# Recuperatorio Parcialito N°3

Ivan Litteri - 106223

08/03/2021

## Ejercicio 1

### Consigna

Contamos con un grafo dirigido que modela un ecosistema. En dicho grafo, cada vértice es una especie, y cada arista (v, w) indica que v es depredador natural de w. Considerando la horrible tendencia del ser humano por llevar a la extinción especies, algo que nos puede interesar es saber si existe alguna especie que, si llegara a desaparecer, rompería todo el ecosistema: quienes la depredan no tienen un sustituto (y, por ende, pueden desaparecer también) y/o quienes eran depredados por esta ya no tienen amenazas, por lo que crecerán descontroladamente. Implementar un algoritmo que reciba un grafo de dichas características y devuelva una lista de todas las especies que cumplan lo antes mencionado. Indicar y justificar la complejidad del algoritmo implementado.

### Resolución

```python
def _obtener_especies_en_peligro(ecosistema: Grafo, v, visitados: set, es_raiz: bool, orden: dict, mas_bajo: dict, especies_en_extincion: list) -> None:
    visitados.add(v)
    hijos = 0
    for w in ecosistema.adyacentes(v):
        if w not in visitados:
            orden[w] = orden[v] + 1
            hijos += 1
            _obtener_especies_en_peligro(ecosistema, w, visitados, False, orden, mas_bajo, especies_en_extincion)
            if mas_bajo[w] >= orden[v]:
                especies_en_extincion.append(v)
            mas_bajo[v] = min(mas_bajo[v], mas_bajo[w])
        else:
            mas_bajo[v] = min(mas_bajo[v], orden[w])
    if es_raiz and hijos > 1:
        especies_en_extincion.append(v)

def copiar_grafo(grafo_original: Grafo) -> Grafo:
    copia = Grafo(es_dirigido=False)

    # Copia los vertices
    for v in grafo_orignial:
        copia.agregar_vertice(v)

    # Copia las aristas
    for v in grafo_original:
        for w in grafo_original.adycentes(v):
            copia.agregar_arista(v, w)

    return copia

```

```python
def especies_en_peligro_de_extincion(grafo: Grafo) -> list:
    # Copia el grafo original a uno no dirigido para poder utilizar el algoritmo de punto de articulacion.
    copia_grafo = copiar_grafo(grafo)
    especies_en_peligro = []
    v = copia_grafo.vertice_aleatorio()
    # Busca los puntos de articulacion en la copia del grafo original y los va agregando en la lista a medida que los encuentra.
    _obtener_especies_en_peligro(copia_grafo, v, set(), True, {v: 0}, {v: 0}, especies_en_peligro)
    return especies_en_peligro

/* Complejidad
    Copiar el grafo original cuesta O(V + E).
    Y la funcion que obtiene las especies en peligro de extincion que pueden
llegar a romper el ecosistema, es el algoritmo para encontrar vertices que son
puntos de articulacion, y cuesta O(V + E) porque hace un recorrido bfs (que
cuesta O(V + E)).
    Entonces la complejidad de toda la funcion es O(V + E) + O(V + E) = O(2V + 2E) =
= O(2 (V + E)) = O(V + E).
*/
```

# Ejercicio 2

## Consigna

¡Se acaba la vida en la Tierra! La NASA construyó una nave espacial para continuar viviendo en Marte, el problema es que la capacidad de la misma (M) es reducida. El presidente de la NASA, "Ja Mao", decidió que solo entrarán en la nave las M personas con el nombre más corto. Implementar en C una función que reciba un arreglo con el nombre de todas las personas del Planeta Tierra, y su largo, N; y que devuelva una lista con los pasajeros habilitados, en O(N + M log N). Justificar el orden del algoritmo.

## Resolución

```c
int nombre_mas_corto(const void *a, const void *b)
{
    if (a == NULL && b == NULL)
    {
        return 0;
    }

    else if (a == NULL || b == NULL)
    {
        return a == NULL ? 1 : -1;
    }

    size_t largo_a = strlen((char *)a);
    size_t largo_b = strlen((char *)b);

    return largo_a == largo_b ? 0 : largo_a > largo_b ? -1 : 1;
}

lista_t *pasajeros_habilitados(char **personas, size_t n, size_t m)
{
    heap_t *pasajeros_ordenados;

    // Crea un heap a partir del arreglo original.
    if ((pasajeros_ordenados = heap_crear_arr(personas, n, nombre_mas_corto)) == NULL)
    {
        return NULL;
```

```
    } // O(n)

    lista_t *pasajeros_habilitados;

    // Crea una lista para enlistar a los pasajeros habilitados.
    if ((pasajeros_habilitados = lista_crear(NULL)) == NULL)
    {
        heap_destruir(pasajeros_ordenados);
        return NULL;
    }

    // Enlista "m" pasajeros.
    while m--
    {
        if (!lista_insertar_ultimo(heap_desencolar(pasajeros_ordenados)))
        {
            heap_destruir(pasajeros_ordenados);
            lista_destruir(pasajeros_habilitados);
            return NULL;
        }
    } // O(m log n)

    heap_destruir(pasajeros_ordenados);

    return pasajeros_habilitados;
}

/* Complejidad
    Transformar el arreglo en heap cuesta O(n) ya que la primitiva utiliza heapify y este es
O(n) siendo n los elementos dentro del arreglo.
    Desencolar el heap en la lista cuesta O(m log n), porque densencolar del heap cuesta
O(log n) siendo n la cantidad de elementos encolados, e inserto cada vez que desencolo del
heap en la lista, lo cual lo hago "m" veces, por lo tanto O(m log n) siendo m la cantidad de
personas que entraran en la nave y "n" la cantidad de personas encoladas en espera en el
    heap.
    Finalmente el total de costo de la funcion es el costo de encolar a todos en el heap y
    el
de desencolar a "m" del heap en la lista, por lo tanto O(m + m log n).
*/
```

# Ejercicio 3

## Consigna

Definir si las siguientes afirmaciones son verdaderas o falsas. Justificar.

a) En un grafo bipartito no pueden haber ciclos con cantidad impar de vértices que lo compongan.

b) En un árbol (grafo no dirigido, conexo y sin ciclos) todos los vértices con al menos dos adyacentes son puntos de articulación.

c) En un grafo dirigido, no existe camino de un vértice v de una componente fuertemente conexa hacia un vértice w de otra componente fuertemente conexa.

## Resolución

a) **VERDADERO**. Si un grafo contiene un ciclo de $n$ vértices, o es un ciclo de $n$ vértices, si $n$ es par entonces si es bipartito porque puedo colorear los vértices del mismo con 2 colores, obteniendo dos conjuntos disjuntos de vértices de un mismo conjunto que no se relacionan entre sí. Usando como ejemplo Un ciclo de 2 vértices con aristas: $V = \{A, B\}, E = \{(A, B), (B, A)\}$ puedo usar un color

para A y otro para B y obtener dos conjuntos disjuntos $color1 = \{A\}, color2 = \{B\}$. Usando como ejemplo un ciclo con $n$ impar: $V = \{A, B, C\}, E = \{(A, B), (A, C), (B, C)\}$, si coloreo con 2 colores alternando en el ciclo (para verificar que sea bipartito), me encuentro con que hay 2 vértices de un color relacionándose entre sí, cosa que no cumple con la definición de grafo bipartito.

b) **VERDADERO**. En un árbol, todo vértice que tenga 2 adyacentes es punto de articulación, porque, es el único vértice por el cuál se puede llegar a sus adyacentes (por definición de árbol).

c) **FALSO**. En un grafo dirigido, puede no existir un camino de un vértice $V$ a de una componente fuertemente conexa hacia un vértice $W$ de otra componente fuertemente conexa, pero es no es cierto para todo par de vértices $V, W$. Ejemplo: tengo una componente fuertemente conexa formada por vértices $A, B, C$ y otra componente fuertemente conexa formada por $E, F, G$, existe una arista $(B- > E)$, gracias a esta arista, existe un camino desde un vértice $V$ de la primera componente conexa a cualquier vértice $V$ de la segunda componente conexa.