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
```

El problema se resuelve creando una lista con los arcos de retroceso

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)

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

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

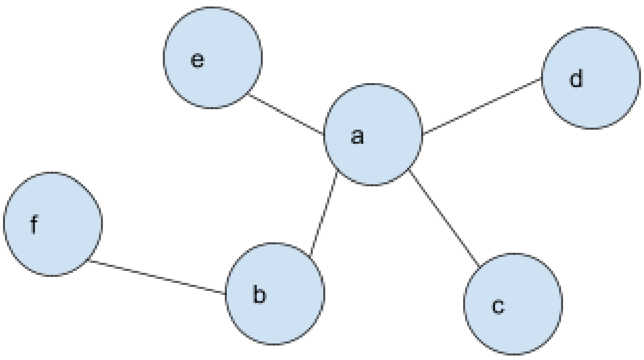
<u>Demostración por contradicción</u>: Supongo que la conclusión es falsa, por lo tanto

2)

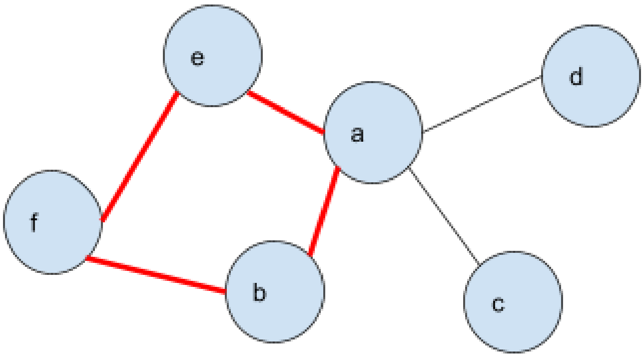
$\forall\ grafo\ G\ que\ sea\ un\ \text{á}rbol.$
$Sean\ u,\ v \in V\ dos\ v\text{é}rtices\ cualquiera\ de\ G$

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

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

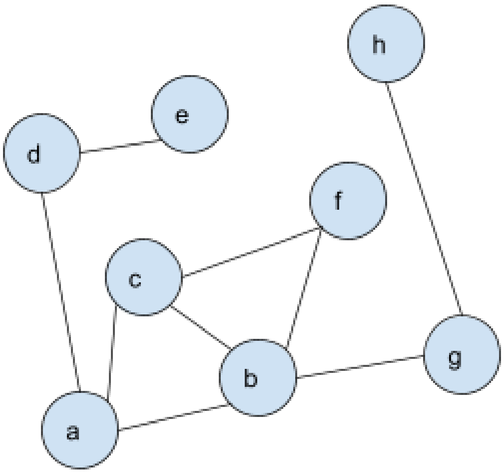
Ejercicio 13

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.

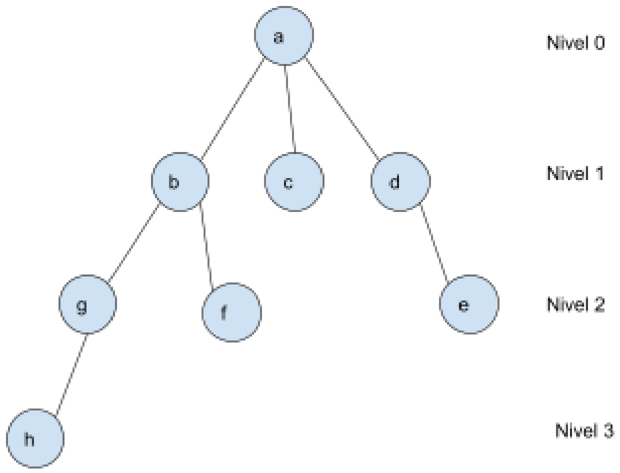
$Para\ todo\ árbol\ BFS\ G = (V, A),\ si\ se\ toman\ dos\ vértices\ cualesquiera\ v,\ u \in V\ tales\ que$
$(u,\ v)\ \notin A.$
$Los\ niveles\ de\ u\ y\ v\ difieren\ como\ máximo\ en\ 1$

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"

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

Procedo a demostrar por por contradicción, es decir

2) $Los\ niveles\ de\ u\ y\ v\ difieren\ como\ máximo\ en\ 1$

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**