# Anatomie d'une salade

**Nom :** LALEU Maxime
**Numéro de candidat :** 19276

**TIPE – Travaux d'Initiative Personnelle Encadrés**
**Session 2025**

**graph.py**

```python
import networkx as nx
import matplotlib.pyplot as plt
import sqlite3 as sql
import winsound
import time
import threading
import keyboard
import pickle
import heapq

class Graphe:
    def __init__(self):
        self.graph = nx.Graph()

    def ajouter_sommet(self, sommet):
        """Ajoute un sommet au graphe."""
        self.graph.add_node(sommet)

    def ajouter_arrete(self, sommet1, sommet2, poids=None):
        """Ajoute une arête entre deux sommets. Un poids peut être ajouté."""
        self.graph.add_edge(sommet1, sommet2, weight=poids)

    def existe_arrete(self, sommet1, sommet2):
        """Vérifie si une arête existe entre deux sommets."""
        return self.graph.has_edge(sommet1, sommet2)

    def poids_arrete(self, sommet1, sommet2):
        """Recupère le poids d'une arrete"""
        return self.graph.get_edge_data(sommet1, sommet2)['weight']

    def incrementer_poids_arrete(self, sommet1, sommet2, pas = 1):
        if self.graph.has_edge(sommet1, sommet2):
            self.graph[sommet1][sommet2]['weight'] = self.graph[sommet1][sommet2]['weight'] + pas

    def afficher_graphe(self):
        """Affiche le graphe en utilisant matplotlib."""
        pos = nx.spring_layout(self.graph)  # Positionnement automatique des sommets
        weights = nx.get_edge_attributes(self.graph, 'weight')

        nx.draw(self.graph, pos, with_labels=True, node_color='skyblue', node_size=3000, font_size=10, font_weight='bold')
        if weights:
            nx.draw_networkx_edge_labels(self.graph, pos, edge_labels=weights)

        plt.title("Représentation du Graphe")
        plt.show()
```

```
46
47  def beep():
48      """ Émet un bip en boucle jusqu'à ce que l'utilisateur appuie sur une touche. """
49      while not keyboard.is_pressed("enter"):
50          winsound.Beep(1000, 500)
51          time.sleep(0.5)
52
53  db = str
54
55  def make_Vgs_grph(d:db, test=False, show = True)->Graphe:
56      cnect = sql.connect(d)
57      cur = cnect.cursor()
58      if test:
59          txtest = "test"
60      else:
61          txtest = ""
62
63
64      cur.execute("Select max(id) from ingred{}".format(txtest))
65      n_ingred = cur.fetchall()[0][0]
66      l_ingred = [i for i in range(1, n_ingred+1)]
67      g = Graphe()
68      for i in l_ingred:
69          g.ajouter_sommet(i)
70
71      print("Sommet ok")
72      cur.execute("select * from jointab{}".format(txtest))
73      print("Select all ok")
74      data = cur.fetchall()
75      n_data = len(data)
76      ci = 0
77      start = time.time()
78      acttime = start
79      print("z'est parti")
80      print("n data:", n_data)
81      for i in range(n_data):
82          if (ci == 100):
83              ci = 0
84              print(i, time.time()-acttime)
85              acttime = time.time()
86          ci+=1
87          for j in range(i+1, n_data):
88              r1, i1 = data[i]
89              r2, i2 = data[j]
90              if r1 == r2:
91                  if g.existe_arrete(i1, i2):
92                      g.incrementer_poids_arrete(i1, i2)
93                  else:
94                      g.ajouter_arrete(i1, i2, 1)
```

```
    print("arrete ok")
    print("Final time:", time.time()-start)
    bip_thread = threading.Thread(target=beep, daemon=True)
    bip_thread.start()
    keyboard.wait("enter")
    if (show):
        print("graphe:")
        g.afficher_graphe()
    with open("graphe{}.bin".format(txtest), "wb") as f:
        pickle.dump(g, f)
    print("ok")
    cnect.close()

# Charge le graphe depuis le fichier .bin
def charger_graphe(filepath: str):
    with open(filepath, "rb") as f:
        graphe = pickle.load(f)
    return graphe

# Calcule le PageRank pondéré
def calculer_pagerank(graphe):
    return nx.pagerank(graphe.graph, weight='weight')

# Charge la table ingred pour avoir les noms
def get_ingredient_names(db_path):
    conn = sql.connect(db_path)
    cur = conn.cursor()
    cur.execute("SELECT id, name FROM ingred")
    id_to_name = {row[0]: row[1] for row in cur.fetchall()}
    conn.close()
    return id_to_name

# Affiche du classement PageRank des ingrédients
def afficher_pagerank_top(graphe, db_path, top_n=20):
    pagerank_scores = calculer_pagerank(graphe)
    id_to_name = get_ingredient_names(db_path)

    sorted_pagerank = sorted(pagerank_scores.items(), key=lambda x: x[1], reverse=True)

    print(f"Top {top_n} ingrédients selon PageRank :\n")
    for ing_id, score in sorted_pagerank[:top_n]:
        name = id_to_name.get(ing_id, "UNKNOWN")
        print(f"{name:<20} : {score:.6f}")

class KMSTMaxTopM:
    def __init__(self, graph: nx.Graph, k: int, m: int):
        self.graph = graph
        self.k = k
        self.m = m
```

```
        self.top_trees = []
        self.tree_set = set()
        self.call_count = 0
        self.last_log = time.time()

    def add_solution(self, weight, tree):
        tree_frozen = frozenset(tree)
        if tree_frozen in self.tree_set:
            return
        self.tree_set.add(tree_frozen)
        heapq.heappush(self.top_trees, (weight, tree_frozen))
        if len(self.top_trees) > self.m:
            _, removed = heapq.heappop(self.top_trees)
            self.tree_set.remove(removed)

    def dfs(self, current, visited, edge_sum, depth):
        self.call_count += 1
        if self.call_count % 100000 == 0:
            now = time.time()
            best = max(self.top_trees, default=(0, None))[0]
            print(f"[{self.call_count} appels] Niveau {depth} | Sommet: {current} | Poids: {edge_sum} | BestMax: {best} | Δt: {now - self.last_log:.2f}s")
            self.last_log = now

        if len(visited) == self.k:
            self.add_solution(edge_sum, visited.copy())
            return

        for neighbor, data in sorted(self.graph[current].items(), key=lambda x: -x[1]['weight']):
            if neighbor in visited:
                continue
            max_edge = data['weight']
            projected = edge_sum + max_edge + (self.k - len(visited) - 1) * max_edge
            if self.top_trees and projected <= self.top_trees[0][0]:
                continue
            visited.add(neighbor)
            self.dfs(neighbor, visited, edge_sum + data['weight'], depth + 1)
            visited.remove(neighbor)

    def run(self):
        for node in self.graph.nodes:
            self.dfs(node, {node}, 0, 1)
        return sorted(self.top_trees, reverse=True)

class EdgeRMWCSTopM:
    def __init__(self, graph: nx.Graph, k: int, m: int, required_nodes: set):
        self.graph = graph
        self.k = k
```

```
        self.m = m
        self.required_nodes = required_nodes
        self.top_solutions = []
        self.solution_set = set()
        self.call_count = 0
        self.last_log = time.time()

    def add_solution(self, weight, nodes):
        frozen = frozenset(nodes)
        if frozen in self.solution_set:
            return
        self.solution_set.add(frozen)
        heapq.heappush(self.top_solutions, (weight, frozen))
        if len(self.top_solutions) > self.m:
            _, removed = heapq.heappop(self.top_solutions)
            self.solution_set.remove(removed)

    def dfs(self, current, visited, weight_sum, depth):
        self.call_count += 1
        if self.call_count % 100_000 == 0:
            now = time.time()
            best = max(self.top_solutions, default=(0, None))[0]
            print(f"[{self.call_count} calls] Depth: {depth} | Node: {current} | Weight: {weight_sum:.2f} | BestMax: {best:.2f} | Δt: {now - self.last_log:.2f}s")
            self.last_log = now

        if len(visited) == self.k:
            if self.required_nodes.issubset(visited):
                self.add_solution(weight_sum, visited.copy())
            return

        for neighbor in sorted(self.graph.neighbors(current), key=lambda n: -self.graph[current][n]['weight']):
            if neighbor in visited:
                continue

            edge_weight = self.graph[current][neighbor]['weight']
            projected_weight = weight_sum + edge_weight + (self.k - len(visited) - 1) * edge_weight

            if self.top_solutions and projected_weight <= self.top_solutions[0][0]:
                continue

            visited.add(neighbor)
            self.dfs(neighbor, visited, weight_sum + edge_weight, depth + 1)
            visited.remove(neighbor)

    def run(self):
        start_nodes = self.required_nodes if self.required_nodes else self.graph.nodes
```

```
        for start in start_nodes:
            self.dfs(start, {start}, 0, 1)
        return sorted(self.top_solutions, reverse=True)


def map_ingred (f, l):
    nl = []
    for el in l:
        nl.append(f(el))
    return nl






actions = ["fetchone", "autre ingred", "next", "quitter"]
"""

if __name__ == "__main__":
    actions = ["k-mst", "pagerank"]
    with open("graphe.bin", "rb") as f:
        g = pickle.load(f)
    print("Actions: ", actions)
    choice = input(">")
    db_path = "C:/Users/xavie/Documents/Etudes/TIPE/Salad/main.db"
    if (choice=="k-mst"):
        k = 10
        m = 15
        algo = KMSTMaxTopM(g.graph, k, m)
        print("Début de la recherche...")
        start = time.time()
        top_results = algo.run()
        end = time.time()
        print(f"\nTop {m} arbres de poids maximum pour k = {k} :")
        for i, (poids, sommets) in enumerate(top_results, 1):
            print(f"  #{i} - Poids: {poids}, Sommets:
{map_ingred(base.get_ingred,sorted(sommets))}")
        print(f"\nTemps total: {end - start:.2f} sec")
    elif (choice=="pagerank"):
        afficher_pagerank_top(g, db_path)
    elif(choice=="redgemwcs"):
        assert nx.has_path(g.graph, 1, 2), "Sommets obligatoires non connectés"

        k = 15
        m = 10
        algo = EdgeRMWCSTopM(g.graph, k, m, set([5405, 14]))
        print("Début de la recherche...")
        start = time.time()
        top_results = algo.run()
```

```
        end = time.time()
        print(f"\nTop {m} sous graphes de poids maximum pour k = {k} :")
        for i, (poids, sommets) in enumerate(top_results, 1):
            print(f"  #{i} - Poids: {poids}, Sommets:
{map_ingred(base.get_ingred,sorted(sommets))}")
        print(f"\nTemps total: {end - start:.2f} sec")

"""
file_created = True

if __name__ == "__main__":
    db_path = "C:/Users/xavie/Documents/Etudes/TIPE/Salad/main.db"
    if (file_created):
        graphe_path = "graphetest.bin"  # ou "graphetest.bin"
        g = charger_graphe(graphe_path)
        g.afficher_graphe()
        #afficher_pagerank_top(g, db_path, top_n=50)
    else:
        make_Vgs_grph(db_path, test= True, show=False)
```

**base.py**

```python
import sqlite3 as sql
import math
import ast
import sqlite3
import itertools

db = str

def list_ingred(d:db)->list[str]:
    cnect = sql.connect(d)
    cur = cnect.cursor()
    cur.execute("select NER from test order by field1")
    data = cur.fetchall()
    cnect.close()
    return list(dict.fromkeys(ing for tup in data for ing in ast.literal_eval(tup[0])))


def creer_et_remplir_tables(d:db):
    # Connexion à la base de données
    conn = sql.connect(d)
    cursor = conn.cursor()

    # Supprimer les anciennes tables pour un nettoyage (optionnel)
    cursor.execute("DROP TABLE IF EXISTS ingredtest;")
    cursor.execute("DROP TABLE IF EXISTS jointabtest;")

    # Créer les tables 'ingred' et 'jointabtest' si elles n'existent pas
    cursor.execute('''
    CREATE TABLE IF NOT EXISTS ingredtest (
        id INTEGER PRIMARY KEY,
        name TEXT UNIQUE
    );
    ''')

    cursor.execute('''
    CREATE TABLE IF NOT EXISTS jointabtest (
        recipe_id INTEGER,
        ingredient_id INTEGER,
        FOREIGN KEY (recipe_id) REFERENCES test(field1),
        FOREIGN KEY (ingredient_id) REFERENCES ingredtest(id),
        PRIMARY KEY (recipe_id, ingredient_id)
    );
    ''')

    # Créer l'index après avoir créé les tables
    cursor.execute("CREATE INDEX IF NOT EXISTS idx_ingredient_name ON ingredtest(name);")
    cursor.execute("CREATE INDEX IF NOT EXISTS idx_recipe_id ON jointabtest(recipe_id);")
```

```
48
49      # Récupérer toutes les recettes et leurs ingrédients NER
50      cursor.execute("SELECT field1, NER FROM test")
51      recettes = cursor.fetchall()
52
53      # Créer des listes pour les insertions en masse
54      ingredients_to_insert = []
55      jointab_to_insert = set()  # Utiliser un set pour éviter les duplications
56
57      # Traitement des recettes sans calculs de progression
58      for i, recette in enumerate(recettes):
59          field1 = recette[0]
60          # Changed json.loads to ast.literal_eval
61          ner_ingredients = ast.literal_eval(recette[1])
62
63          for ingredient in ner_ingredients:
64              # Ajouter l'ingrédient à la liste pour insertion en masse
65              ingredients_to_insert.append((ingredient,))
66
67              # Ajouter l'association (recipe_id, ingredient) dans un set pour éviter les
    doublons
68              jointab_to_insert.add((field1, ingredient))
69
70      # Insertion des ingrédients en masse
71      cursor.executemany("INSERT OR IGNORE INTO ingredtest (name) VALUES (?)",
    ingredients_to_insert)
72
73      # Récupérer les IDs des ingrédients nouvellement insérés
74      cursor.execute("SELECT id, name FROM ingredtest")
75      ingredient_ids = {name: id for id, name in cursor.fetchall()}
76
77      # Construire la liste des insertions pour jointab, avec les IDs des ingrédients
78      jointab_to_insert = [(field1, ingredient_ids[ingredient]) for field1, ingredient in
    jointab_to_insert]
79
80      # Insertion des associations (jointab) en masse
81      cursor.executemany("INSERT OR IGNORE INTO jointabtest (recipe_id, ingredient_id)
    VALUES (?, ?)", jointab_to_insert)
82
83      # Valider les changements et fermer la connexion
84      conn.commit()
85      conn.close()
86
87      print("Traitement terminé.")
88
89
90  def buildPMI(d: str):
91      conn = sql.connect(d)
92      cursor = conn.cursor()
93
```

```
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS PMI(
            id INTEGER,
            ext_id INTEGER,
            pmi REAL
        );
    ''')

    # Nombre total de recettes (DISTINCT est préférable)
    cursor.execute("SELECT COUNT(DISTINCT recipe_id) FROM jointabtest")
    n_recipes = cursor.fetchone()[0]

    # Probabilités individuelles P(i)
    cursor.execute(f"""
        SELECT ingredient_id, COUNT(ingredient_id) * 1.0 / {n_recipes}
        FROM jointab
        GROUP BY ingredient_id
    """)
    # Transforme en dict pour accès rapide
    self_probas = dict(cursor.fetchall())

    # Probabilités conjointes P(i, j)
    cursor.execute(f"""
        SELECT
            j1.ingredient_id AS i,
            j2.ingredient_id AS j,
            COUNT(DISTINCT j1.recipe_id) * 1.0 / {n_recipes} AS pij
        FROM jointab j1
        JOIN jointab j2
            ON j1.recipe_id = j2.recipe_id AND j1.ingredient_id < j2.ingredient_id
        GROUP BY i, j
    """)
    ext_probas = cursor.fetchall()

    for i, j, pij in ext_probas:
        pi = self_probas.get(i)
        pj = self_probas.get(j)

        if pi and pj and pij > 0:
            pmi = math.log(pij / (pi * pj))
            cursor.execute(
                "INSERT INTO PMI (id, ext_id, pmi) VALUES (?, ?, ?)",
                (i, j, pmi)
            )

    conn.commit()
    conn.close()

d = "C:/Users/xavie/Documents/Etudes/TIPE/Salad/main.db"
```

```python

def get_ingred(id):
    conn = sql.connect(d)
    cur = conn.cursor()

    cur.execute("SELECT name FROM ingred WHERE id={}".format(id))
    ingred = cur.fetchone()[0]

    conn.close()
    return ingred

def count_cooccurrences(db_path: str):
    """
    Counts the co-occurrences of each ingredient pair across all recipes
    and stores them in a new table named 'cooc'.

    Args:
        db_path (str): The path to the SQLite database file (e.g., 'main.db').
    """
    conn = None
    try:
        conn = sqlite3.connect(db_path)
        cursor = conn.cursor()

        # Crée la table cooc si elle n'éxiste pas
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS cooc (
                id INTEGER NOT NULL,
                ext_id INTEGER NOT NULL,
                cooc_count INTEGER NOT NULL,
                PRIMARY KEY (id, ext_id),
                FOREIGN KEY (id) REFERENCES ingred(id),
                FOREIGN KEY (ext_id) REFERENCES ingred(id)
            );
        """)
        cursor.execute("CREATE INDEX IF NOT EXISTS idx_cooc_id ON cooc (id);")
        cursor.execute("CREATE INDEX IF NOT EXISTS idx_cooc_ext_id ON cooc (ext_id);")
        conn.commit()
        print("Table 'cooc' created or already exists.")

        # Recupère les liens ingrédients recètes
        cursor.execute("SELECT recipe_id, ingredient_id FROM jointab ORDER BY recipe_id;")
        recipe_ingredient_data = cursor.fetchall()

        # Création du dictionaire recète -> ingrédient
        recipes_ingredients = {}
        for recipe_id, ingred_id in recipe_ingredient_data:
            if recipe_id not in recipes_ingredients:
                recipes_ingredients[recipe_id] = []
```

```
            recipes_ingredients[recipe_id].append(ingred_id)

        # Calcul des co-occurrences
        cooccurrence_counts = {}
        for recipe_id, ingredient_list in recipes_ingredients.items():
            if len(ingredient_list) >= 2:
                for ing1, ing2 in itertools.combinations(sorted(ingredient_list), 2):
                    pair = (ing1, ing2)
                    if pair not in cooccurrence_counts:
                        cooccurrence_counts[pair] = 0
                    cooccurrence_counts[pair] += 1

        # Préparation des données à l'insertion multiple
        data_to_insert = []
        for (id1, id2), count in cooccurrence_counts.items():
            data_to_insert.append((id1, id2, count))

        # Nétoyage de la table
        cursor.execute("DELETE FROM cooc;")
        conn.commit()
        print("Existing data in 'cooc' table cleared.")

        # Insertion multiple
        cursor.executemany("INSERT INTO cooc (id, ext_id, cooc_count) VALUES (?, ?, ?)",
data_to_insert)
        conn.commit()
        print(f"Successfully counted and inserted {len(data_to_insert)} co-occurrence
pairs into 'cooc' table.")

    except sqlite3.Error as e:
        print(f"An SQLite error occurred: {e}")
    finally:
        if conn:
            conn.close()

if __name__ == "__main__":

    creer_et_remplir_tables(d)
    #list_ingred(d)
    #buildPMI(d)
    #count_cooccurrences(d)
```

**fverif.c**

```c
#include "stdio.h"
#include "stdlib.h"
#include "stdbool.h"

int sup_int = 1000000;

struct couple
{
    int a;
    int b;
};
typedef struct couple couple;

struct couplage
{
    couple val;
    struct couplage* next;
};
typedef struct couplage couplage;

struct graphe
{
    int n;
    int** adj;
    int** W;
};
typedef struct graphe graphe;

graphe* init_graphe(int n){
    graphe* res = malloc(sizeof(graphe));
    res->n = n;
    res->adj = malloc(n*sizeof(int*));
    res->W = malloc(n*sizeof(int*));
    for (int i = 0; i < n; i++)
    {
        res->adj[i] = malloc(n*sizeof(int));
        res->W[i] = malloc(n*sizeof(int));
        for (int j = 0; j < n; j++)
        {
            res->adj[i][j] = 0;
            res->W[i][j] = sup_int;
        }
    }
    return res;
}

void free_graph(graphe* g){
```

```c
    for (int i = 0; i < g->n; i++)
    {
        free(g->adj[i]);
    }
    free(g->adj);
    free(g);
}

void add_edge(graphe* g, int u, int v, int w){
    g->adj[u][v] = 1;
    g->adj[v][u] = 1;
    g->W[u][v] = w;
    g->W[v][u] = w;
}

void del_edge(graphe* g, int u, int v){
    g->adj[u][v] = 0;
    g->adj[v][u] = 0;
}

couple init_couple(int a, int b){
    couple res;
    res.a = a;
    res.b = b;
    return res;
}

void free_couplage(couplage* c){
    if (c == NULL) return;
    free_couplage(c->next);
    free(c);
}


couplage* append_couplage(couplage* c, couple a){
    couplage* new = malloc(sizeof(couplage));
    new->val = a;
    new->next = c;
    return new;
}

bool est_couplage_du_graphe(graphe* g, couplage* c){
    for (couplage* actc = c; actc != NULL; actc = actc->next)
    {
        if (g->adj[actc->val.a][actc->val.b] != 1){
            return false;
        }
    }
    return true;
```

```c
}

void dfs_connexe_acyclique(int node, int parent, bool* visited, int** adj, int max, bool* cycle) {
    visited[node] = true;
    for (int v = 0; v < max; v++) {
        if (adj[node][v]) {
            if (!visited[v]) {
                dfs_connexe_acyclique(v, node, visited, adj, max, cycle);
            } else if (v != parent) {
                *cycle = true;
            }
        }
    }
}

bool est_connexe_et_acyclique(couplage* c) {
    if (c == NULL) return true;
    int max = 0;
    for (couplage* actc = c; actc != NULL; actc = actc->next) {
        if (actc->val.a > max) max = actc->val.a;
        if (actc->val.b > max) max = actc->val.b;
    }
    max += 1;
    int** adj = malloc(max * sizeof(int*));
    for (int i = 0; i < max; i++) {
        adj[i] = calloc(max, sizeof(int));
    }
    bool* used = calloc(max, sizeof(bool));
    for (couplage* actc = c; actc != NULL; actc = actc->next) {
        int a = actc->val.a;
        int b = actc->val.b;
        adj[a][b] = 1;
        adj[b][a] = 1;
        used[a] = true;
        used[b] = true;
    }
    bool* visited = calloc(max, sizeof(bool));
    bool cycle = false;
    int start = -1;
    for (int i = 0; i < max; i++) {
        if (used[i]) {
            start = i;
            break;
        }
    }
    dfs_connexe_acyclique(start, -1, visited, adj, max, &cycle);
    for (int i = 0; i < max; i++) {
        if (used[i] && !visited[i]) {
```

```
            cycle = true;
            break;
        }
    }
    for (int i = 0; i < max; i++) {
        free(adj[i]);
    }
    free(adj);
    free(used);
    free(visited);
    return !cycle;
}

int compte_sommet(couplage* c, int n){
    int res = 0;
    int* deja_vu = malloc(n*sizeof(int));
    for (int i = 0; i < n; i++)
    {
        deja_vu[i] = 0;
    }


    for (couplage* actc = c; actc != NULL; actc = actc->next)
    {
        if (deja_vu[actc->val.a] == 0){
            res++;
            deja_vu[actc->val.a] = 1;
        }
        if (deja_vu[actc->val.b] == 0){
            res++;
            deja_vu[actc->val.b] = 1;
        }
    }
    free(deja_vu);
    return res;
}

bool verif_poids(int** W, couplage* c, int w){
    int cmt = 0;
    for (couplage* actc = c; actc != NULL; actc = actc->next)
    {
        cmt += W[actc->val.a][actc->val.b];
    }
    return cmt >= w;
}

bool verif(graphe* g, couplage*c, int k, int w){
    return est_couplage_du_graphe(g,c) && est_connexe_et_acyclique(c) && compte_sommet(c,
g->n) == k && verif_poids(g->W, c, w);
```

```c
}

bool contient_terminaux(couplage* c, int* terminaux, int nb_term, int n) {
    int* vus = calloc(n, sizeof(int));
    for (couplage* actc = c; actc != NULL; actc = actc->next) {
        vus[actc->val.a] = 1;
        vus[actc->val.b] = 1;
    }
    for (int i = 0; i < nb_term; i++) {
        if (!vus[terminaux[i]]) {
            free(vus);
            return false;
        }
    }
    free(vus);
    return true;
}

void dfs_connexe(int node, bool* visited, int** adj, int max) {
    visited[node] = true;
    for (int v = 0; v < max; v++) {
        if (adj[node][v] && !visited[v]) {
            dfs_connexe(v, visited, adj, max);
        }
    }
}

bool est_connexe(couplage* c) {
    if (c == NULL) return false;
    int max = 0;
    for (couplage* actc = c; actc != NULL; actc = actc->next) {
        if (actc->val.a > max) max = actc->val.a;
        if (actc->val.b > max) max = actc->val.b;
    }
    max += 1;

    int** adj = malloc(max * sizeof(int*));
    for (int i = 0; i < max; i++) {
        adj[i] = calloc(max, sizeof(int));
    }

    bool* used = calloc(max, sizeof(bool));
    for (couplage* actc = c; actc != NULL; actc = actc->next) {
        int a = actc->val.a;
        int b = actc->val.b;
        adj[a][b] = 1;
        adj[b][a] = 1;
        used[a] = true;
        used[b] = true;
```

```
    }

    bool* visited = calloc(max, sizeof(bool));
    int start = -1;
    for (int i = 0; i < max; i++) {
        if (used[i]) {
            start = i;
            break;
        }
    }

    dfs_connexe(start, visited, adj, max);

    for (int i = 0; i < max; i++) {
        if (used[i] && !visited[i]) {
            for (int j = 0; j < max; j++) free(adj[j]);
            free(adj);
            free(used);
            free(visited);
            return false;
        }
    }

    for (int i = 0; i < max; i++) free(adj[i]);
    free(adj);
    free(used);
    free(visited);
    return true;
}

bool verif_edge_rmwcs(graphe* g, couplage* c, int poids_min, int* terminaux, int nb_term)
{
    return est_couplage_du_graphe(g, c)
        && est_connexe(c)
        && contient_terminaux(c, terminaux, nb_term, g->n)
        && verif_poids(g->W, c, poids_min);
}


int main(void){
    int n = 6;
    graphe* g = init_graphe(n);
    add_edge(g, 1, 2, 2);
    add_edge(g, 3, 2, 3);
    add_edge(g, 1, 3, 1);
    add_edge(g, 3, 5, 2);
    add_edge(g, 1, 4, 3);
    add_edge(g, 5, 4, 3);
    couplage* c = NULL;
```

```c
    c = append_couplage(c, init_couple(1, 2));
    c = append_couplage(c, init_couple(3, 2));
    c = append_couplage(c, init_couple(3, 5));

    if (verif(g,c,4,6)) {
        printf("C'est un arbre à 5 sommets\n");
    } else {
        printf("Ce n'est pas un arbre à 5 sommets\n");
    }

    c = NULL;
    c = append_couplage(c, init_couple(1, 2));
    c = append_couplage(c, init_couple(3, 2));
    c = append_couplage(c, init_couple(3, 5));

    int terminaux[2] = {1, 5}; // sommets imposés
    int seuil = 6;

    if (verif_edge_rmwcs(g, c, seuil, terminaux, 2)) {
        printf("Certificat valide pour edge-r-MWCS\n");
    } else {
        printf("Certificat invalide\n");
    }

    c = NULL;
    c = append_couplage(c, init_couple(1, 2));
    c = append_couplage(c, init_couple(3, 2));

    if (verif_edge_rmwcs(g, c, seuil, terminaux, 2)) {
        printf("Certificat valide pour edge-r-MWCS\n");
    } else {
        printf("Certificat invalide\n");
    }

    c = NULL;
    c = append_couplage(c, init_couple(1, 2));
    c = append_couplage(c, init_couple(4, 5));  // déconnecté de (1,2)
    if (verif_edge_rmwcs(g, c, seuil, terminaux, 2)) {
        printf("Certificat valide pour edge-r-MWCS\n");
    } else {
        printf("Certificat invalide\n");
    }

    free_couplage(c);
    free_graph(g);
    return 0;
}
```