Tema 2. Algoritmos en Grafos

Definiciones

Se suele definir un grafo $G = (V, E)$ como un conjunto de vértices V y de aristas $E \subseteq VxV$. Usualmente, la complejidad de los algoritmos sobre grafos suele medirse en función del número de vértices $|V|$ y número de aristas $|E|$. Por convenios de notación cuando nos refiramos a estas cantidades en notación asintótica, sustituiremos $|V|$ y $|E|$ por V y E. Ejemplo: $O(VE)$ en lugar de $O(|V||E|)$

Existen dos representaciones típicas tanto en grafos no dirigidos como en dirigidos:

- Lista de adyacencia: Representación compacta para grafos dispersos ($|E| << |V|^2$). No aseguran un acceso rápido a la hora de comprobar si hay una arista entre dos vértices dados.
- Matriz de adyacencia: Aseguran un acceso rápido a la hora de comprobar si hay una arista entre dos vértices dados. Se requiere una memoria de $\Theta(V^2)$, y no depende de la densidad del grafo. Para grafos densos (aquéllos en los que $|E| \cong |V|^2$) ya no se desperdicia tanta memoria.

Recorrido en profundidad

El recorrido en profundidad (*depth-first search*) tiene como estrategia profundizar en el grafo siempre que sea posible. Dado un vértice, antes de visitar su hermano se visita a su hijo (equivalente a un recorrido en pre-orden). Suele implementarte se manera recursiva. Se tiene que incluir un conjunto de vértices visitados para evitar ciclos en la búsqueda.

```
# Recorrido en profundidad recursivo
def dfs_rec_aux(node, graph, visited):
    print("Visiting node " + str(node))
    visited.add(node)
    for neigh in graph[node - 1]:
        if neigh not in visited:
            dfs_rec_aux(neigh, graph, visited)

def dfs_rec(graph):
    print("Recorrido en Profundidad")
    node = 1
    visited = set()
    if node not in visited:
        dfs_rec_aux(node, graph, visited)

# Recorrido en profundidad iterativo
def dfs_it_aux(graph, node, visited):
    q = deque()
    q.appendleft(node-1)
    while q:  # Si una estructura de datos esta vacía devolverá False y sino True
        aux = q.popleft()
        if aux not in visited:
            print("Visiting node " + str(aux+1))
            visited.add(aux)
```

```
                for adj in graph[aux]:
                    if adj-1 not in visited:
                        q.appendleft(adj-1)

def dfs_it(graph):
    print("Recorrido en Profundidad iterativo")
    visited = set()
    # ncc = 0
    for node in range(1, len(graph)):
        if node-1 not in visited:
                # ncc += 1
                # print(f"Componente conexa {ncc}:")
            dfs_it_aux(graph, node, visited)

# Alternativa de recorrido en profundidad iterativo (Una sola funcion)
def dfs_it_alt(graph):
        print("Recorrido en Profundidad iterativo en una sola funcion")
    visitados = set()
    pila = []
    origen = 0
    pila.append(origen)
    while pila:
        actual = pila.pop()
        if actual not in visitados:
            print("Visiting node " + str(actual+1))
            visitados.add(actual)
                for key in graph[actual]:
                    if key-1 not in visitados:
                        pila.append(key-1)

# Main
GND_adj_list = [[2, 4, 8],
                [1, 3, 4],
                [2, 4, 5],
                [1, 2, 3, 7],
                [3, 6],
                [5, 7],
                [4, 6, 9],
                [1, 9],
                [7, 8]]
dfs_rec(GND_adj_list)
dfs_it(GND_adj_list)
dfs_it_alt(GND_adj_list)
```

Análisis de la complejidad $\rightarrow$ Cada vértice se visita una única vez $\Rightarrow n$ llamadas al procedimiento $rp \Rightarrow \Theta(V)$. El algoritmo examina todas las aristas $\Rightarrow \Theta(E)$. La complejidad global del algoritmo es $\Theta(max(V, E))$.

El recorrido en profundidad de un grafo conexo G crea un árbol de recubrimiento T. Las aristas de T son un subcojunto de las aristas de G. La raíz de T es el punto de partida de la exploración de G. Si el grafo no es conexo, se obtiene un árbol por cada componente conexa (bosque). La exploración en profundidad de un grafo visita las nodos del grafo en "pre-orden".

Recorrido en anchura

El recorrido en anchura (*breath-first search*) se llama así porque recorre la frontera en anchura. Visita todos los vértices a una distancia k antes de descubrir el primer vértice a la distancia $k+1$. Suele implementarte se manera iterativa. Se tiene que incluir una cola con los vértices visitados para evitar ciclos y establecer el orden en la búsqueda.

Dado un grafo $G = (V, E)$ y un vértice inicial s:

- Calcula la distancia (menor número de vértices) desde s hasta los vértices alcanzables
- Produce un árbol de recorrido en anchura donde la raíz es s

- Para cualquier vértice v alcanzable desde s, la ruta en el árbol de recorrido en anchura desde s hasta v es el camino más corto entre esos dos vértices
- Es un algoritmo adecuado para grafos dirigidos y no dirigidos

```
from collections import deque
# Recorrido en anchura de manera recursiva
def bfs_rec_aux(graph, visited, q):
    while q:
        actual = q.popleft()
        for neigh in graph[actual]:
            if neigh-1 not in visited:
                print("Visiting node " + str(neigh))
                visited.add(neigh-1)
                q.append(neigh-1)
        bfs_rec_aux(graph, visited, q)

def bfs_rec(graph):
    print("Recorrido en Anchura recursivo")
    node = 1
    visited = set()
    q = deque()
    if node-1 not in visited:
        print("Visiting node " + str(node))
        visited.add(node-1)
        q.append(node-1)
        bfs_rec_aux(graph, visited, q)


# Recorrido en anchura de manera iterativa
def bfs_it_aux(graph, node, visited):
    q = deque()
    print("Visiting node " + str(node))
    visited[node - 1] = True
    q.append(node)
    while q:  # Si una estructura de datos esta vacía devolverá False y sino True
        aux = q.popleft()
        for neigh in graph[aux - 1]:
            if not visited[neigh - 1]:
                q.append(neigh)
                visited[neigh - 1] = True
                print("Visiting node " + str(neigh))

def bfs_it(graph):
        size = len(graph)
    print("Recorrido en Anchura")
    visited = [False] * size
    # ncc = 0
    for neigh in range(1, size):
        if not visited[neigh]:
                # ncc += 1
                # print(f"Componente conexa {ncc}:")
            bfs_it_aux(graph, neigh, visited)

# Alternativa al recorrido en anchura iterativo con una sola funcion
def bfs_it_alt(graph):
    print("Alternativa recorrido en Anchura iterativo en una sola funcion")
    q = deque()
    visited = set()
    nodo = 1
    q.append(nodo-1)
    while q:
        actual = q.popleft()
        if actual not in visited:
            print("Visiting node " + str(actual+1))
```

```
            visited.add(actual)
            for adj in graph[actual]:
                if adj-1 not in visited:
                    q.append(adj-1)

# Main
GND_adj_list = [[2, 4, 8],
                [1, 3, 4],
                [2, 4, 5],
                [1, 2, 3, 7],
                [3, 6],
                [5, 7],
                [4, 6, 9],
                [1, 9],
                [7, 8]]
bfs_rec(GND_adj_list)
bfs_it(GND_adj_list)
bfs_it_alt(GND_adj_list)
```

Análisis de complejidad → El tiempo requerido para realizar un recorrido en anchura de un grafo es el mismo que para un recorrido en profundidad ⇒ $\Theta(max(V, E))$. También genera árboles de recubrimiento. Si el grafo es conexo ⇒ un único árbol. Se emplea en exploraciones parciales de grafos, para hallar el camino más corto entre dos puntos de un grafo, etc.

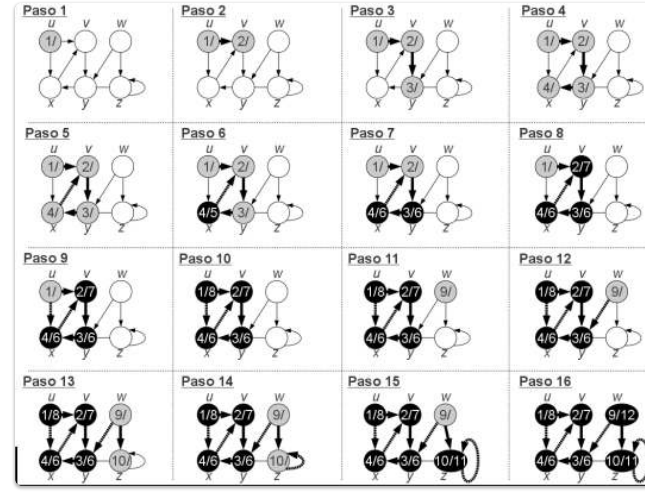
Algoritmos sobre grafos

Ordenación topológica

Dado un grafo dirigido y acíclico, se denomina ordenación topológica a una disposición lineal de los nodos tal que, dado un arco (u, v), el nodo u esté antes que v en la ordenación. Un vértice se visita sí y sólo sí se han visitados todos sus predecesores. Algunas aplicaciones prácticas son la representación de fases de un proyecto (PERT) y la evaluación de atributos en la fase semántica de un compilador.

Recorrido en profundidad. Distancia y finalización. Los vértices se colorean como sigue:

- Inicialmente son todos blancos
- Cuando se visitan, se colorean de gris (denotado por $d[v]$)
- Cuando toda la adyacencia se ha visitado se colorea de negro (denotado por $f[v]$)



Propiedades:

- Un vértice es blanco hasta $d[v]$, gris desde $d[v]$ hasta $f[v]$ y negro en adelante
- $d[v]$ y $f[v]$ (con $d[v] < f[v]$ para todo v) toman valores entre 1 y $2 * |V|$

```
from collections import deque

# Ordenación Topológica
def toposortVisit(data, node):
    data["time"] += 1
    data["state"][node] = "VISITED"
    data["discovered"][node] = data["time"]
    for adj in data["g"][node]:
        if data["state"][adj] == "NOT_VISITED":
            data["parent"][adj] = node
            toposortVisit(data, adj)
    data["state"][node] = "FINISHED"
```

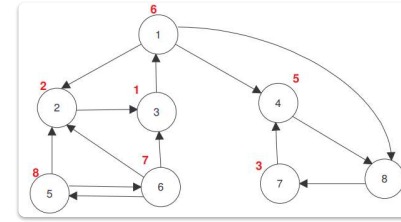
```
    data["time"] += 1
    data["finalized"][node] = data["time"]
    data["queue"].appendleft(node)

def toposort_rec(g):
    n = len(g)
    data = {
        "g": g,
        "state": {},
        "time": 0,
        "parent": {},
        "discovered": {},
        "queue": deque(),
        "finalized": {}
    }
    for node in g.keys():
        data["state"][node] = "NOT_VISITED"
        data["parent"][node] = None
        data["discovered"][node] = 0
        data["finalized"][node] = 0
    for node in g.keys():
        if data["state"][node] == "NOT_VISITED":
            toposortVisit(data, node)
    print(data["queue"])

# Alternativa Ordenación Topológica iterativa
def toposort_it(g):
    recorrido = "" # String con el path
    aristas_entrantes = {i: 0 for i in g} # Diccionario con el numero de aristas entrantes por cada clave (nodo
del grafo)
    # Calculo las aristas entrantes de cada nodo del grafo
    for node in g:  # por cada nodo del grafo
        for neigh in g[node]:  # Por cada vecino del nodo
            aristas_entrantes[neigh] += 1  # sumo 1 a las aristas entrantes del vecino
        # Busco los nodos iniciales (aquellos sin predecesor)
    nodos_iniciales = []
    for i in g:  # Por cada nodo
        if aristas_entrantes[i] == 0:
            nodos_iniciales.append(i) # Si no tiene ninguna arista entrante es nodo inicial

    while nodos_iniciales:  # Mientras haya nodos iniciales
        origen = nodos_iniciales.pop()
        recorrido += f"{origen} "
        for adj in g[origen]:  # Por cada vecino del nodo inicial
            aristas_entrantes[adj] -= 1  # Le resto una arista entrante
            if aristas_entrantes[adj] == 0:
                nodos_iniciales.append(adj)  # Si se queda sin aristas entrantes lo añado a los nodos iniciales
    return recorrido.strip()

# Main
GD = {
    "calcetines": ["zapatos"],
    "pantalon": ["zapatos", "cinturon"],
    "camisa": ["cinturon", "jersey"],
    "zapatos":[],
    "cinturon": [],
    "jersey": []
}

toposort_rec(GD)
print(toposort_it(GD))
```

Componentes fuertemente conexas (Grafos dirigidos)

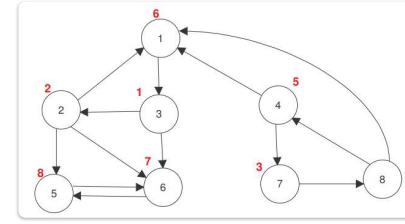
Un grafo dirigido es fuertemente conexo si para todo par de vértices diferentes, u y v, existe camino de u a v ($u \rightsquigarrow v$) y existe camino de v a u ($v \rightsquigarrow u$). Para descomponer un grafo G en sus componentes fuertemente conexas se usa el siguiente algoritmo:

1. Aplicar ordenación topológica sobre G. Recorrido en profundidad de G etiquetando los vértices con la numeración en el orden inverso del recorrido.
2. Calcular el grafo traspuesto G^T (el que se obtiene invirtiendo el sentido de todas las aristas de G). Las componentes fuertemente conexas de ambos grafos, G y G^T, son las mismas.
3. Aplicar búsqueda en profundidad sobre G^T iniciando la búsqueda en los nodos de mayor a menor tiempo de finalización (en orden decreciente de las etiquetas) obtenidos en la primera ejecución de búsqueda en profundidad.
4. El resultado será un bosque de árboles. Cada árbol es un componente fuertemente conexo.

Un ejemplo para acabar. Sea $G =< V, E >$ el grafo de la figura,



1. Lanzamos un recorrido en profundidad (o los que hagan falta) y asociamos a cada vértice la numeración en el orden inverso que le corresponde en ese recorrido (valor en rojo fuera del nodo):
2. Invertimos las aristas de G y obtenemos G^T:
3. Recorrido en profundidad comenzando por el vértice con la mayor numeración en el orden inverso (vértice con etiqueta 5 y numeración 8) que genera el árbol: $5 \rightarrow 6$
4. Recorrido en profundidad comenzando por el vértice con la mayor numeración en el orden inverso (vértice con etiqueta 6 y numeración 6) que genera el árbol: $1 \rightarrow 3 \rightarrow 2$
5. Idem comenzando por el vértice con etiqueta 4 y numeración 5 que genera el árbol: $4 \rightarrow 7 \rightarrow 8$



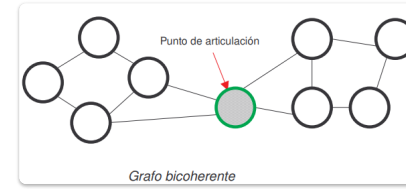
Y ya tenemos las 3 CFC calculadas.

```
def invert_graph(grafo):
    graph_t = {i: [] for i in grafo}
    for i in grafo:
        for j in grafo[i]:
            graph_t[j].append(i)
    return graph_t

def dfs_count(grafo, path):
    recorrido = []
    ncc = 0
    visited = [False] * len(grafo)
    for i in (grafo if path is None else path):
        if not visited[i - 1]:
            ncc += 1
            pila = []
            pila.append(i)
            while pila:
                origen = pila.pop()
                if not visited[origen - 1]:
                    visited[origen - 1] = True
                    if path is None: recorrido.append(origen)
                    for adj in grafo[origen]:
                        if not visited[adj - 1]:
                            pila.append(adj)
    if path is None:
        return recorrido
    else:
        return ncc

def cfc(g):
    path = dfs_count(g, None)
    graph_t = invert_graph(g)
    path = list(reversed(path))
```

```
    return dfs_count(graph_t, path)

# Main
graph = {1: [2, 4, 8], 2: [3], 3: [1], 4: [8],
         5: [2, 6], 6: [2, 3, 5], 7: [4], 8: [7]}
print(cfc(graph))
```

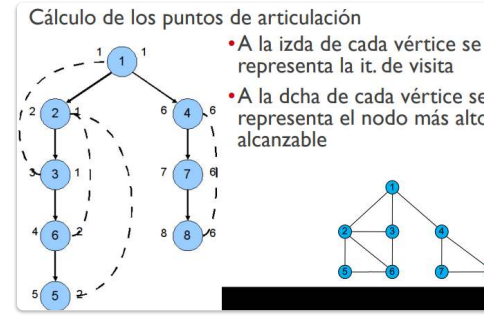
Puntos de articulación

- Puntos de articulación: un vértice v de un grafo conexo es un punto de articulación si el subgrafo que se obtiene al eliminarlo (junto con sus aristas) es no conexo.
- Grafo biconexo (o no articulado): un grafo es biconexo si es conexo y no tiene puntos de articulación. En una red que resulte ser un grafo biconexo aunque falle un nodo, el resto de la red sigue manteniendo la conectividad.
- Grafo bicoherente (o 2-arista conexo): Un grafo bicoherente es un grafo en el que todo punto de articulación está unido mediante al menos dos aristas con cada una de las componentes del grafo que quedarían al eliminar el vértice punto de articulación. Una red con estas características sigue funcionando aunque falle una línea de la red (una arista).



Cálculo de los puntos de articulación

Refinando la idea y suponiendo que cada vértice 'sabe' hasta donde puede subir (para ello usaremos un vector indexado por número de vértice, `masalto`, que contiene la numeración en el orden del recorrido del vértice más alto al que puede ascender), podemos hacer el siguiente análisis:
Sea u un vértice cualquiera de V:



- SI u es la raíz del árbol y tiene más de un hijo, es un punto de articulación.
- Si u no tiene hijos en el TDFS, entonces u NO es punto de articulación.
- Si u tiene hijos en el TDFS, sea x un hijo de u tal que:
  - $masalto[x] < u.num - dfs$, si eliminamos el vértice u entonces los vértices del subárbol cuya raíz es x NO quedan desconectados del resto del árbol porque partiendo de x existe una cadena de aristas de G, que no incluye la arista (u, x), y que nos lleva por encima de u.
  - $masalto[x] \geq u.num - dfs$, si eliminamos u los vértices del subárbol cuya raíz es x quedan desconectados del resto del árbol porque partiendo de x no existe una cadena de aristas en G que nos lleve por encima de u.

Entonces, u es punto de articulación si y sólo si tiene un hijo x tal que cumple la condición que $masalto[x] \geq u.num - dfs$.

Algoritmo:

- Realizar un recorrido en profundidad (modificado) de G
  - Obtener el árbol de recubrimiento
  - Calcular la iteración de descubrimiento de cada nodo del árbol
- Calcular $masalto[u]$ como
  - Bajando (todo lo que se necesite) por las aristas usadas en el árbol de recubrimiento
  - Subiendo (por sólo una) arista que no pertenezca al árbol
- Los puntos de articulación de G serán
  - La raíz sí y sólo sí tiene más de un hijo
  - Cualquier nodo u que tenga un hijo x tal que $masalto[x] \geq u.num - dfs$