```python
def dfs(G, origen, destino, depth=0):

    path = []
    expanded = 1

    visited = {}
    previous = {}

    for node in G.nodes():
        visited[node] = False
        previous[node] = None

    stack = [origen]
    trobat = False



    while ((len(stack) != 0) and (not trobat)):

        nodoActual = stack.pop()
        expanded += 1

        vecinos = [nodoVecino
                   for nodoVecino in G.neighbors(nodoActual)
                   if not visited[nodoVecino]]

        for nodoVecino in vecinos:

            visited[nodoVecino] = True
            stack.append (nodoVecino)
            previous[nodoVecino] = nodoActual

            if (nodoVecino == destino):
                trobat = True

        visited[nodoActual] = True



    path = [destino]

    while previous[path[0]] != origen:
        path = [previous[path[0]]] + path

    path = [origen] + path

    return {
        'path' : path,
        'expanded' : expanded
    }
def bfs(G, origen, destino):


    path = []
    expanded = 0

    queue = [origen]

    for nodo in G.nodes():
        G.node[nodo].update({'previous': None,
                             'distance': float('inf')})

    G.node[origen]['distance'] = 0

    nodoActual = origen



    while (len(queue) != 0):

        nodoActual = queue[0]
        queue.remove(queue[0])
        expanded += 1

        for nodoVecino in G.neighbors(nodoActual):

            if (G.node[nodoVecino]['distance'] == float('inf')):
                G.node[nodoVecino]['distance'] = G.node[nodoActual]['distance'] + 1
                G.node[nodoVecino]['previous'] = nodoActual

                queue.append(nodoVecino)



    nodoActual = destino

    while (nodoActual != origen):
        path.append(nodoActual)
        nodoActual = G.node[nodoActual]['previous']

    path.append(origen)
    path.reverse()

    return {'path': path, 'expanded': expanded}
```

```python
# DFS recursivo que te devuelve todos los subgrafos de un grafo

def dfs(G):

    expanded = 0

    visited = []
    pending = []
    subgraphs = []

    for nodo in G.nodes():
        if (nodo not in visited):

            pending.append(nodo)
            subgraph = []

            exploreDFS(G, visited, pending, subgraph)
            expanded += len(subgraph)

            subgraphs.append(subgraph)

    return {
        "subgraphs": subgraphs,
        "expanded": expanded
    }

def exploreDFS(G,visited,pending,subgraph):

    if (len(pending) != 0):
        nodo = pending.pop()

        if nodo not in visited:
            visited.append(nodo)
            subgraph.append(nodo)

            for nodoVecino in G.neighbors(nodo):
                if nodoVecino not in visited:
                    pending.append(nodoVecino)

        exploreDFS(G,visited,pending,subgraph)
```

```python
def kruskal(lst):

    # El primer pas és ordenar tots els nodes en funció d
    lst = sorted(lst, key=lambda x: x[2])

    # Agafem tots els nodes únics. Ara la llista té tres
    list1, list2, _ = zip(*lst)
    unique_nodes = set(list1+list2)

    # Inicialitzem les dues variables rank i parent que e
    rank = defaultdict(int)
    parent = {n: n for n in unique_nodes}

    # Aquí guardem les arestes que formen part del MST
    tree = []

    # Aquí guardarem totes les versions de la variable 't
    tree_states = []

    # Recorrem totes les parelles del graf.
    # ...És necessari? Podeu pensar una solució millor
    for node1, node2, weight in lst:

        # Busquem el node arrel de cada un dels nodes
        parent1 = find(parent, node1)
        parent2 = find(parent, node2)

        # Si tenen el mateix pare, no fem res i continuem
        if parent1 == parent2:
            continue

        # OBSERVACIÓ: Aquesta aresta forma part del MST
        tree.append((node1, node2, weight))

        union(parent, rank, parent1, parent2)

        # Guardem totes les versions del diccionari parent
        tree_states.append(tree.copy())

    return tree, tree_states
```

```python
FS recursivo que te devuelve el path minimo de un nodo origen a un nodo destino

Made by JUDIT ;) """

dfs_rec(G, origen, destino):
expanded = 0
visited = {}
prev = {}

for node in G.nodes():
    visited[node] = False

visited, prev, expanded = exploreDFS_rec(G, origen, destino, visited, prev, expanded)

current = destino
path = []
while current != origen:
    path.append(current)
    current = prev[current]
path.append(origen)
path.reverse()

return {
    'path' : path,
    'expanded' : expanded
}

exploreDFS_rec(G, current, destino, visited, prev, expanded):

if current == destino:
    return visited, prev, expanded

expanded += 1
visited[current] = True

for neighbor in nx.neighbors(G, current):
    if visited[neighbor] == False:
        prev[neighbor] = current
        visited, prev, expanded = exploreDFS_rec(G, neighbor, destino, visited, prev, e

return visited, prev, expanded
```

```python
def isCyclic(G, u, visited, prev, path):
    # Pone en visitado el actual
    visited[u]= True
    # Recorre los vecinos
    for v in G.neighbors(u):
        # Si el vecino no está visitado le visitamos
        if not visited[v]:
            if isCyclic(G,v,visited,u,path):
                path.append(v)
                return True
        # Si un vecino está visitado y su padre no, hay un ciclo
        elif  prev != v:
            path.append(v)
            return True
    path = []
    return False

def detect_cycle(G):
    path = []
    visited = {}
    for u in G.node:
        visited[u] = False
    for u in G.nodes():
        if not visited[u]: #Don't recur for u if it is already visited
            if isCyclic(G,u,visited,-1,path):
                path.append(path[0])
                path.reverse()
                return True, path
    return False, path
```

```python
# BFS que te encuentra los subgrafos de un grafo que le pasas por parámetro

def bfs(G):

    visited = []
    pending = []
    subgraphs = []

    expanded = 0

    for nodeActual in G.nodes():

        if (nodeActual not in visited):

            subgraph = []
            pending.append(nodeActual)

            while (len(pending) != 0):
                nodo = pending.pop(0)

                if (nodo not in visited):
                    expanded += 1
                    visited.append(nodo)
                    subgraph.append(nodo)

                    for nodoVecino in G.neighbors(nodo):
                        if ((nodoVecino not in visited) and (nodoVecino not in pending)

                            pending.append(nodoVecino)

            subgraphs.append(subgraph)

    return {
        'subgraphs' : subgraphs,
        'expanded' : expanded
    }
```

```python
# En Python, podeu representar l'infinit com a float('inf')
import heapq
from functools import lru_cache
from collections import defaultdict

@lru_cache
def dijkstra(G, origin, destination):
    """
    Params
    ======
    :G: Graf del qual en volem extreure el camí mínim. Ha de ser un objecte de la class
    :origin: Índex del node origen
    :destination: Índex del node destí

    Returns
    =======
    Un diccionari amb tres elements:

    :path: Una llista de nodes del camí més curt entre els nodes 'origin' i 'destinatio
    :expanded: El nombre de nodes que s'han visitat per trobar la solució
    :distance: La distancia del camí mínim entre 'origin' i 'destination'
    """
    #Fem ús d'un conjunt per anar afegint els nodes ja visitats
    visited = set()

    #Llista de nodes del camí més curt entre els nodes 'origin' i 'destination'
    path = []

    #Nombre de nodes que s'han visitat per trobar la solució
    expanded = 0;

    #Diccionari que conté per cada node visitat els seus predecessors
    predecessors = {}

    #heapq que emmagatzema distancia i node
    pq = []

    #nodeCosts guarda el cost dels nodes des del node origen, primer els inicialitzem t
    nodeCosts = defaultdict(lambda: float('inf'))

    #iInicialitzem el cost del node origen a 0
    nodeCosts[origin] = 0;

    #Afegim al heap la distància de l'origen a ell mateix, que és 0
    heapq.heappush(pq, (0, origin))

    while pq and (destination not in visited):
        dist, n = heapq.heappop(pq)

        if n not in visited:
            expanded += 1
            visited.add(n)

            for neighbor in G.neighbors(n):
                route = nodeCosts[n] + 1
                if(route < nodeCosts[neighbor]):
                    nodeCosts[neighbor] = route
                    predecessors[neighbor] = n
                    heapq.heappush(pq,(nodeCosts[neighbor], neighbor))

    #Anirem desde el node destí cap enrere per trobar el camí més curt entre aquest i l
    #Iniciem a l'últim node (destination)
    v = destination

    #Mitjançant el diccionari creat prèviament (predecessors) fem un bucle fins arribar
    while (v != origin):
        path.append(v) #Afegim el node actual al path
        v = predecessors[v]

    #Afegim a path el node origin
    path.append(origin)

    #Capgirem la llista
    path.reverse()

    return {
        'path': path,
        'expanded': expanded,
        'distance': nodeCosts[destination]
    }
```

```python
def checkpoint(G, origin, destination, extra):
    """
    Params
    ======
    :G: Graf del qual en volem extreure el camí mínim. Ha de ser un objecte de la class
    :origin: Índex del node origen
    :destination: Índex del node destí
    :extra: Índex d'un node extra per on ha de passar el camí

    Returns
    =======
    Un diccionari amb dos elements:

    :path: Una llista de nodes del camí més curt entre els nodes 'origin' i 'destinatio
    :distance: La distància del camí.
    """
    #En aquest exercici com només és té un node extra (intermig entre origin i destiona
    #de la funció dijkstra desdel node origin fins al node extra, i posteriorment del n

    path1 = dijkstra(G, origin, extra)
    path2 = dijkstra(G, extra, destination)

    #De cadascuna de les variables 'path1' i 'path2' només ens interessa pel path el se
    #En la variable 'path1' agafem tot el path fins el penúltim node, per tal de no rep

    path = path1['path'][:-1] + path2['path']

    #Ara volem extreure el return de la key 'distance' de les dues variables per tal d'
    distance = path1['distance'] + path2['distance']
    return {'path': path,
            'distance': distance}
```

```python
from itertools import permutations
def checkpoints_list(G, origin, destination, extras):
    """
    Params
    ======
    :G: Graf del qual en volem extreure el camí mínim. Ha de ser un objecte de la class
    :origin: Índex del node origen
    :destination: Índex del node destí
    :extras: Llista d'índexs de nodes per on ha de passar el camí.

    Returns
    =======
    Un diccionari amb dos elements:

    :path: Una llista de nodes del camí més curt entre els nodes 'origin' i 'destinatio
    :distance: La distància del camí.
    """

    #Llista de nodes del camí més curt entre els nodes 'origin' i 'destination'
    path = []

    #Distància del camí més curt
    distance = float('inf')

    #Diccionari on guardarem els tres retorns de la funció dijsktra
    values = {}

    #Creem una llista amb el node origin, els nodes extras i el destination, per poster
    #fer les diferents permutacions
    lista = extras
    lista.append(origin)
    lista.append(destination)

    #Fem les permutacions
    p1 = permutations(lista)

    #Creem una segona llista per tal de guardar només les permutacions on el node origi
    #de la llista i el node destination al final de la llista
    p2 = []

    #Ens quedem nomes amb les combinacions en que el node origin sigui al principi de l
    #destination al final de la llista
    for permutation in p1:
        if(permutation[0] == origin and permutation[-1] == destination):
            p2.append(permutation)

    #Per cada permutació de la llista, iterem
    for permutation in p2:
        distanceAux = 0
        pathAux = []
        m = 0
        n = 1

        #Les permutacions són de l'estil (node1, node2, node3, node4, node5, node6), vo
        #dijsktra per combinacions entre ells, per exemple: node1 & node2, i la següent
        #seria node2 & node3
        while n <= len(permutation)-1:
            values = dijkstra(G, permutation[m], permutation[n])
            distanceAux += values['distance']
            pathAux += values['path']
            m += 1
            n += 1

        #Si la distància trobada és més petita que la guardada, la canviem
        if distanceAux < distance:
            distance = distanceAux
            path = pathAux

    return {'path': path,
            'distance': distance}
```

```python
from collections import defaultdict
def compute_frequency(text):
    """
    Params
    ======
    :text: El text que volem codificar

    Returns
    =======
    :dct: Un diccionari amb el nombre d
    """
    dct = defaultdict(int)

    for item in text:
        dct[item] += 1

    return dict(dct)
```

```python
def encode(text, diccionari):
    """
    Donat un text a codificar i un

    Params
    ======
    :text: El text que volem codif
    :diccionari: El diccionari de

    Returns
    =======
    :code: Una representació del t

    """
    code = ''

    for lletra in text:
        code += diccionari[lletra]

    return code
```

```python
def holes(G, origin, destination, holes_list=[], penalty=50):
    """
    Params
    ======
    :G: Graf del qual en volem extreure el camí mínim. Ha de ser un objecte de la class
    :origin: Índex del node origen
    :destination: Índex del node destí
    :holes_list: Una llista de punts que tindran penalització
    :penalty: Valor enter de penalització

    Returns
    =======
    Un diccionari amb dos elements:

    :path: Una llista de nodes del camí més curt entre els nodes 'origin' i 'destinatio
    :distance: La distància del camí.
    """

    #Fem ús d'un conjunt per anar afegint els nodes ja visitats
    visited = set()

    #Llista de nodes del camí més curt entre els nodes 'origin' i 'destination'
    path = []

    #Diccionari que conté per cada node visitat els seus predecessors
    predecessors = {}

    #heapq que emmagatzema distancia i node
    pq = []

    #nodeCosts guarda el cost dels nodes des del node origen, primer e
    nodeCosts = defaultdict(lambda: float('inf'))

    #iInicialitzem el cost del node origen a 0
    nodeCosts[origin] = 0;

    #Afegim al heap la distància de l'origen a ell mateix, que és 0
    heapq.heappush(pq, (0, origin))

    #És el mateix algorisme que dijsktra però amb la diferència de que
    #es troba a 'holes_list', si és així, s'aplica una penalització
    while pq and (destination not in visited):
        dist, n = heapq.heappop(pq)

        if n not in visited:
            visited.add(n)

            for neighbor in G.neighbors(n):
                if (neighbor in holes_list):
                    route = nodeCosts[n] + penalty + 1
                else:
                    route = nodeCosts[n] + 1

                if(route < nodeCosts[neighbor]):
                    nodeCosts[neighbor] = route
                    predecessors[neighbor] = n
                    heapq.heappush(pq,(nodeCosts[neighbor], neighbor))

    #Anirem desde el node destí cap enrere per trobar el camí més curt entre aquest i l
    #Iniciem a l'últim node (destination)
    v = destination

    #Mitjançant el diccionari creat prèviament (predecessors) fem un bucle fins arribar
    while (v != origin):
        path.append(v) #Afegim el node actual al path
        v = predecessors[v]

    #Afegim a path el node origin
    path.append(origin)

    #Capgirem la llista
    path.reverse()

    return {
        'path': path,
        'distance': nodeCosts[destination]
    }
```

```python
def decode(text, diccionari):
    """
    Donat un text a decodificar i un diccionari de con

    Params
    ======
    :text: El text que volem decodificar (caràcters '.
    :diccionari: El diccionari de conversió que hem fe

    Returns
    =======
    :code: El text resultant de la decodificació.
    """
    i = 0
    j = 1
    code = ''
    inv_dict = {v: k for k, v in diccionari.items()}
    saltat = False

    for simbol in text:
        if inv_dict.get(text[i:j]) and simbol in '-.':
            code += inv_dict.get(text[i:j])

            if(saltat):
                i = j
                j += 1
                saltat = False
            else:
                i += 1
                j += 1
        else:
            j += 1
            saltat = True
    return code
```

```python
def assign_codes(text, counts):
    """
    Aquesta funció construeix el diccionari de conversió de lletres a símbols '.

    Params
    ======
    :text: El text que volem convertir
    :counts: El diccionari de freqüències que ens retorna la funció compute_freq

    Returns
    =======
    :codes: El diccionari de conversió. Per exemple: {'C': '--', 'B': '-.', 'A':
    """
    # Llista de nodes on cada lletra és un node i té com a valor la freqüència c
    nodes_list = []

    # Omplim la llista amb les dades del diccionari amb les lletres i freqüencie
    for key, value in counts.items():
        nodes_list.append(Node(key, value))

    # Mentre la llista tingui dos nodes com a mínim
    while len(nodes_list) > 1:
        # Ordenem la llista de menor a major freqüència del node
        nodes_list = sorted(nodes_list, key=lambda x: x.value)

        # Extraiem els dos nodes amb la freqüència més petita
        left = nodes_list[0]
        right = nodes_list[1]

        # Assignem el caràcter '.' al node de la dreta i el caràcter '-' al node
        left.set_code('-')
        right.set_code('.')

        # Guardem els nodes left i right assignant-los al pare, que tindrà la su
        newNode = Node(left.node+right.node, left.value+right.value,left,right)

        # Eliminem els dos nodes de la llista i afegim el node pare
        nodes_list.remove(left)
        nodes_list.remove(right)
        nodes_list.append(newNode)

    codes = {}
    addNodes(nodes_list[0], codes)
    return codes

def addNodes(node, dictCodificacio, val = ''):
    newVal = val + node.code

    # Recorregut inorder per recurssió
    if(node.left):
        addNodes(node.left, dictCodificacio, newVal)
    if(node.right):
        addNodes(node.right, dictCodificacio, newVal)
```