

# Anatomie d'une salade

**Nom :** LALEU Maxime

**Numéro de candidat :** 19276

## graph.py

```
1 import networkx as nx
2 import matplotlib.pyplot as plt
3 import sqlite3 as sql
4 import winsound
5 import time
6 import threading
7 import keyboard
8 import pickle
9 import heapq
10
11 class Graphe:
12     def __init__(self):
13         self.graph = nx.Graph()
14
15     def ajouter_sommet(self, sommet):
16         """Ajoute un sommet au graphe."""
17         self.graph.add_node(sommet)
18
19     def ajouter_arrete(self, sommet1, sommet2, poids=None):
20         """Ajoute une arête entre deux sommets. Un poids peut être ajouté."""
21         self.graph.add_edge(sommet1, sommet2, weight=poids)
22
23     def existe_arrete(self, sommet1, sommet2):
24         """Vérifie si une arête existe entre deux sommets."""
25         return self.graph.has_edge(sommet1, sommet2)
26
27     def poids_arrete(self, sommet1, sommet2):
28         """Recupère le poids d'une arrete"""
29         return self.graph.get_edge_data(sommet1, sommet2)['weight']
30
31     def incrementer_poids_arrete(self, sommet1, sommet2, pas = 1):
32         if self.graph.has_edge(sommet1, sommet2):
33             self.graph[sommet1][sommet2]['weight'] = self.graph[sommet1][sommet2]
34             ['weight'] + pas
35
36     def afficher_graphe(self):
37         """Affiche le graphe en utilisant matplotlib."""
38         pos = nx.spring_layout(self.graph) # Positionnement automatique des sommets
39         weights = nx.get_edge_attributes(self.graph, 'weight')
40
41         nx.draw(self.graph, pos, with_labels=True, node_color='skyblue', node_size=3000,
42 font_size=10, font_weight='bold')
43
44         if weights:
45             nx.draw_networkx_edge_labels(self.graph, pos, edge_labels=weights)
46
47         plt.title("Représentation du Graphe")
48         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         txttest = "test"
60     else:
61         txttest = ""
62
63
64     cur.execute("Select max(id) from ingred{}".format(txttest))
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(txttest))
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.increments_poids_arrete(i1, i2)
93                 else:
94                     g.ajouter_arrete(i1, i2, 1)

```

```

95     print("arrete ok")
96     print("Final time:", time.time()-start)
97     bip_thread = threading.Thread(target=beep, daemon=True)
98     bip_thread.start()
99     keyboard.wait("enter")
100     if (show):
101         print("graphe:")
102         g.afficher_graphe()
103     with open("graphe{}.bin".format(txtest), "wb") as f:
104         pickle.dump(g, f)
105     print("ok")
106     cnect.close()
107
108 # Charge le graphe depuis le fichier .bin
109 def charger_graphe(filepath: str):
110     with open(filepath, "rb") as f:
111         graphe = pickle.load(f)
112     return graphe
113
114 # Calcule le PageRank pondéré
115 def calculer_pagerank(graphe):
116     return nx.pagerank(graphe.graph, weight='weight')
117
118 # Charge la table ingred pour avoir les noms
119 def get_ingredient_names(db_path):
120     conn = sql.connect(db_path)
121     cur = conn.cursor()
122     cur.execute("SELECT id, name FROM ingred")
123     id_to_name = {row[0]: row[1] for row in cur.fetchall()}
124     conn.close()
125     return id_to_name
126
127 # Affiche du classement PageRank des ingrédients
128 def afficher_pagerank_top(graphe, db_path, top_n=20):
129     pagerank_scores = calculer_pagerank(graphe)
130     id_to_name = get_ingredient_names(db_path)
131
132     sorted_pagerank = sorted(pagerank_scores.items(), key=lambda x: x[1], reverse=True)
133
134     print(f"Top {top_n} ingrédients selon PageRank :\n")
135     for ing_id, score in sorted_pagerank[:top_n]:
136         name = id_to_name.get(ing_id, "UNKNOWN")
137         print(f"{name:<20} : {score:.6f}")
138
139 class KMSTMaxTopM:
140     def __init__(self, graph: nx.Graph, k: int, m: int):
141         self.graph = graph
142         self.k = k
143         self.m = m

```

```

144         self.top_trees = []
145         self.tree_set = set()
146         self.call_count = 0
147         self.last_log = time.time()
148
149     def add_solution(self, weight, tree):
150         tree_frozen = frozenset(tree)
151         if tree_frozen in self.tree_set:
152             return
153         self.tree_set.add(tree_frozen)
154         heapq.heappush(self.top_trees, (weight, tree_frozen))
155         if len(self.top_trees) > self.m:
156             _, removed = heapq.heappop(self.top_trees)
157             self.tree_set.remove(removed)
158
159     def dfs(self, current, visited, edge_sum, depth):
160         self.call_count += 1
161         if self.call_count % 100000 == 0:
162             now = time.time()
163             best = max(self.top_trees, default=(0, None))[0]
164             print(f"[{self.call_count} appels] Niveau {depth} | Sommet: {current} | Poids: {edge_sum} | BestMax: {best} | Δt: {now - self.last_log:.2f}s")
165             self.last_log = now
166
167         if len(visited) == self.k:
168             self.add_solution(edge_sum, visited.copy())
169             return
170
171         for neighbor, data in sorted(self.graph[current].items(), key=lambda x: -x[1]
['weight']):
172             if neighbor in visited:
173                 continue
174             max_edge = data['weight']
175             projected = edge_sum + max_edge + (self.k - len(visited) - 1) * max_edge
176             if self.top_trees and projected <= self.top_trees[0][0]:
177                 continue
178             visited.add(neighbor)
179             self.dfs(neighbor, visited, edge_sum + data['weight'], depth + 1)
180             visited.remove(neighbor)
181
182     def run(self):
183         for node in self.graph.nodes:
184             self.dfs(node, {node}, 0, 1)
185         return sorted(self.top_trees, reverse=True)
186
187 class EdgeRMWCSTopM:
188     def __init__(self, graph: nx.Graph, k: int, m: int, required_nodes: set):
189         self.graph = graph
190         self.k = k

```

```

191         self.m = m
192         self.required_nodes = required_nodes
193         self.top_solutions = []
194         self.solution_set = set()
195         self.call_count = 0
196         self.last_log = time.time()
197
198     def add_solution(self, weight, nodes):
199         frozen = frozenset(nodes)
200         if frozen in self.solution_set:
201             return
202         self.solution_set.add(frozen)
203         heapq.heappush(self.top_solutions, (weight, frozen))
204         if len(self.top_solutions) > self.m:
205             _, removed = heapq.heappop(self.top_solutions)
206             self.solution_set.remove(removed)
207
208     def dfs(self, current, visited, weight_sum, depth):
209         self.call_count += 1
210         if self.call_count % 100_000 == 0:
211             now = time.time()
212             best = max(self.top_solutions, default=(0, None))[0]
213             print(f"[{self.call_count} calls] Depth: {depth} | Node: {current} | Weight:
214 {weight_sum:.2f} | BestMax: {best:.2f} | Δt: {now - self.last_log:.2f}s")
215             self.last_log = now
216
217         if len(visited) == self.k:
218             if self.required_nodes.issubset(visited):
219                 self.add_solution(weight_sum, visited.copy())
220             return
221
222         for neighbor in sorted(self.graph.neighbors(current), key=lambda n: -
223 self.graph[current][n]['weight']):
224             if neighbor in visited:
225                 continue
226
227             edge_weight = self.graph[current][neighbor]['weight']
228             projected_weight = weight_sum + edge_weight + (self.k - len(visited) - 1) *
229 edge_weight
230
231             if self.top_solutions and projected_weight <= self.top_solutions[0][0]:
232                 continue
233
234             visited.add(neighbor)
235             self.dfs(neighbor, visited, weight_sum + edge_weight, depth + 1)
236             visited.remove(neighbor)
237
238     def run(self):
239         start_nodes = self.required_nodes if self.required_nodes else self.graph.nodes

```

```

237         for start in start_nodes:
238             self.dfs(start, {start}, 0, 1)
239         return sorted(self.top_solutions, reverse=True)
240
241
242 def map_ingred (f, l):
243     nl = []
244     for el in l:
245         nl.append(f(el))
246     return nl
247
248
249
250
251
252 actions = ["fetchone", "autre ingred", "next", "quitter"]
253 """
254
255 if __name__ == "__main__":
256     actions = ["k-mst", "pagerank"]
257     with open("graphe.bin", "rb") as f:
258         g = pickle.load(f)
259     print("Actions: ", actions)
260     choice = input(">")
261     db_path = "C:/Users/xavie/Documents/Etudes/TIPE/Salad/main.db"
262     if (choice=="k-mst"):
263         k = 10
264         m = 15
265         algo = KMSTMaxTopM(g.graph, k, m)
266         print("Début de la recherche...")
267         start = time.time()
268         top_results = algo.run()
269         end = time.time()
270         print(f"\nTop {m} arbres de poids maximum pour k = {k} :")
271         for i, (poids, sommets) in enumerate(top_results, 1):
272             print(f"  #{i} - Poids: {poids}, Sommets:
{map_ingred(base.get_ingred,sorted(sommets))}")
273             print(f"\nTemps total: {end - start:.2f} sec")
274         elif (choice=="pagerank"):
275             afficher_pagerank_top(g, db_path)
276         elif(choice=="redgemwcs"):
277             assert nx.has_path(g.graph, 1, 2), "Sommets obligatoires non connectés"
278
279             k = 15
280             m = 10
281             algo = EdgeRMWCSTopM(g.graph, k, m, set([5405, 14]))
282             print("Début de la recherche...")
283             start = time.time()
284             top_results = algo.run()

```

```

285         end = time.time()
286         print(f"\nTop {m} sous graphes de poids maximum pour k = {k} :")
287         for i, (poids, sommets) in enumerate(top_results, 1):
288             print(f"    #{i} - Poids: {poids}, Sommets:
{map_ingred(base.get_ingred,sorted(sommets))}")
289             print(f"\nTemps total: {end - start:.2f} sec")
290
291     """
292     file_created = True
293
294     if __name__ == "__main__":
295         db_path = "C:/Users/xavie/Documents/Etudes/TIPE/Salad/main.db"
296         if (file_created):
297             graphe_path = "graphetest.bin" # ou "graphetest.bin"
298             g = charger_graphe(graphe_path)
299             g.afficher_graphe()
300             #afficher_pagerank_top(g, db_path, top_n=50)
301         else:
302             make_Vgs_grph(db_path, test= True, show=False)

```



## base.py

```
1 import sqlite3 as sql
2 import math
3 import ast
4 import sqlite3
5 import itertools
6
7 db = str
8
9 def list_ingred(d:db)->list[str]:
10     cnect = sql.connect(d)
11     cur = cnect.cursor()
12     cur.execute("select NER from test order by field1")
13     data = cur.fetchall()
14     cnect.close()
15     return list(dict.fromkeys(ing for tup in data for ing in ast.literal_eval(tup[0])))
16
17
18 def creer_et_remplir_tables(d:db):
19     # Connexion à la base de données
20     conn = sql.connect(d)
21     cursor = conn.cursor()
22
23     # Supprimer les anciennes tables pour un nettoyage (optionnel)
24     cursor.execute("DROP TABLE IF EXISTS ingredtest;")
25     cursor.execute("DROP TABLE IF EXISTS jointabtest;")
26
27     # Créer les tables 'ingred' et 'jointabtest' si elles n'existent pas
28     cursor.execute('''
29 CREATE TABLE IF NOT EXISTS ingredtest (
30     id INTEGER PRIMARY KEY,
31     name TEXT UNIQUE
32 );
33 ''')
34
35     cursor.execute('''
36 CREATE TABLE IF NOT EXISTS jointabtest (
37     recipe_id INTEGER,
38     ingredient_id INTEGER,
39     FOREIGN KEY (recipe_id) REFERENCES test(field1),
40     FOREIGN KEY (ingredient_id) REFERENCES ingredtest(id),
41     PRIMARY KEY (recipe_id, ingredient_id)
42 );
43 ''')
44
45     # Créer l'index après avoir créé les tables
46     cursor.execute("CREATE INDEX IF NOT EXISTS idx_ingredient_name ON ingredtest(name);")
47     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

```

```

94     cursor.execute('''
95         CREATE TABLE IF NOT EXISTS PMI(
96             id INTEGER,
97             ext_id INTEGER,
98             pmi REAL
99         );
100     ''')
101
102     # Nombre total de recettes (DISTINCT est préférable)
103     cursor.execute("SELECT COUNT(DISTINCT recipe_id) FROM jointabtest")
104     n_recipes = cursor.fetchone()[0]
105
106     # Probabilités individuelles P(i)
107     cursor.execute(f"""
108         SELECT ingredient_id, COUNT(ingredient_id) * 1.0 / {n_recipes}
109         FROM jointab
110         GROUP BY ingredient_id
111     """)
112     # Transforme en dict pour accès rapide
113     self_probas = dict(cursor.fetchall())
114
115     # Probabilités conjointes P(i, j)
116     cursor.execute(f"""
117         SELECT
118             j1.ingredient_id AS i,
119             j2.ingredient_id AS j,
120             COUNT(DISTINCT j1.recipe_id) * 1.0 / {n_recipes} AS pij
121         FROM jointab j1
122         JOIN jointab j2
123             ON j1.recipe_id = j2.recipe_id AND j1.ingredient_id < j2.ingredient_id
124         GROUP BY i, j
125     """)
126     ext_probas = cursor.fetchall()
127
128     for i, j, pij in ext_probas:
129         pi = self_probas.get(i)
130         pj = self_probas.get(j)
131
132         if pi and pj and pij > 0:
133             pmi = math.log(pij / (pi * pj))
134             cursor.execute(
135                 "INSERT INTO PMI (id, ext_id, pmi) VALUES (?, ?, ?)",
136                 (i, j, pmi)
137             )
138
139     conn.commit()
140     conn.close()
141
142     d = "C:/Users/xavie/Documents/Etudes/TIPE/Salad/main.db"

```

```

143
144 def get_ingred(id):
145     conn = sql.connect(d)
146     cur = conn.cursor()
147
148     cur.execute("SELECT name FROM ingred WHERE id={}".format(id))
149     ingred = cur.fetchone()[0]
150
151     conn.close()
152     return ingred
153
154 def count_cooccurrences(db_path: str):
155     """
156     Counts the co-occurrences of each ingredient pair across all recipes
157     and stores them in a new table named 'cooc'.
158
159     Args:
160         db_path (str): The path to the SQLite database file (e.g., 'main.db').
161     """
162     conn = None
163     try:
164         conn = sqlite3.connect(db_path)
165         cursor = conn.cursor()
166
167         # Crée la table cooc si elle n'existe pas
168         cursor.execute("""
169             CREATE TABLE IF NOT EXISTS cooc (
170                 id INTEGER NOT NULL,
171                 ext_id INTEGER NOT NULL,
172                 cooc_count INTEGER NOT NULL,
173                 PRIMARY KEY (id, ext_id),
174                 FOREIGN KEY (id) REFERENCES ingred(id),
175                 FOREIGN KEY (ext_id) REFERENCES ingred(id)
176             );
177         """)
178         cursor.execute("CREATE INDEX IF NOT EXISTS idx_cooc_id ON cooc (id);")
179         cursor.execute("CREATE INDEX IF NOT EXISTS idx_cooc_ext_id ON cooc (ext_id);")
180         conn.commit()
181         print("Table 'cooc' created or already exists.")
182
183         # Recupère les liens ingrédients recettes
184         cursor.execute("SELECT recipe_id, ingredient_id FROM jointab ORDER BY recipe_id;")
185         recipe_ingredient_data = cursor.fetchall()
186
187         # Création du dictionnaire recette -> ingrédient
188         recipes_ingredients = {}
189         for recipe_id, ingred_id in recipe_ingredient_data:
190             if recipe_id not in recipes_ingredients:
191                 recipes_ingredients[recipe_id] = []

```

```

192         recipes_ingredients[recipe_id].append(ingred_id)
193
194     # Calcul des co-occurrences
195     cooccurrence_counts = {}
196     for recipe_id, ingredient_list in recipes_ingredients.items():
197         if len(ingredient_list) >= 2:
198             for ing1, ing2 in itertools.combinations(sorted(ingredient_list), 2):
199                 pair = (ing1, ing2)
200                 if pair not in cooccurrence_counts:
201                     cooccurrence_counts[pair] = 0
202                 cooccurrence_counts[pair] += 1
203
204     # Préparation des données à l'insertion multiple
205     data_to_insert = []
206     for (id1, id2), count in cooccurrence_counts.items():
207         data_to_insert.append((id1, id2, count))
208
209     # Nettoyage de la table
210     cursor.execute("DELETE FROM cooc;")
211     conn.commit()
212     print("Existing data in 'cooc' table cleared.")
213
214     # Insertion multiple
215     cursor.executemany("INSERT INTO cooc (id, ext_id, cooc_count) VALUES (?, ?, ?)",
data_to_insert)
216     conn.commit()
217     print(f"Successfully counted and inserted {len(data_to_insert)} co-occurrence
pairs into 'cooc' table.")
218
219     except sqlite3.Error as e:
220         print(f"An SQLite error occurred: {e}")
221     finally:
222         if conn:
223             conn.close()
224
225 if __name__ == "__main__":
226
227     creer_et_remplir_tables(d)
228     #list_ingred(d)
229     #buildPMI(d)
230     #count_cooccurrences(d)

```

## fverif.c

```
1  #include "stdio.h"
2  #include "stdlib.h"
3  #include "stdbool.h"
4
5  int sup_int = 1000000;
6
7  struct couple
8  {
9      int a;
10     int b;
11 };
12 typedef struct couple couple;
13
14 struct couplage
15 {
16     couple val;
17     struct couplage* next;
18 };
19 typedef struct couplage couplage;
20
21 struct graphe
22 {
23     int n;
24     int** adj;
25     int** W;
26 };
27 typedef struct graphe graphe;
28
29 graphe* init_graphe(int n){
30     graphe* res = malloc(sizeof(graphe));
31     res->n = n;
32     res->adj = malloc(n*sizeof(int*));
33     res->W = malloc(n*sizeof(int*));
34     for (int i = 0; i < n; i++)
35     {
36         res->adj[i] = malloc(n*sizeof(int));
37         res->W[i] = malloc(n*sizeof(int));
38         for (int j = 0; j < n; j++)
39         {
40             res->adj[i][j] = 0;
41             res->W[i][j] = sup_int;
42         }
43     }
44     return res;
45 }
46
47 void free_graph(graphe* g){
```

```

48     for (int i = 0; i < g->n; i++)
49     {
50         free(g->adj[i]);
51     }
52     free(g->adj);
53     free(g);
54 }
55
56 void add_edge(graphe* g, int u, int v, int w){
57     g->adj[u][v] = 1;
58     g->adj[v][u] = 1;
59     g->W[u][v] = w;
60     g->W[v][u] = w;
61 }
62
63 void del_edge(graphe* g, int u, int v){
64     g->adj[u][v] = 0;
65     g->adj[v][u] = 0;
66 }
67
68 couple init_couple(int a, int b){
69     couple res;
70     res.a = a;
71     res.b = b;
72     return res;
73 }
74
75 void free_couplage(couplage* c){
76     if (c == NULL) return;
77     free_couplage(c->next);
78     free(c);
79 }
80
81
82 couplage* append_couplage(couplage* c, couple a){
83     couplage* new = malloc(sizeof(couplage));
84     new->val = a;
85     new->next = c;
86     return new;
87 }
88
89 bool est_couplage_du_graphe(graphe* g, couplage* c){
90     for (couplage* actc = c; actc != NULL; actc = actc->next)
91     {
92         if (g->adj[actc->val.a][actc->val.b] != 1){
93             return false;
94         }
95     }
96     return true;

```

```

97 }
98
99 void dfs_connexe_acyclique(int node, int parent, bool* visited, int** adj, int max, bool*
cycle) {
100     visited[node] = true;
101     for (int v = 0; v < max; v++) {
102         if (adj[node][v]) {
103             if (!visited[v]) {
104                 dfs_connexe_acyclique(v, node, visited, adj, max, cycle);
105             } else if (v != parent) {
106                 *cycle = true;
107             }
108         }
109     }
110 }
111
112 bool est_connexe_et_acyclique(couplage* c) {
113     if (c == NULL) return true;
114     int max = 0;
115     for (couplage* actc = c; actc != NULL; actc = actc->next) {
116         if (actc->val.a > max) max = actc->val.a;
117         if (actc->val.b > max) max = actc->val.b;
118     }
119     max += 1;
120     int** adj = malloc(max * sizeof(int*));
121     for (int i = 0; i < max; i++) {
122         adj[i] = calloc(max, sizeof(int));
123     }
124     bool* used = calloc(max, sizeof(bool));
125     for (couplage* actc = c; actc != NULL; actc = actc->next) {
126         int a = actc->val.a;
127         int b = actc->val.b;
128         adj[a][b] = 1;
129         adj[b][a] = 1;
130         used[a] = true;
131         used[b] = true;
132     }
133     bool* visited = calloc(max, sizeof(bool));
134     bool cycle = false;
135     int start = -1;
136     for (int i = 0; i < max; i++) {
137         if (used[i]) {
138             start = i;
139             break;
140         }
141     }
142     dfs_connexe_acyclique(start, -1, visited, adj, max, &cycle);
143     for (int i = 0; i < max; i++) {
144         if (used[i] && !visited[i]) {

```



```

145         cycle = true;
146         break;
147     }
148 }
149 for (int i = 0; i < max; i++) {
150     free(adj[i]);
151 }
152 free(adj);
153 free(used);
154 free(visited);
155 return !cycle;
156 }
157
158 int compte_sommet(couplage* c, int n){
159     int res = 0;
160     int* deja_vu = malloc(n*sizeof(int));
161     for (int i = 0; i < n; i++)
162     {
163         deja_vu[i] = 0;
164     }
165
166     for (couplage* actc = c; actc != NULL; actc = actc->next)
167     {
168         if (deja_vu[actc->val.a] == 0){
169             res++;
170             deja_vu[actc->val.a] = 1;
171         }
172         if (deja_vu[actc->val.b] == 0){
173             res++;
174             deja_vu[actc->val.b] = 1;
175         }
176     }
177     free(deja_vu);
178     return res;
179 }
180
181
182 bool verif_poids(int** W, couplage* c, int w){
183     int cmt = 0;
184     for (couplage* actc = c; actc != NULL; actc = actc->next)
185     {
186         cmt += W[actc->val.a][actc->val.b];
187     }
188     return cmt >= w;
189 }
190
191 bool verif(graphe* g, couplage* c, int k, int w){
192     return est_couplage_du_graphe(g,c) && est_connexe_et_acyclique(c) && compte_sommet(c,
g->n) == k && verif_poids(g->W, c, w);

```

```

193 }
194
195 bool contient_terminaux(couplage* c, int* terminaux, int nb_term, int n) {
196     int* vus = calloc(n, sizeof(int));
197     for (couplage* actc = c; actc != NULL; actc = actc->next) {
198         vus[actc->val.a] = 1;
199         vus[actc->val.b] = 1;
200     }
201     for (int i = 0; i < nb_term; i++) {
202         if (!vus[terminaux[i]]) {
203             free(vus);
204             return false;
205         }
206     }
207     free(vus);
208     return true;
209 }
210
211 void dfs_connexe(int node, bool* visited, int** adj, int max) {
212     visited[node] = true;
213     for (int v = 0; v < max; v++) {
214         if (adj[node][v] && !visited[v]) {
215             dfs_connexe(v, visited, adj, max);
216         }
217     }
218 }
219
220 bool est_connexe(couplage* c) {
221     if (c == NULL) return false;
222     int max = 0;
223     for (couplage* actc = c; actc != NULL; actc = actc->next) {
224         if (actc->val.a > max) max = actc->val.a;
225         if (actc->val.b > max) max = actc->val.b;
226     }
227     max += 1;
228
229     int** adj = malloc(max * sizeof(int*));
230     for (int i = 0; i < max; i++) {
231         adj[i] = calloc(max, sizeof(int));
232     }
233
234     bool* used = calloc(max, sizeof(bool));
235     for (couplage* actc = c; actc != NULL; actc = actc->next) {
236         int a = actc->val.a;
237         int b = actc->val.b;
238         adj[a][b] = 1;
239         adj[b][a] = 1;
240         used[a] = true;
241         used[b] = true;

```

```

242     }
243
244     bool* visited = calloc(max, sizeof(bool));
245     int start = -1;
246     for (int i = 0; i < max; i++) {
247         if (used[i]) {
248             start = i;
249             break;
250         }
251     }
252
253     dfs_connexe(start, visited, adj, max);
254
255     for (int i = 0; i < max; i++) {
256         if (used[i] && !visited[i]) {
257             for (int j = 0; j < max; j++) free(adj[j]);
258             free(adj);
259             free(used);
260             free(visited);
261             return false;
262         }
263     }
264
265     for (int i = 0; i < max; i++) free(adj[i]);
266     free(adj);
267     free(used);
268     free(visited);
269     return true;
270 }
271
272 bool verif_edge_rmwcs(graphe* g, couplage* c, int poids_min, int* terminaux, int nb_term)
273 {
274     return est_couplage_du_graphe(g, c)
275         && est_connexe(c)
276         && contient_terminaux(c, terminaux, nb_term, g->n)
277         && verif_poids(g->W, c, poids_min);
278 }
279
280 int main(void){
281     int n = 6;
282     graphe* g = init_graphe(n);
283     add_edge(g, 1, 2, 2);
284     add_edge(g, 3, 2, 3);
285     add_edge(g, 1, 3, 1);
286     add_edge(g, 3, 5, 2);
287     add_edge(g, 1, 4, 3);
288     add_edge(g, 5, 4, 3);
289     couplage* c = NULL;

```

```

290     c = append_couplage(c, init_couple(1, 2));
291     c = append_couplage(c, init_couple(3, 2));
292     c = append_couplage(c, init_couple(3, 5));
293
294     if (verif(g,c,4,6)) {
295         printf("C'est un arbre à 5 sommets\n");
296     } else {
297         printf("Ce n'est pas un arbre à 5 sommets\n");
298     }
299
300     c = NULL;
301     c = append_couplage(c, init_couple(1, 2));
302     c = append_couplage(c, init_couple(3, 2));
303     c = append_couplage(c, init_couple(3, 5));
304
305     int terminaux[2] = {1, 5}; // sommets imposés
306     int seuil = 6;
307
308     if (verif_edge_rmwcs(g, c, seuil, terminaux, 2)) {
309         printf("Certificat valide pour edge-r-MWCS\n");
310     } else {
311         printf("Certificat invalide\n");
312     }
313
314     c = NULL;
315     c = append_couplage(c, init_couple(1, 2));
316     c = append_couplage(c, init_couple(3, 2));
317
318     if (verif_edge_rmwcs(g, c, seuil, terminaux, 2)) {
319         printf("Certificat valide pour edge-r-MWCS\n");
320     } else {
321         printf("Certificat invalide\n");
322     }
323
324     c = NULL;
325     c = append_couplage(c, init_couple(1, 2));
326     c = append_couplage(c, init_couple(4, 5)); // déconnecté de (1,2)
327     if (verif_edge_rmwcs(g, c, seuil, terminaux, 2)) {
328         printf("Certificat valide pour edge-r-MWCS\n");
329     } else {
330         printf("Certificat invalide\n");
331     }
332
333     free_couplage(c);
334     free_graph(g);
335     return 0;
336 }

```