Parte 1

Ejercicio 1

Implementar la función crear grafo que dada una lista de vértices y una lista de aristas cree un grafo con la representación por Lista de Adyacencia.

```python
def initialize(cls, nodes, key_pairs=[], dir_graph=False):
    # vertices: list(GraphNode)
    # key_pairs: list(node_key_0, node_key_1), adjacent nodes
    # dir_graph: The graph could be constructed as directed

    graph = Graph()
    # Inserts all the nodes
    for node in nodes:
        graph.add(node)

    # Creates the nodes link
    if not dir_graph:

        for (node_key0, node_key1) in key_pairs:
            graph.link(node_key0, node_key1)
    else:
        for (node_key0, node_key1) in key_pairs:
            graph.link_dir(node_key0, node_key1)
    return graph
```

Todo el código básico relacionado con el grafo está implementado en una clase, donde link(u, v) es un método que crea la arista (u,v) y la (v, u). De manera similar link_dir(u, v) solamente crea la arista (u, v)

Ejercicio 2

Implementar la función exists Path

```python
def exists_path(graph, node0, node1):
    """Returns if there is a path that connects the vertices"""
    # Uses a similar algorithm to Breadth-First-Search

    starting_node = graph[node0]
    end_node = graph[node1]

    queue = [starting_node]
    traversed_keys = set()

    while len(queue):
        node = queue.pop(0)
        if node.key == end_node.key:
            return True
        # Stores the traversed vertex in a dictionary
        traversed_keys.add(node.key)
        # Loops throught all the adjacent vertices and adds them to the queue
        for adjacent_node in graph.get_adjacent(node):
            # If the vertex wasn't added to the queue, adds
            if adjacent_node.key not in traversed_keys:
                queue.append(adjacent_node)

    return False
```

Ejercicio 3

Implementación de una función que determina si un grafo es conexo

```python
def is_connected(graph):
    """
    Uses a DFS approach. With DFS we can count the amount of
    connected components in the main for loop. If the counter is
    greater than 1, it means it's not connected
    """
    def _visit_node(graph, node, visited_keys):
        # Recursive function, used to visit all the adyacent nodes
        if node.key in visited_keys:
            return

        visited_keys.add(node.key)

        for adjacent_node in graph.get_adjacent(node):
            _visit_node(graph, adjacent_node, visited_keys)


    visited_keys = set()
    components = 0
    for node in graph.nodes:

        if node.key not in visited_keys:

            # More than one connected component
            if components == 1:
                return False

            components += 1
            _visit_node(graph, node, visited_keys)

    return True
```

Ejercicio 4

Implementación de una función que determina si un grafo es un árbol, es decir que es **conexo** y **no tiene ciclos**

```python
def is_tree(graph: Graph):
    """
    Uses a DFS approach.
    Cycle detection: we can check if any of the adjacent nodes
        have already been visited, and if it's not an immediate parent, it means
        we found a regession edge -> cycles -> not tree
    Connected: As in the is_connected(graph) implementation, counts the amount
        of connected components, and in a tree it should be just one
    """

    def visit_is_tree_node(graph, node, visited_keys, immediate_parent):

        # Node already visited
        if node.key in visited_keys:
            return True

        # Visited for the first time
        visited_keys.add(node.key)

        # Loops throught all the adjacent nodes
        for adjacent_node in graph.get_adjacent(node):

            if adjacent_node.key in visited_keys:

                # If the node is already visited and it's not it's parent
                # it means we found a regression edge, making a cycle, then it's not a tree
                if adjacent_node != immediate_parent: return False

            tree = visit_is_tree_node(graph, adjacent_node, visited_keys, node)
            if not tree: return False

        return True

    # Uses hash table
    visited_keys = set()
    components = 0

    for node in graph.nodes:

        if node.key not in visited_keys:

            # Not connected
            if components == 1:
                return False

            components += 1
            tree = visit_is_tree_node(graph, node, visited_keys, None)

        if not tree:
            return False

    return True
```

Ejercicio 4

Función que determina si un grafo es completo, es decir que existe una arista para cada par de vértices. En la implementación asumo que el grafo es simple y que no hay aristas repetidas

```python
def is_complete(graph):
    """
    A complete graph is an undirected graph in which every
    pair of distinct vertices is connected by a unique edge
    This implementation assumes that the graph is simple
    """
    for node in graph.nodes:

        if len(graph) - 1 != len(list(graph.get_adjacent(node))):
            return False

    return True
```

Ejercicio 6

Implementar una función que dado un grafo devuelva una lista de aristas que si se eliminan el grafo se convierte en un árbol. Respetar la siguiente especificación.

```python
def convert_tree(graph):
    """
    Returtns the list of edges that when removed, make the graph a tree
    Note it's almost the same algorithm as is_tree
    """

    def _visit(graph, node, visited_keys, remove_edges, immnediate_parent):

        if node.key in visited_keys:
            return

        visited_keys.add(node.key)

        for adjacent_node in graph.get_adjacent(node):

            if adjacent_node.key in visited_keys:

                if adjacent_node != immnediate_parent:
                    # Regression edge
                    remove_edges.append((node, adjacent_node))

            _visit(graph, adjacent_node, visited_keys, remove_edges, node)

    remove_edges = []
    visited_keys = set()

    for node in graph.nodes:
        _visit(graph, node, visited_keys, remove_edges, None)


    return remove_edges
```

<u>El problema se resuelve creando una lista con los arcos de retroceso</u>

Parte 2

Ejercicio 7

Implementar una función que retorna la cantidad de componentes conexas

```python
def connections_count(graph):

    """
    Returns the amount of connected components
    Uses DFS, and counts the component in the main loop
    """
    def _visit(graph, node, visited_keys):
        if node.key in visited_keys:
            return

        visited_keys.add(node.key)

        for adjacent_node in graph.get_adjacent(node):
            _visit(graph, adjacent_node, visited_keys)

    visited_keys = set()
    connections = 0

    for node in graph.nodes:

        if node.key not in visited_keys:
            connections += 1
            _visit(graph, node, visited_keys)

    return connections
```

Ejercicio 8

Implementar una función que crea un árbol de expansión usando búsqueda en anchura

```python
def breadth_first_search_tree(graph, root):
    """
    Returns a Tree
    """

    root_node = graph[root]

    tree_graph = Graph()
    tree_graph.add(root_node)

    # Stores the keys of the traversed nodes
    traversed_keys = set()
    traversed_keys.add(root_node.key)
    queue = [root_node]
    while len(queue):
        node = queue.pop(0)

        for adjacent_node in graph.get_adjacent(node):

            if adjacent_node.key not in traversed_keys:
                queue.append(adjacent_node)

                # Adds the node and links with it's parent
                tree_graph.add(adjacent_node)
                tree_graph.link(node, adjacent_node)
                traversed_keys.add(adjacent_node.key)

    return tree_graph
```

Ejercicio 9

Implementar una función que crea un árbol de expansión usando búsqueda en profundidad

```python
def depth_first_search_tree(graph : Graph, root):

    tree_graph = Graph()
    root_node = graph[root]

    visited_keys = set()

    time = 0

    # Visits the root
    dfs_visit(graph, tree_graph, root_node, time, visited_keys)

    # Visits the other nodes
    for node in graph.nodes:

        # Visits the node
        if node.key not in visited_keys:
            dfs_visit(graph, tree_graph, node, time, visited_keys)

    return tree_graph

def dfs_visit(graph, tree_graph, node, time, visited_keys):

    tree_graph.add(node)
    visited_keys.add(node.key)

    time += 1
    node.t0 = time

    for adjacent_node in graph.get_adjacent(node):

        # Backwards edge
        if adjacent_node.key not in visited_keys:
            dfs_visit(graph, tree_graph, adjacent_node, time, visited_keys)
            tree_graph.link(node, adjacent_node)

    time += 1
    node.t1 = time
```

Ejercicio 10

Implementar una función que retorna el camino más corto entre dos vértices

```python
def best_road(graph, start, end):
    """
    Returns a list of the edges that make the shortest path
    Uses a BFS approach.
    Stores all the posible paths inside a list, for each iteration,
    a new path is added to paths, until we find the end
    """

    start_node = graph[start]
    end_node = graph[end]

    if start_node == end_node:
        return [start_node]

    visited_keys = set()
    visited_keys.add(start_node.key)

    paths = [[start_node]]
    path_index = 0

    while path_index < len(paths):

        path = paths[path_index]
        last_node = path[-1]

        for adjacent_node in graph.get_adjacent(last_node):

            # Already visited
            if adjacent_node.key in visited_keys:
                continue

            # Path found
            if adjacent_node == end_node:
                return path + [adjacent_node]

            visited_keys.add(last_node.key)
            paths.append(path + [adjacent_node])

        path_index += 1

    return []

```

La implementación usa BFS, pero además guarda cada uno de los caminos. Una vez que se encuentra el nodo final se retorna el camino que llegó a ese nodo. Podemos asegurar que es el más corto por la propiedad de que BFS siempre toma el camino más corto para cada par de vértices

Ejercicio 11 (Opcional)

Crear una función que determine si un grafo es bipartito

```python
def is_bipartite(graph):

    # A graph G=(V, E) is bipartite when for each edge
    # (u, v) of E, u and v belong to different sets

    def _visit_bipartite(graph, node, group_0, group_1, parent_group_0):

        # 1) When the node is already added
        if node.key in group_0:

            # If the node is in the same group as the parent
            if parent_group_0:
                return False

            return True

        if node.key in group_1:

            # If the node is in the same group as the parent
            if not parent_group_0:
                return False

            return True

        # 2) Adds the node to it's corresponding list (different to parent)
        if parent_group_0:
            group_1.add(node.key)

        else:
            group_0.add(node.key)

        for adjacent_node in graph.get_adjacent(node):
            bipartite = _visit_bipartite(graph, adjacent_node, group_0, group_1, not parent_group_0)
            if not bipartite:
                return False

        return True


    group_0 = set()
    group_1 = set()

    for node in graph.nodes:

        if node.key in group_0 or node.key in group_1:
            continue
        bipartite = _visit_bipartite(graph, node, group_0, group_1, False)
        if not bipartite:
            return False

    return True
```

La idea del algoritmo anterior consiste en usar DFS, y en lugar de tener una única lista de nodos visitados tenemos 2. Además es necesario saber a qué grupo pertenece el nodo "parent" del actual, para poder agregar al nodo actual en otro grupo. De esta manera, cuando el nodo que visitamos ya fué agregado y además pertenece al mismo grupo que el padre podemos decir que no es bipartito (esto último se puede ver en los condicionales de _visit_bipartite)

Ejercicio 12

Demostrar que si el grafo G es un árbol y se le agrega una arista nueva entre cualquier par de vértices se forma exactamente un ciclo y deja de ser un árbol.

1)
$\forall\ grafo\ G\ que\ sea\ un\ \text{á}rbol.$
$Sean\ u,\ v \in V\ dos\ v\text{é}rtices\ cualquiera\ de\ G$

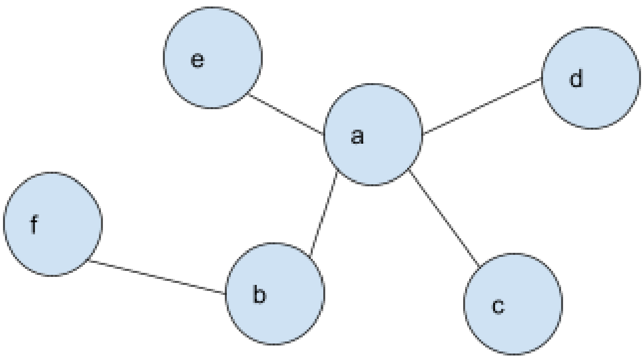
$Si\ agregamos\ la\ arista\ (u,\ v)\ al\ \text{á}rbol,\ entonces\ deja\ de\ ser\ un\ \text{á}rbol$

<u>Demostración por contradicción</u>: Supongo que la conclusión es falsa, por lo tanto

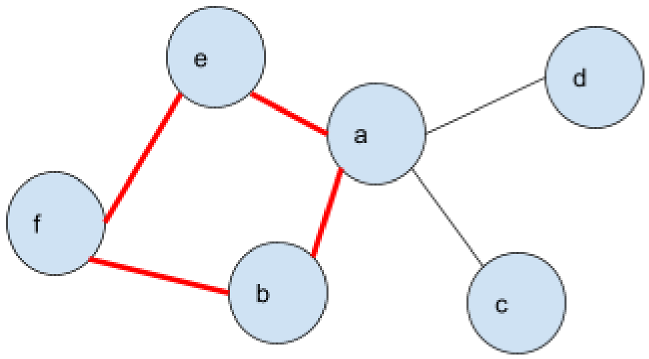
2)
$\forall\ grafo\ G\ que\ sea\ un\ \text{á}rbol.$
$Sean\ u,\ v \in V\ dos\ v\text{é}rtices\ cualquiera\ de\ G$

$Si\ agregamos\ la\ arista\ (u,\ v)\ al\ \text{á}rbol,\ entonces\ el\ grafo\ sigue\ siendo\ un\ \text{á}rbol$

Sea $G = \{V = \{a,\ b,\ c,\ d,\ e,\ f\}, A = \{(a,\ b),\ (a,\ c),\ (a,\ d),\ (a,\ e),\ (b,\ f)\}\}$ el siguiente grafo. Se observa que es un árbol, pues no tiene ciclos



Agregando una arista para un par de vértices $(e,\ f)$



Vemos que se forma un ciclo, razón por la cuál G deja de ser un árbol. Esto nos muestra que el argumento 2) es falso.
Por lo tanto, queda demostrado que $1)$ es verdadero

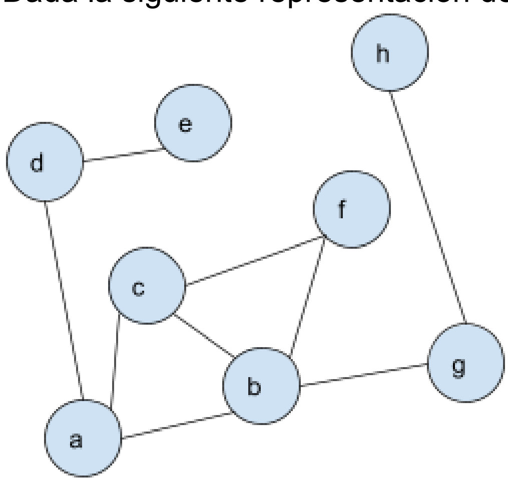
Ejercicio 13

Demostrar que si la arista (u, v) no pertenece al árbol BFS, entonces los niveles de u y v difieren a lo sumo en 1.

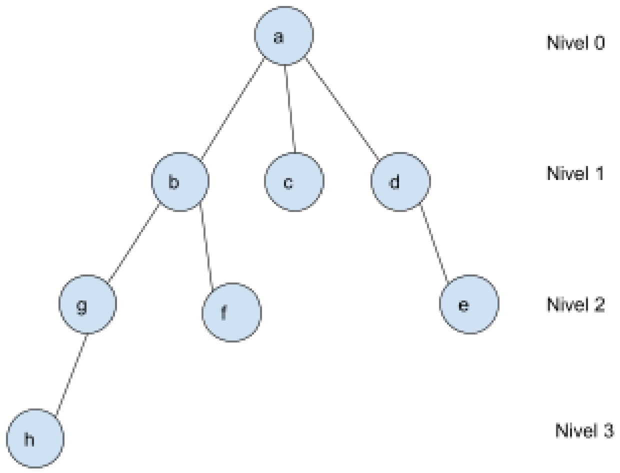
$Para\ todo\ \text{árbol}\ BFS\ G = (V, A),\ si\ se\ toman\ dos\ \text{vértices}\ cualesquiera\ v,\ u \in V\ tales\ que$
$(u,\ v)\ \notin A.$
$Los\ niveles\ de\ u\ y\ v\ difieren\ como\ \text{máximo}\ en\ 1$

Creo que el enunciado es falso, por lo tanto busco un contraejemplo

Dada la siguiente representación de un grafo G



Su árbol de expansión generado por BFS considerando como prioridad al órden alfabético y tomando como raíz al vértice a resulta:



Se observa que $(h,\ a)\ \notin\ A$ y difieren en 3 niveles, por lo tanto queda demostrado por contraejemplo que el enunciado es **falso**

Ejercicio 13''

Interpreto "a lo sumo" como "por lo menos"

$Para\ todo\ \text{árbol}\ BFS\ G\ =\ (V, A),\ si\ se\ toman\ dos\ v\text{é}rtices\ cualesquiera\ v,\ u\ \in\ V\ tales\ que$
$(u,\ v)\ \ \notin\ A.$
$Los\ niveles\ de\ u\ y\ v\ difieren\ por\ lo\ menos\ en\ 1$

Procedo a demostrar por por contradicción, es decir

2) $Los\ niveles\ de\ u\ y\ v\ difieren\ como\ m\text{á}ximo\ en\ 1$

Gracias al punto anterior, sabemos que los niveles de $u,\ v$, pueden diferir en más de un nivel, por lo tanto 2) enunciado es falso, lo que hace que el enunciado principal sea **verdadero**

Ejercicio 14

Implementar el algoritmo de Prim para MST

```python
def prim(graph: dict):

    heap = Heap(len(graph))
    smaller_weights = {}

    # Initializes the heap and weights as maximum
    for node in graph.keys():
        min_weight = 1e7
        smaller_weights[node] = min_weight
        heap.add(node, min_weight)

    # Updates the first key, so it gets extracted first
    first_key = list(graph.keys())[0]
    heap.update_key(first_key, 0)

    # Holds the parent-child results
    parent = {}

    while len(heap):

        # Gets the edge with the smallest weight
        min_key, min_weight = heap.pop()

        for (adyacent_node, weight) in graph[min_key]:

            # If it doesn't make a cycle and the weight isn't the smallest
            if adyacent_node in heap and weight < heap.access(adyacent_node):
                parent[adyacent_node] = (min_key, weight)

                # Adds to the tree
                heap.update_key(adyacent_node, weight)

    # Builds the graph, using the parent dict
    result_tree = {}
    total_weight = 0
    for node in parent.keys():
        child, weight = parent[node]
        weight_insert_double(result_tree, node, child, weight)

        total_weight += weight

    print(f"Weight: {total_weight}")

    return result_tree
```

La implementación anterior se basa en el uso de Min Heap.

1. Lo primero que hago es agregar todas las keys al Heap, donde el peso mínimo de cada una es inicializado con un número muy grande (supongamos que es infinito)
2. Luego, selecciono la primera key del diccionario, y le pongo como peso mínimo 0, de tal manera que en el siguiente bucle sea “popeado” primero
3. Parent es un diccionario, que dado un vértice se almacena su parent dentro del árbol mínimo de expansión
4. Si hay elementos restantes en heap, hay vértices que todavía no han sido agregados al árbol
5. Obtengo el vértice u, junto a su peso, el cuál resulta el menor de todo el grafo
6. Itero por todos los vértices adyacentes v. Si todavía no lo agregamos y el peso de la arista (u, v) es menor que el actual, entonces actualizo el valor en heap y vínculo con parent. Nótese que todavía no retiramos al vértice del heap, por lo cuál puede llegar a existir una arista de v donde su peso sea menor y su parent no termine siendo u
7. Finalmente, construyo el MST teniendo como datos al diccionario parent

Complejidad:

Se observa que el contenido del bucle for interior, el cuál itera por todos los vértices se va a ejecutar $O(|V| + |A|)$, pues con en while vamos a iterar por todos los vértices de V, y con el bucle for, vamos a terminar realizando $2|A|$ iteraciones.

Pero dentro del bucle, usamos a Min Heap. En el peor de los casos, siempre accedemos dentro del condicional if, haciendo que ejecutemos la operación update_key, la cuál tiene una complejidad temporal $O(log(|V|))$.

Por lo tanto, la complejidad de mi algoritmo resulta $O((V + A)log(V))$

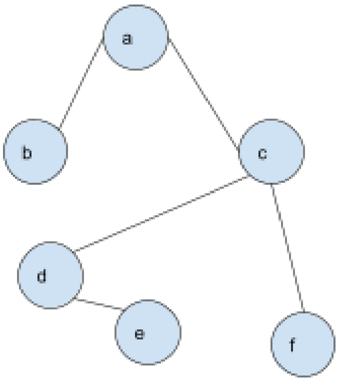
Ejercicio 15

Implementar el algoritmo de Kruskal para MST

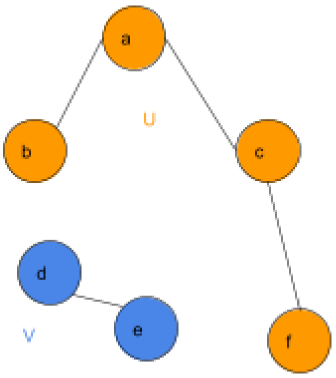
Ejercicio 16

Demostrar que si la arista (u,v) de costo mínimo tiene un nodo en U y otro en V - U, entonces la arista (u,v) pertenece a un árbol abarcador de costo mínimo.

Supongamos que tenemos un MST T del grafo $G = (V, A)$ y se ve de la siguiente manera



Si nosotros retiramos una arista cualquiera de T, vamos a obtener un bosque, formado por T_1 y T_2



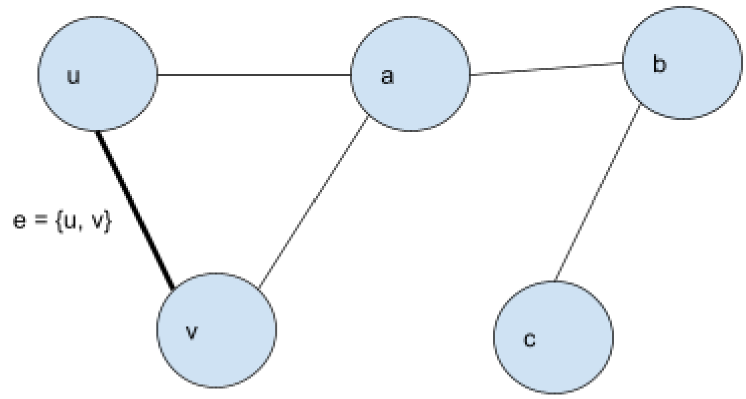
Ahora, podemos observar lo que plantea el ejercicio, dos grupos de vértices U y V, donde tenemos una arista de costo mínimo (c, d). Se observa que si agregamos la arista, es imposible que se formen ciclos, pues $c \in U$ y $d \in V$. Lo cuál demuestra que el enunciado es verdadero

Ejercicio 17

Sea e la arista de mayor costo de algún ciclo de G(V,A) . Demostrar que existe un árbol abarcador de costo mínimo AACM(V,A-e) que también lo es de G.

Lo anterior dice que existe un AACM de G que no incluye a e

Es importante observar que la arista e forma un ciclo, gracias a esto podemos asegurar que los vértices u y v pueden ser agregados al AACM mediante otra arista que no sea e



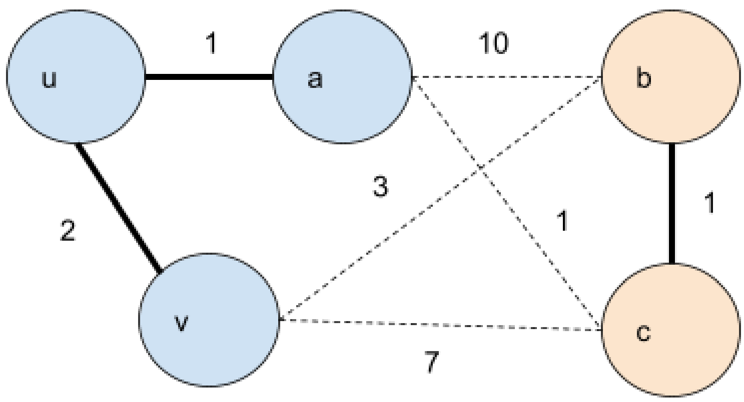
Sabemos que $w(\{u,\ v\})\ =\ max$, es la arista de mayor peso. e no necesariamente es agregada al ACMM, pues si es la única con ese peso, entonces el peso del AACM no sería mínimo. Con lo cual, podemos finalizar la demostración

Nótese que si existe otra arista con ese peso y también forma parte del ciclo, es posible que e se agregue al ACMM, por lo tanto

Ejercicio 18

Demostrar que si unimos dos AACM por un arco (arista) de costo mínimo el resultado es un nuevo AACM. (Base del funcionamiento del algoritmo de Kruskal)

Supongo que el enunciado es falso, es decir que tras unir por el arco de costo mínimo no obtenemos un nuevo AACM. (2)



En el ejemplo anterior, tenemos dos árboles de costo mínimo, por el lado izquierdo el azúl, con un peso total de 3, y por el derecho el naranja, con un peso total de 1.

Si agregamos cualquiera de las aristas rayadas, obtenemos un árbol, pues el grafo resulta conexo y sin ciclos. Para poder satisfacer a (2) agregamos una de las posibles aristas de tal manera que el peso no sea el mínimo, es decir, debemos agregar alguna de las siguientes $\{a,\ b\}$, $\{v,\ b\}$, $\{v,\ c\}$. Pero estas aristas no son de costo mínimo, nos encontramos frente a una contradicción a una de las premisas, por lo tanto (2) es falso.
Lo cuál demuestra automáticamente el enunciado

Ejercicio 19

Explique qué modificaciones habría que hacer en el algoritmo de Prim sobre el grafo no dirigido y conexo G(V,A), o sobre la función de costo c(v1,v2)-> R para lograr:

1. Obtener un árbol de recubrimiento de costo máximo
2. Obtener un árbol de recubrimiento cualquiera
3. Dado un conjunto de aristas E ∈ A, que no forman un ciclo, encontrar el árbol de recubrimiento mínimo $G^c(V,\ A^c)\ tal\ que\ E\ \in\ A^c$

1. En el algoritmo de Prim siempre tenemos que retirar aristas que sean adyacentes a los nodos que ya hemos agregado al árbol y que no formen ciclos. Para lograr lo que pide el enunciado, simplemente tenemos que retirarlas de otra manera, es decir, usar una estructura que siempre retorne la arista de mayor peso que cumpla con las condiciones anteriores. Para esto podríamos usar la estructura max-heap, en lugar de min-heap
2. En este caso, cada vez que retiramos las aristas, no hay que verificar que sea la de mayor o menor peso, sino que la retiramos sin importar su peso y la agregamos al árbol verificando que no forme ciclos. De esta manera no podemos asegurar que obtengamos un AACMin o AACMax, por lo que decimos que es un árbol abarcador cualquiera
3. No entiendo la consigna. Supongo que $E \subset A$, donde las aristas de E no forman ciclos. En ese caso, si queremos encontrar el AACM de $G = (V,\ E)$, entonces no es necesario verificar la formación de ciclos, sinó agregar las aristas adyacentes de menor peso sucesivamente, hasta que se hayan agregado todos los vértices

Ejercicio 20

Sea G un grafo conexo, no dirigido y ponderado, donde todas las aristas tienen el mismo costo. Suponiendo que G está implementado usando matriz de adyacencia, haga en pseudocódigo un algoritmo $O(V^2)$ que devuelva una matriz M de VxV donde: M[u, v] = 1 si (u,v) ∈ A y (u, v) estará obligatoriamente en todo árbol abarcador de costo mínimo de G, y cero en caso contrario.

Breve descripción

Las aristas obligatorias son aquellas que son aristas *"puente"*, es decir que debemos encontrar aquellas aristas tal que retiradas aumenta la cantidad de componentes conexas. Para hacer esto utilizo el algoritmo de Tarjan , el cual utiliza D.F.S. para poder recorrer el grafo.

Vamos a tener un par de estructuras adicionales, tales como el tiempo en el que descubrimos al vértice por primera vez **discovery_time**, el tiempo del nodo adyacente más chico **low_adjacent_time** y otra lista que almacena al antecesor inmediato **parent**

La idea fundamental, es encontrar una arista (u, v) tal que el subárbol con raíz v no tiene ninguna otra arista conectada con antecesores de u (u incluído). Si esto se cumple, entonces tenemos que discovery_time[u] $<$ low_adjacent_time[v].

Implementación

Código para crear el grafo usando matriz de adyacencia

```python
def initialize_adj_matrix(node_count, init_val=0):
    return [[init_val for _ in range(node_count)] for _ in range(node_count)]


def add_edge_matrix(graph: list, e1, e2):
    graph[e1][e2] = 1
    graph[e2][e1] = 1
```

Función principal, en la que se observa la inicialización de las estructuras mencionadas previamente

```python
def get_bridges(graph: dict):

    """
    Uses Tarjan's theorem, therefore the time complexity is O(V + E)
    """

    node_count = len(graph)
    visited = [False] * node_count
    discovery_time = [None] * node_count
    parent = [None] * node_count
    low_adjacent_time = [None] * node_count
    bridges = []
    time = 0

    for node in range(node_count):
        if not visited[node]:
            _dfs_bridge(graph, node, visited, discovery_time, low_adjacent_time, parent, time, bridges)

    return bridges
```

Función recursiva de D.F.S. En la línea 25 se observa la condición mencionada

```python
1  def _dfs_bridge(graph, node, visited, discovery_time, low_adjacent_time, parent, time: int, bridges):
2
3      visited[node] = True
4      discovery_time[node] = time
5      low_adjacent_time[node] = time
6      time += 1
7
8      # Iterates through all the row
9      for adyacent, is_adyacent in enumerate(graph[node]):
10
11         # Only care about adyacent nodes
12         if not is_adyacent:
13             continue
14
15         if adyacent == parent[node]:
16             continue
17
18         if not visited[adyacent]:
19             parent[adyacent] = node
20             _dfs_bridge(graph, adyacent, visited, discovery_time, low_adjacent_time, parent, time, bridges)
21
22             # Propagates the lower adyacent time
23             low_adjacent_time[node] = min(low_adjacent_time[node], low_adjacent_time[adyacent])
24
25             if low_adjacent_time[adyacent] > discovery_time[node]:
26                 # The tree rooted with adyacent has no regression edge to
27                 # node parents
28                 bridges.append((node, adyacent))
29         else:
30             low_adjacent_time[node] = min(low_adjacent_time[node], discovery_time[adyacent])
```

Ejercicio 21

```python
class Node:
    def __init__(self, key) -> None:
        self.key = key
        self.d = float("inf")
        self.parent = None
```

Etapa de inicialización

```python

def dijkstra(graph, start_key, end_key):

    weights = {}        # Stores the weight of each (u, v) edge
    nodes = {}          # Stores the Node instances

    edges_count = 0

    # Creates the node instances, and fills the weights dict
    for node_key in graph.keys():

        # The node could be added here
        nodes[node_key] = Node(node_key)

        for ady, weight in graph[node_key]:
            edges_count += 1
            weights[(node_key, ady)] = weight

    # Creates the heap and initializes d attrubute as +infinite
    heap = Heap(edges_count)
    for node_key in graph.keys():
        heap.add(node_key, float("inf"))

    visited_keys = set()

    # Sets the d value of the starting node to 0
    # this way the heap pops the start first
    heap.update_key(start_key, 0)
    nodes[start_key].d = 0
```

Etapa de ejecución

```
while len(heap):

    (node_key, d) = heap.pop()
    visited_keys.add(node_key)

    # Gets the node, containing d, and parent data
    node = nodes[node_key]

    for ady, weight in graph[node_key]:

        if ady not in visited_keys:

            ady_node = nodes[ady]

            # Relax the adyacent
            modified = relax(node, ady_node, weights)

            # Only if the relax call modified the value
            if modified:
                heap.update_key(ady, ady_node.d)

end_node = nodes[end_key]

# No path, unreachable from start_key
if end_node.parent is None:
    return None

# Traverses the path backwards, using the parent attribute
path = []
parent = end_node
while parent is not None:
    path.append(parent.key)
    parent = parent.parent

return path
```

Funcionamiento

- Etapa de inicialización
  - weights es un diccionario que contiene el peso de toda arista $(u,\ v)$
  - nodes es un diccionario que dada la key de un vértice, retorna su nodo correspondiente. Nótese que los nodos guardan la información de d y de $parent$
  - heap es una $min\ heap$, usada como $cola$ para ir retirando los nodos con menor d
  - Se observa que actualizo el valor $d\ =\ 0$ del nodo de entrada, o comienzo, para que sea el primero en ser retirado de heap

- Etapa de ejecución
  - Se retira el nodo con menor d
  - Se lo agrega al set de visitados
  - Para todo adyacente que no haya sido visitado:
    - Se lo *"relaja"*
    - Si el valor d del nodo adyacente fue modificado, entonces se actualiza en *heap*

- Etapa de finalización:
  - Si el nodo *"final"* no tiene parent, entonces "start_key" y "end_key" se encuentran en dos componentes conexas disjuntas del grafo -> None
  - Caso contrario, existe un camino. Se recorre el camino de parents, el cuál resulta ser el mejor camino y se lo retorna

Complejidad

- En la etapa de inicialización, tenemos una complejidad $O(V)$, pues iteramos por los vértices/nodos dos veces.
- En la etapa de ejecución, mediante la estructura heap, vamos a iterar por todos los vértices del grafo, y para cada uno de estos vamos a iterar por todos sus adyacentes. Es decir $O(V\ +\ A)$, pero observamos que en el peor caso, siempre vamos a tener que actualizar el valor d de los nodos, una operación de orden $O(log(V))$. Razón por la cuál, la complejidad temporal en esta etapa es de $O((V\ +\ A)log(V))$
- En la etapa de finalización, en el peor de los casos, vamos a tener que recorrer $|V|$ nodos buscando el camino más corto, lo cuál hace $O(V)$

Claramente la complejidad resulta determinada por la etapa de ejecución, resultando

$$O((V\ +\ A)log(V))$$