", cola_prioridad)

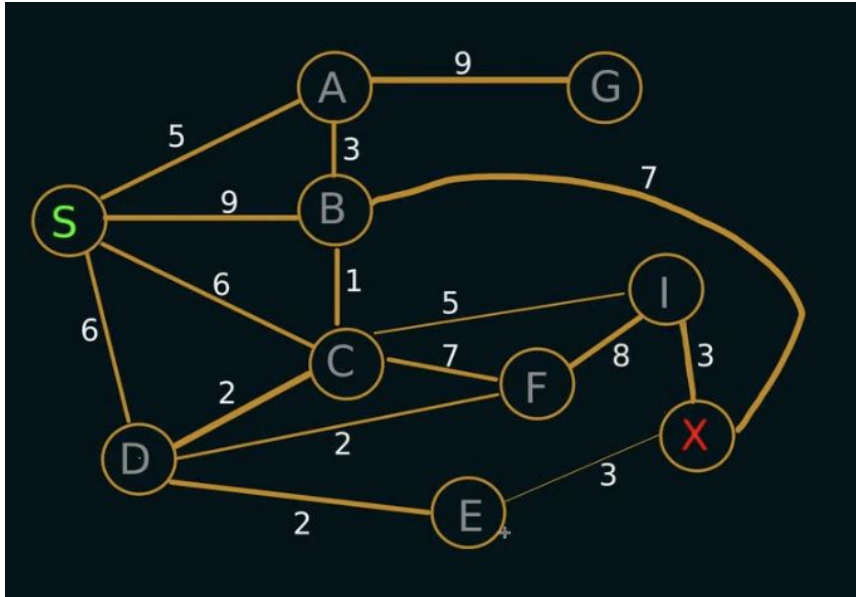
    return float('inf'), []
```

Grafo de ejemplo con el camino mínimo coloreado



# Ejercicio:

Resolver el siguiente grafo utilizando el algoritmo de búsqueda de costo uniforme:



## Actividad:

- Se desea encontrar el mejor camino de "S" a "X" (Menor costo)
- Resolver el ejercicio manualmente.
- Crear la relación de los nodos, y programarla
- Mostrar el grafo correspondiente que represente la unión de todos los nodos
- Usar el algoritmo de búsqueda de costo uniforme e imprimir el mejor camino y su costo final.

