In [ ]:
```
matrice_poids = [[0, -1, 1, 1],
                 [-1, 0, 1, -1],
                 [1, 1, 0, 1],
                 [1, -1, 1, 0]]

configs = [(0,0,0,0), [0,0,0,1], [0,0,1,0], [0,0,1,1], [0,0,1,1],
           [0,1,0,0], [0,1,0,1], [0,1,1,0], [0,1,1,1],
           [1,0,0,0], [1,0,0,1], [1,0,1,0], [1,0,1,1],
           [1,1,0,0], [1,1,0,1], [1,1,1,0], [1,1,1,1]]

f_activation = lambda x: 1 if x > 0 else 0

# on part de la config "0" :

# config = [0, 0, 0, 0]

# testons la cellule "a" aka "cellule 0" :

# cellule=0


# on active les cellules dans l'ordre a, b, c, d
def activation_asynchrone(config, poids, ncells=4):
    nouvelle_config = config.copy()
    for c in range(ncells):
        somme = 0
        for j in range(ncells):
            somme += nouvelle_config[j] * poids[j][c]
        nouvelle_config[c] = f_activation(somme)
    return nouvelle_config

#on active les cellules en même temps
def activation_synchrone(config, poids, ncells=4):
    nouvelle_config = [0] * ncells
    for c in range(ncells):
        somme = 0
        for j in range(ncells):
            somme += config[j] * poids[j][c]
        nouvelle_config[c] = f_activation(somme)
    return nouvelle_config

def convert_dec(config, ncells=4):

    return sum([config[i] * 2**(ncells-1-i) for i in range(ncells)])

def isCyclicUtil(v, visited, recStack, dict_configs, verbose):
    # Mark current node as visited and adds to recursion stack
    #print(f"visiting node {v}")
    visited[v] = True
    recStack[v] = True

    # Recur for all neighbours if any neighbour is visited and in
    # recStack then graph is cyclic
    neighbour = dict_configs[v]
    if neighbour != v:
        #print(f"checking neighbour (neighbour) of node {v}")
        if not visited[neighbour]:
            if isCyclicUtil(neighbour, visited, recStack, dict_configs, verbose):
                return True
        elif recStack[neighbour]:
            if verbose:
                print(f"arc retour détecté du noeud {v} au noeud (neighbour)")
            return True

    # The node needs to be popped from recursion stack before function ends
    recStack[v] = False
    #print(f"Node {v} removed from recursion stack")
    return False

# Returns true if graph is cyclic else false
def isCyclic(dict_configs):
    dic_length = len(dict_configs)
    visited = [False] * (dic_length)
    recStack = [False] * (dic_length)
    res, verbose = False, True
    nb_cycles = 0
    for node in range(dic_length):
        if not visited[node]:
            if verbose and isCyclicUtil(node, visited, recStack, dict_configs, verbose):
                res = True
                verbose = False
                nb_cycles += 1
            if not verbose and isCyclicUtil(node, visited, recStack, dict_configs, verbose):
                nb_cycles += 1

    return res, nb_cycles


#affichage
def detect_cycles(dict_configs):
    cyclic, nb_cycles = isCyclic(dict_configs)
    print ("CYCLE DETECTE") if cyclic else print("PAS DE CYCLE")
    return cyclic, nb_cycles


nb_cycles = 0
dict_configs = {}
print("----------------------------")
print("Activation synchrone :")
for conf in configs:
    nouvelle_config = activation_synchrone(conf, matrice_poids)
    print("Transition :", conf, "->", nouvelle_config)
    value_conf, value_new_conf = convert_dec(conf), convert_dec(nouvelle_config)
    print(value_conf, "->", value_new_conf)
    dict_configs[value_conf] = value_new_conf
    if conf == nouvelle_config:
        print("Config stable")

cyclic, nb_cycles = detect_cycles(dict_configs)

nb_cycles = 0
dict_configs = {}
print("----------------------------")
print("----------------------------")
print("Activation asynchrone :")
for conf in configs:
    nouvelle_config = activation_asynchrone(conf, matrice_poids)
    print("Transition de la config", conf, "->", nouvelle_config)
    value_conf, value_new_conf = convert_dec(conf), convert_dec(nouvelle_config)
    print(value_conf, "->", value_new_conf)
    dict_configs[value_conf] = value_new_conf
    if conf == nouvelle_config:
        print("Config stable")

cyclic, nb_cycles = detect_cycles(dict_configs)

print("----------------------------")
print("----------------------------")
```

```
----------------------------
Activation synchrone :
Transition :  [0, 0, 0, 0] -> [0, 0, 0, 0]
0 -> 0
Config stable
Transition :  [0, 0, 0, 1] -> [1, 0, 1, 0]
1 -> 10
Transition :  [0, 0, 1, 0] -> [1, 1, 0, 1]
2 -> 13
Transition :  [0, 0, 1, 1] -> [1, 0, 1, 1]
3 -> 11
Transition :  [0, 1, 0, 0] -> [0, 0, 1, 0]
4 -> 2
Transition :  [0, 1, 0, 1] -> [0, 0, 1, 0]
5 -> 2
Transition :  [0, 1, 1, 0] -> [0, 1, 1, 0]
6 -> 6
Config stable
Transition :  [0, 1, 1, 1] -> [1, 0, 1, 0]
7 -> 10
Transition :  [1, 0, 0, 0] -> [0, 0, 1, 1]
8 -> 3
Transition :  [1, 0, 0, 1] -> [1, 0, 1, 1]
9 -> 11
Transition :  [1, 0, 1, 0] -> [1, 0, 1, 1]
10 -> 11
Transition :  [1, 0, 1, 1] -> [1, 0, 1, 1]
11 -> 11
Config stable
Transition :  [1, 1, 0, 0] -> [0, 0, 1, 0]
12 -> 2
Transition :  [1, 1, 0, 1] -> [0, 0, 1, 0]
13 -> 2
Transition :  [1, 1, 1, 0] -> [0, 1, 1, 1]
14 -> 7
Transition :  [1, 1, 1, 1] -> [1, 0, 1, 1]
15 -> 11
arc retour détecté du noeud 13 au noeud 2
CYCLE DETECTE
----------------------------
----------------------------
Activation synchrone :
Transition de la config [0, 0, 0, 0] -> [0, 0, 0, 0]
0 -> 0
Config stable
Transition de la config [0, 0, 0, 1] -> [1, 0, 1, 1]
1 -> 11
Transition de la config [0, 0, 1, 0] -> [1, 0, 1, 1]
2 -> 11
Transition de la config [0, 0, 1, 1] -> [1, 0, 1, 1]
3 -> 11
Transition de la config [0, 1, 0, 0] -> [0, 0, 0, 0]
4 -> 0
Transition de la config [0, 1, 0, 1] -> [0, 0, 1, 1]
5 -> 3
Transition de la config [0, 1, 1, 0] -> [0, 1, 1, 0]
6 -> 6
Config stable
Transition de la config [0, 1, 1, 1] -> [1, 0, 1, 1]
7 -> 11
Transition de la config [1, 0, 0, 0] -> [0, 0, 1, 0]
8 -> 2
Transition de la config [1, 0, 0, 1] -> [1, 0, 1, 1]
9 -> 11
Transition de la config [1, 0, 1, 0] -> [1, 0, 1, 1]
10 -> 11
Transition de la config [1, 0, 1, 1] -> [1, 0, 1, 1]
11 -> 11
Config stable
Transition de la config [1, 1, 0, 0] -> [0, 0, 0, 0]
12 -> 0
Transition de la config [1, 1, 0, 1] -> [0, 0, 1, 1]
13 -> 3
Transition de la config [1, 1, 1, 0] -> [0, 1, 1, 0]
14 -> 6
Transition de la config [1, 1, 1, 1] -> [1, 0, 1, 1]
15 -> 11
PAS DE CYCLE
----------------------------
----------------------------
```

In [ ]:
```
f_activation = lambda x: 1 if x > 0 else 0
# on active les cellules dans l'ordre a, b, c, d
def activation_asynchrone(config, poids, ncells=4):
    nouvelle_config = config.copy()
    for c in range(ncells):
        somme = 0
        for j in range(ncells):
            somme += nouvelle_config[j] * poids[j][c]
        nouvelle_config[c] = f_activation(somme)
    return nouvelle_config

#on active les cellules en même temps
def activation_synchrone(config, poids, ncells=4):
    nouvelle_config = [0] * ncells
    for c in range(ncells):
        somme = 0
        for j in range(ncells):
            somme += config[j] * poids[j][c]
        nouvelle_config[c] = f_activation(somme)
    return nouvelle_config

def convert_dec(config, ncells=4):

    return sum([config[i] * 2**(ncells-1-i) for i in range(ncells)])

def isCyclicUtil(v, visited, recStack, dict_configs, verbose):
    # Mark current node as visited and adds to recursion stack
    #print(f"visiting node {v}")
    visited[v] = True
    recStack[v] = True

    # Recur for all neighbours if any neighbour is visited and in
    # recStack then graph is cyclic
    neighbour = dict_configs[v]
    if neighbour != v:
        #print(f"checking neighbour (neighbour) of node {v}")
        if not visited[neighbour]:
            if isCyclicUtil(neighbour, visited, recStack, dict_configs, verbose):
                return True
        elif recStack[neighbour]:
            if verbose:
                print(f"arc retour détecté du noeud {v} au noeud (neighbour)")
            return True

    # The node needs to be popped from recursion stack before function ends
    recStack[v] = False
    #print(f"Node {v} removed from recursion stack")
    return False

# Returns true if graph is cyclic else false
def isCyclic(dict_configs):
    dic_length = len(dict_configs)
    visited = [False] * (dic_length)
    recStack = [False] * (dic_length)
    res, verbose = False, True
    nb_cycles = 0
    for node in range(dic_length):
        if not visited[node]:
            if verbose and isCyclicUtil(node, visited, recStack, dict_configs, verbose):
                res = True
                verbose = False
                nb_cycles += 1
            if not verbose and isCyclicUtil(node, visited, recStack, dict_configs, verbose):
                nb_cycles += 1

    return res, nb_cycles


#affichage
def detect_cycles(dict_configs):
    cyclic, nb_cycles = isCyclic(dict_configs)
    print ("CYCLE DETECTE") if cyclic else print("PAS DE CYCLE")
    return cyclic, nb_cycles


from random import randint
from itertools import product
# Matrice de poids aléatoire pour un réseau de 10 cellules
def generate_random_weight_matrix(num_cells):
    weight_matrix = []
    for _ in range(num_cells):
        weights = [randint(-1, 1) for _ in range(num_cells)]
        weight_matrix.append(weights)
    return weight_matrix

def generate_configs(ncells): #all binary array of size ncells.
    return [list(i) for i in product([0, 1], repeat=ncells)]



def analyser_reseau(ncells, activation_mode, nb_simul=100):
    stable_states = 0
    cyclic_states = 0
    dict_configs = {}

    config_initiale = [randint(0, 1) for _ in range(ncells)]
    matrice_poids = generate_random_weight_matrix(ncells)
    init = True


    if (activation_mode == "synchrone"):
        print("----------------------------")
        print("Activation synchrone :")
        for conf in generate_configs(ncells):

            if init:
                nouvelle_config = activation_synchrone(config_initiale, matrice_poids, ncells)
                #print(f"Transition : ", config_initiale, "->", nouvelle_config)
                value_conf, value_new_conf = convert_dec(conf, 10), convert_dec(nouvelle_config, 10)
                #print(value_conf, "->", value_new_conf)
                dict_configs[value_conf] = value_new_conf
                if config_initiale == nouvelle_config:
                    print("Config stable")
                    stable_states += 1
                init = not init
            else:
                nouvelle_config = activation_synchrone(conf, matrice_poids, ncells)
                #print(f"Transition : ", conf, "->", nouvelle_config)
                value_conf, value_new_conf = convert_dec(conf, 10), convert_dec(nouvelle_config, 10)
                #print(value_conf, "->", value_new_conf)
                dict_configs[value_conf] = value_new_conf
                if conf == nouvelle_config:
                    print("Config stable")
                    stable_states += 1
        cyclic, nb_cycles = detect_cycles(dict_configs)
        if cyclic:
            cyclic_states += nb_cycles
    elif (activation_mode == "asynchrone"):
        print("----------------------------")
        print("Activation asynchrone :")
        for conf in generate_configs(ncells):

            if init:
                nouvelle_config = activation_asynchrone(config_initiale, matrice_poids, ncells)
                #print(f"Transition : ", config_initiale, "->", nouvelle_config)
                value_conf, value_new_conf = convert_dec(conf, 10), convert_dec(nouvelle_config, 10)
                #print(value_conf, "->", value_new_conf)
                dict_configs[value_conf] = value_new_conf
                if config_initiale == nouvelle_config:
                    print("Config stable")
                    stable_states += 1
                init = not init
            else:
                nouvelle_config = activation_asynchrone(conf, matrice_poids, ncells)
                #print(f"Transition : ", conf, "->", nouvelle_config)
                value_conf, value_new_conf = convert_dec(conf, 10), convert_dec(nouvelle_config, 10)
                #print(value_conf, "->", value_new_conf)
                dict_configs[value_conf] = value_new_conf
                if conf == nouvelle_config:
                    print("Config stable")
                    stable_states += 1
        cyclic, nb_cycles = detect_cycles(dict_configs)
        if cyclic:
            cyclic_states += nb_cycles

    else:
        raise ValueError("Activation_mode doit être 'synchronous' ou 'asynchronous'")



    return stable_states, cyclic_states


stable_states, cyclic_states = analyser_reseau(10, 'synchrone')

print(f"Config stables en synchrone : {stable_states}")
print(f"Cycles en synchrone : {cyclic_states}")
print("----------------------------")
print("----------------------------")

stable_states, cyclic_states = analyser_reseau(10, 'asynchrone')

print(f"Config stables en asynchrone : {stable_states}")
print(f"Cycles en asynchrone : {cyclic_states}")
print("----------------------------")
```

```
----------------------------
Activation synchrone :
arc retour détecté du noeud 480 au noeud 512
CYCLE DETECTE
Config stables en synchrone : 0
Cycles en synchrone : 488
----------------------------
----------------------------
----------------------------
Activation asynchrone :
Config stable
Config stable
Config stable
arc retour détecté du noeud 145 au noeud 9
CYCLE DETECTE
Config stables en asynchrone : 3
Cycles en asynchrone : 208
----------------------------
```