Génération de labyrinthes PACMAN

TER S5, M1 Informatique

Lesage Arno, Razafindrabe Keryann, Viale Jean-Jacques

EUR DS4H - Université Côte d'Azur

github.com/KeryannR/TER_S1_F

Sommaire

1. API

Endpoint /

Description : Vérifie que l'API est en fonctionnement.

Méthode : GET

Paramètres : Aucun

Retour : JSON avec un message de confirmation

Exemple :

```
GET /
{"message": "Maze Generator API is running!"}
```

Endpoint /generate

Description : Génère dynamiquement un labyrinthe PACMAN selon les paramètres fournis.

Méthode : GET

Paramètres :

- `xSize`, `ySize` : dimensions du labyrinthe (default 15)
- `minScore` : score minimal pour le labyrinthe généré
- `nPortal` : nombre de portails (default 1)
- `seed` : graine pour reproductibilité
- `nStep` : nombre d'itérations (default 20000)
- `maxBorderSpikeSize` : taille max des prolongements de bord
- `includeTile` : tuiles à inclure

Format du résultat de /generate

Exemple d'utilisation de la route :

```
GET /generate?xSize=30&ySize=30&nStep=20000
```

Le résultat renvoyé par l'API est un JSON structuré :

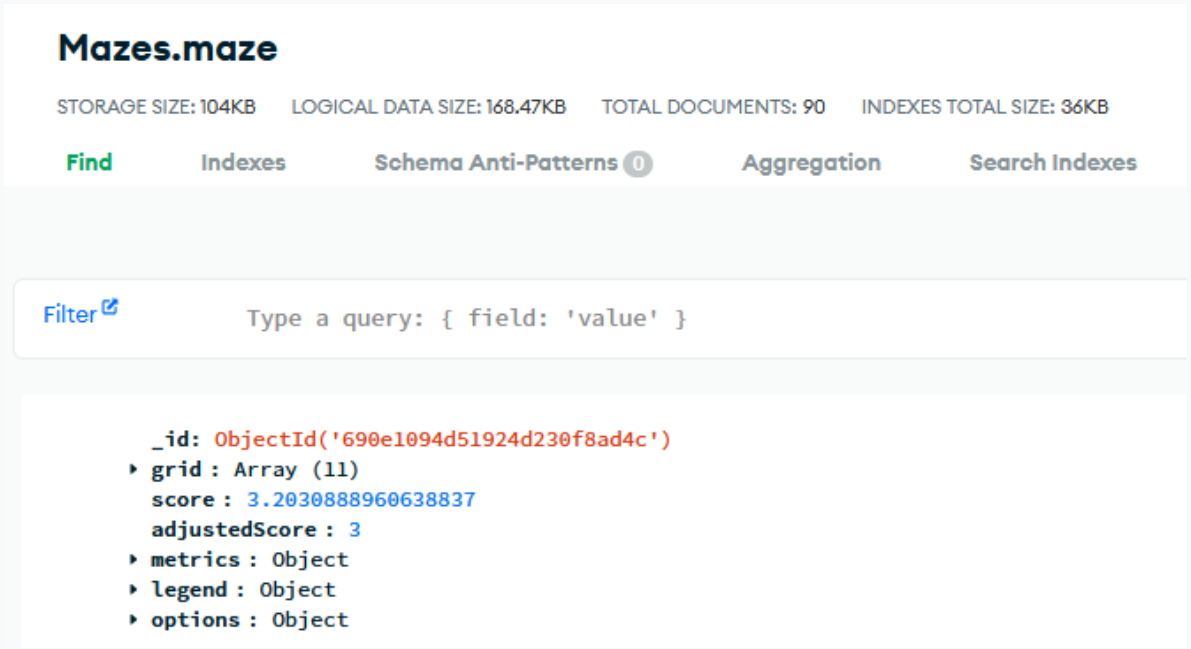

```
{
    "_id": "None",
    "grid": [[0,0,1,...],[...],...],
    "legend": {"0":"path","1":"wall","2":"phantom","3":"portal"},
    "metrics": {"Crossroads%":10.26, "Dead-Ends%":1.28, ...},
    "options": {"xSize":30,"ySize":30,"nStep":2000,...},
    "score": 2.65
}
```

Insertion dans MongoDB

Lorsqu'un labyrinthe est inséré dans MongoDB, la base lui attribue automatiquement un `_id` unique.

Cet `_id` est récupéré avec le code suivant pour ensuite être ajouté au JSON renvoyé par l'API :

```
inserted_id = collection.insert_one(json_data).inserted_id
json_data["_id"] = str(inserted_id)
```



Endpoint /get

Description : Récupère un ou plusieurs labyrinthes depuis la base selon des critères.

Méthode : GET

Paramètres : id, score, xSize, ySize, seed, nStep, nPortal, maxBorderSpikeSize, includeTile, limit

Retour : JSON avec un ou plusieurs labyrinthes correspondant aux filtres.

Exemple :

```
GET /get?xSize=30&ySize=30&limit=5
=> [{ maze1 }, { maze2 }, { maze3 }, { maze4 }, { maze5 }]
```

2. Tests

Objectif des tests

Pourquoi tester ?

Les tests permettent de garantir la **stabilité** et la **fiabilité** de notre API de génération de labyrinthes.

- Vérifier que chaque **endpoint fonctionne correctement** (‘/‘, ‘/generate‘, ‘/get‘)
- Assurer la **cohérence des résultats** entre différentes exécutions
- Tester la **robustesse face aux paramètres invalides**
- Contrôler la **performance** de génération
- Automatiser ces vérifications via un pipeline **CI/CD GitHub Actions**

Tests unitaires de l'API — Fichier : `test_mazeAPI.py`

Les tests unitaires reposent sur le module `unittest`. Ils vérifient chaque fonctionnalité critique de l'API :

- **test_home** : Vérifie la disponibilité de l'API.
- **test_generate_default / fixed_seed** : Teste la génération de labyrinthes.
- **test_invalid_params** : Vérifie la gestion d'erreurs.
- **test_performance** : S'assure que la génération reste rapide.

Exemple :

```
def test_generate_fixed_seed(self):
    r1 = self.client.get('/generate?xSize=10&ySize=10&seed=42')
    r2 = self.client.get('/generate?xSize=10&ySize=10&seed=42')
    self.assertEqual(r1.get_json()["grid"], r2.get_json()["grid"])
```

Tests d'intégration et CI/CD Fichier : `test_cicd.py`

Ces tests assurent que l'API reste fonctionnelle après chaque modification majeure :

- Vérifie la **forme du labyrinthe généré**
- Vérifie la **récupération par _id** dans MongoDB
- Garantit la **compatibilité du format JSON**

Exemple :

```
def test_get_maze_by_id(self):
    test_id = "690d1d58b9055ebd0f144bb6"
    r = self.client.get(f'/get?id={test_id}')
    data = r.get_json()[0]
    self.assertIn("_id", data)
    self.assertIn("grid", data)
```

Intégration Continue (CI/CD)GitHub Actions - Fichier :`main.yml`

Un workflow CI/CD a été mis en place pour :

- Lancer automatiquement les tests à chaque **push / pull request**
- Valider le code avant intégration sur la branche **main**

```
name: MazeGeneration CICD
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.11'
      - run: pip install -r requirements.txt
      - run: python -m unittest discover -v
```

Résultats de l'intégration continue

Exemples de pipeline

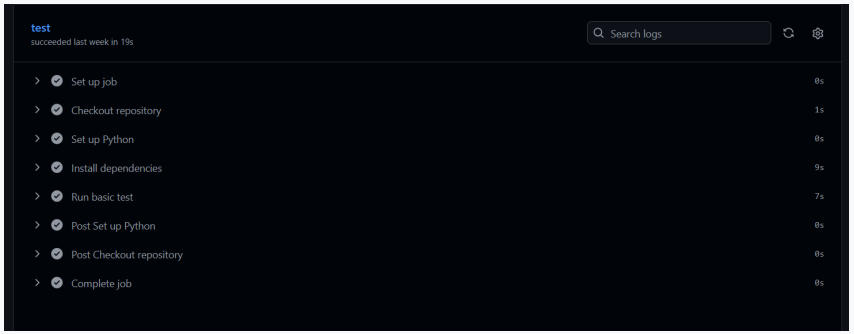


Figure: Tests réussis : tous les checks sont verts

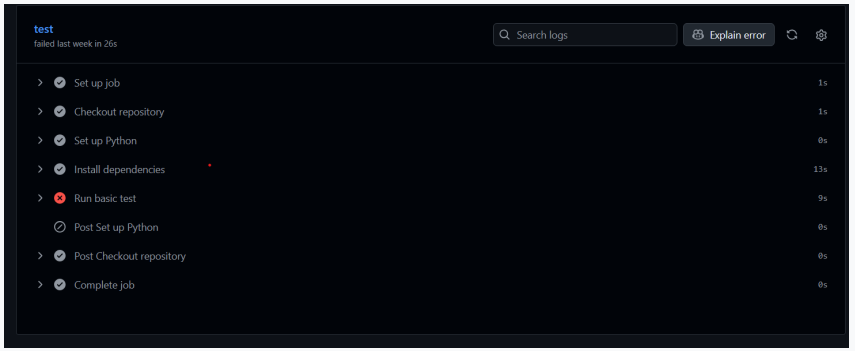


Figure: Échec : un test échoue, pipeline bloqué

→ Les erreurs sont détectées automatiquement avant la mise en production.

3. Évaluation et métriques

Génération du labyrinthe et métriques

Rappel

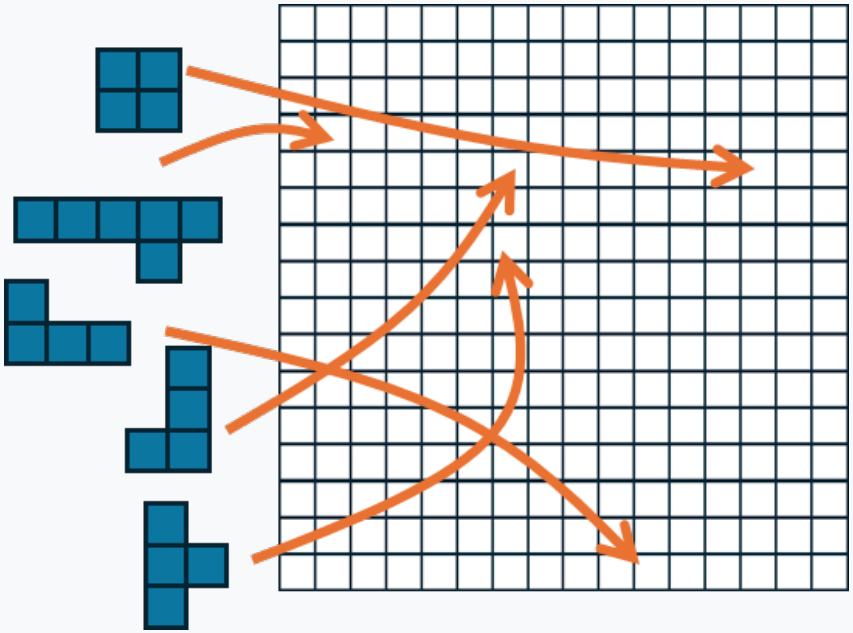


Figure: Algorithme Tetris Modifié

Métriques calculés:

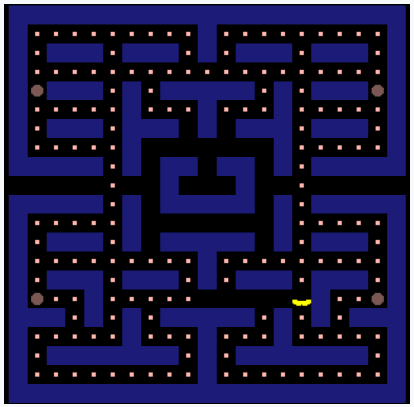
- Proportion de **carrefour**
- Proportion de **jonction**
- Proportion de **virage**
- Proportion de **ligne droite**
- Proportion de **cul-de-sac**
- Proportion de **murs**
- Proportion de **chemin**

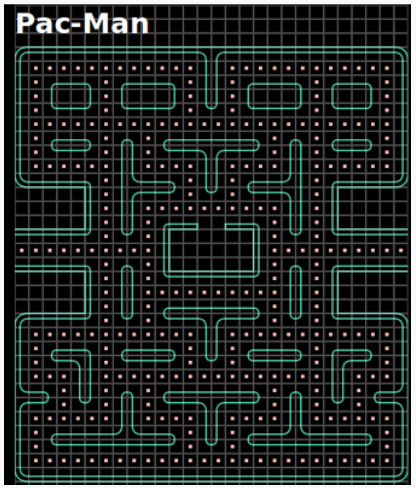
Question : Comment savoir si un labyrinthe est bon ?

1. À la main ? $\longrightarrow$ trop contraignant…
2. Automatiquement ? $\longrightarrow$ oui, mais comment ?

Mesure d'évaluation

Comparaisons





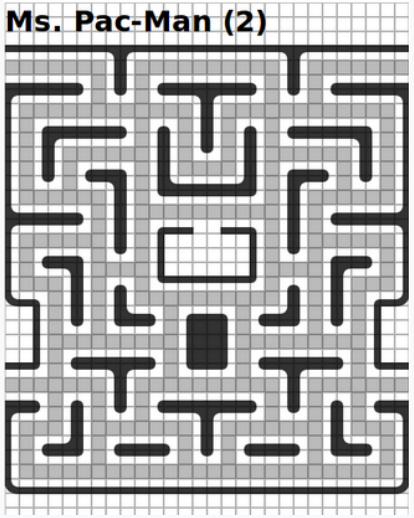


Figure: Quelques "vrai" PACMAN

En moyenne :

- **Carrefours** : 2.034% (↑)
- **Jonctions** : 11.863% (↑)
- **Virages** : 12.213%
- **Lignes droites** : 72.923% (↓)
- **Cul de sac** : 0.967%
- **Murs** : 50.766% (↓)
- **Chemins** : 46.647% (↑)

Mesure d'évaluation

Formule

Soit μ le vecteur de la moyenne des mesures sur les trois labyrinthes et x les mesures sur le labyrinthe cible.

$$\mu = \left(\mu_{\text{cross}}, \mu_{\text{junc}}, \mu_{\text{turn}}, \mu_{\text{straight}}, \mu_{\text{dead}}, \mu_{\text{wall}}, \mu_{\text{path}}\right)$$
$$x = \left(x_{\text{cross}}, x_{\text{junc}}, x_{\text{turn}}, x_{\text{straight}}, x_{\text{dead}}, x_{\text{wall}}, x_{\text{path}}\right)$$

Nous définissons **score** (μ, x) comme :

$$\textbf{score}\,(\mu, x) = 5 \times \frac{x \cdot \mu}{\|x\|\,\|\mu\|}$$

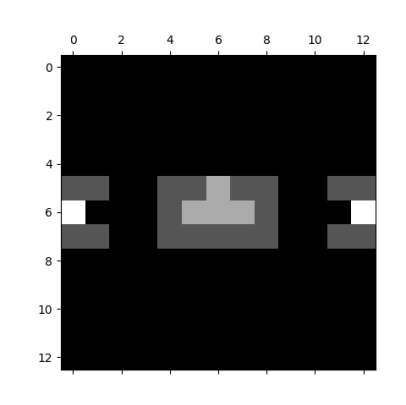
$$\mu_{\text{wall}} > 0.8 \Rightarrow \textbf{score}\,(\mu, x) = 0$$
$$\mu_{\text{path}} > 0.8 \Rightarrow \textbf{score}\,(\mu, x) = 0$$

Interprétation :

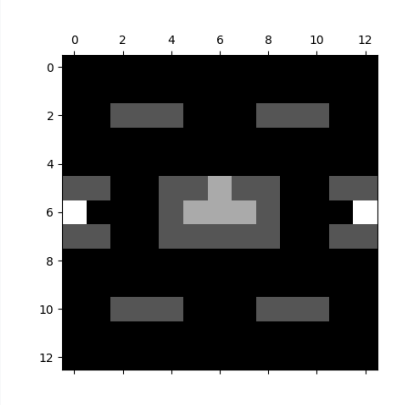
Médiocre : **score** $(\mu, x) \in [0, 2[$
Très mauvais : **score** $(\mu, x) \in [2, 2.5[$
Mauvais : **score** $(\mu, x) \in [2.5, 3[$
Moyen : **score** $(\mu, x) \in [3, 3.5[$
Bien : **score** $(\mu, x) \in [3.5, 4[$
Très bien : **score** $(\mu, x) \geqslant 4$

Mesure d'évaluation

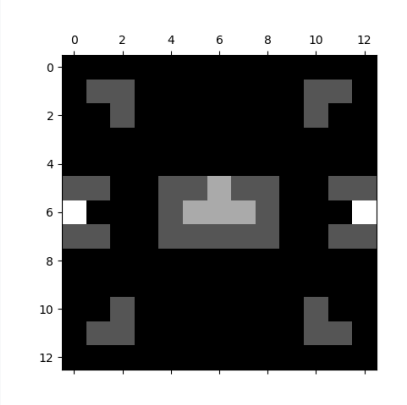
Images



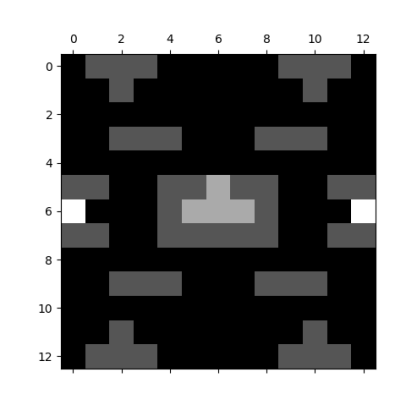
(a) Score: 0 [0]



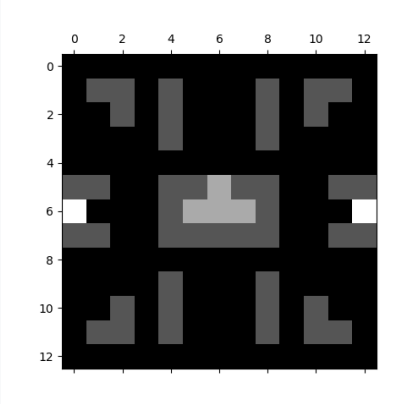
(b) Score: 2.50 [1]



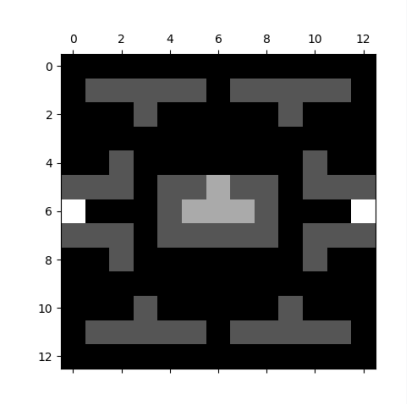
(c) Score: 2.94 [2]



(d) Score: 3.08 [3]



(e) Score: 3.65 [4]



(f) Score: 4.05 [5]

4. PACMAN

Visualisation : PACMAN

But et intégration

L'objectif de cette partie est de **visualiser dynamiquement** les labyrinthes générés à travers une version jouable du jeu **PACMAN**. Les éléments principaux affichés :

- Le labyrinthe issu du JSON (`grid`)
- Le personnage **Pac-Man** contrôlé par l'utilisateur
- Les **fantômes** et leur IA de déplacement
- Les **pellets** et **power pellets**
- Le **score** et les **vies restantes**

Cette simulation est réalisée avec la bibliothèque **Pygame**.

Affichage du score et des vies

Fonction

`draw_score(window, score, lives)`

Le score et les vies sont affichés en haut de la fenêtre à chaque frame.

- **Score** : incrémenté lors de la collecte de pellets (+10) ou de power pellets (+50)
- **Vies** : représentées par des cercles jaunes sous le score

```
def draw_score(window, score, lives):
    text = font_score.render(f"SCORE: {score}", True, WHITE)
    window.blit(text, (10, 10))
    for i in range(lives):
        pygame.draw.circle(window, (255,255,0), (10+i*25, 40), 8)
```

Figure: Affichage du score et des vies en jeu

Les Power Pellets

Activation du mode "Power"

Les **Power Pellets** sont positionnés dans les coins du labyrinthe. Lorsqu'un joueur en mange un :

- Pac-Man active le **mode Power** pendant quelques secondes
- Les fantômes deviennent **vulnérables**
- Le joueur peut les "manger" pour gagner des points supplémentaires

```
if is_power:
    pacman.activate_power(duration=200)
    for ghost in ghosts:
        ghost.set_vulnerable(duration=200)
```

Figure: Activation du mode Power lors d'un Power Pellet

Déplacement fluide de Pac-Man — Méthode `move()`

Le déplacement fluide est assuré par un système de direction et de contre-direction :

- `next_direction` : direction demandée par le joueur
- `direction` : direction actuelle, mise à jour si possible

```
def move(self):
    new_x = self.x + self.next_direction[0]
    new_y = self.y + self.next_direction[1]
    if self.can_move(new_x, new_y):
        self.direction = self.next_direction
    self.x += self.direction[0]
    self.y += self.direction[1]
```

Figure: Déplacement continu dans le labyrinthe

Fantômes — Déplacement et détection de collision

- Chaque fantôme conserve :
  - position actuelle (x, y),
  - position précédente $(x_{\text{prev}}, y_{\text{prev}})$,
  - direction et paramètres de sortie de la cage.
- À chaque frame :
  - on met à jour $(x_{\text{prev}}, y_{\text{prev}}) \leftarrow (x, y)$ avant de déplacer l'entité,
  - on calcule la nouvelle direction puis on avance d'une case.
- Détection de collision avec Pac-Man :
  1. **Collision directe :** si $(x_{ghost}, y_{ghost}) == (x_{pac}, y_{pac})$.
  2. **Collision croisée (échange de cases) :** si

$$(x_{ghost}^{\text{prev}}, y_{ghost}^{\text{prev}}) == (x_{pac}, y_{pac}) \quad \text{et} \quad (x_{ghost}, y_{ghost}) == (x_{pac}^{\text{prev}}, y_{pac}^{\text{prev}}).$$

5. GitHub et collaboration

Organisation

Gantt

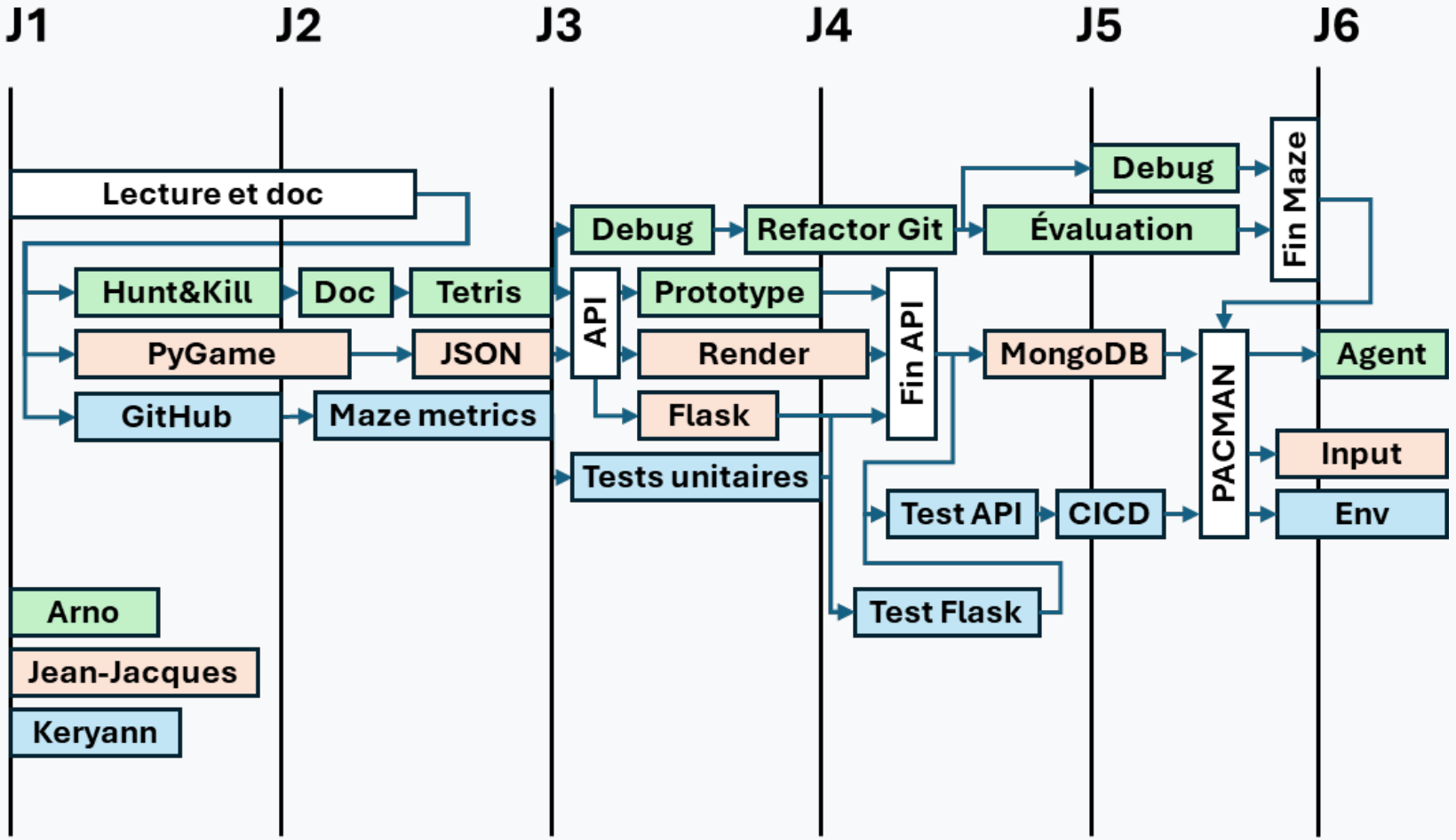


Figure: Diagramme de Gantt jusqu'à aujourd'hui

GitHub

Network Graph

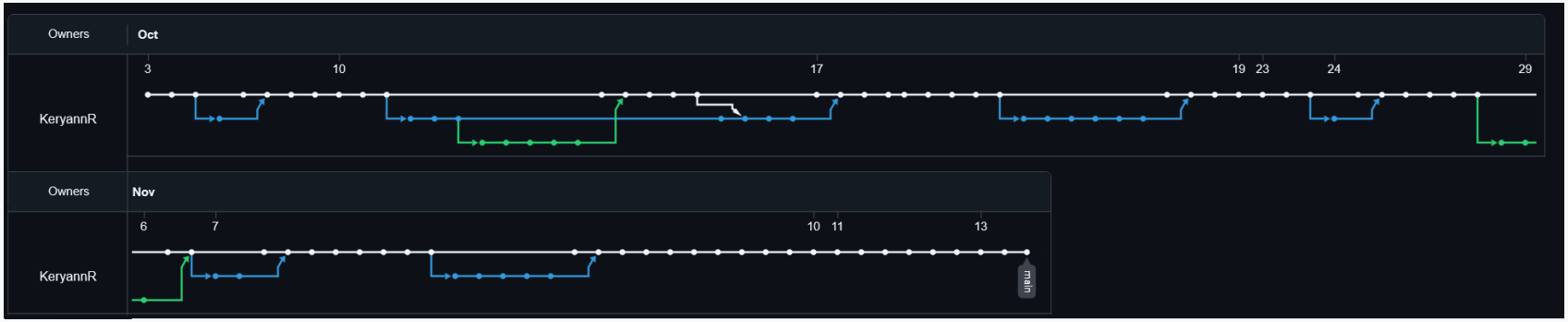


Figure: Network Graph jusqu'à aujourd'hui

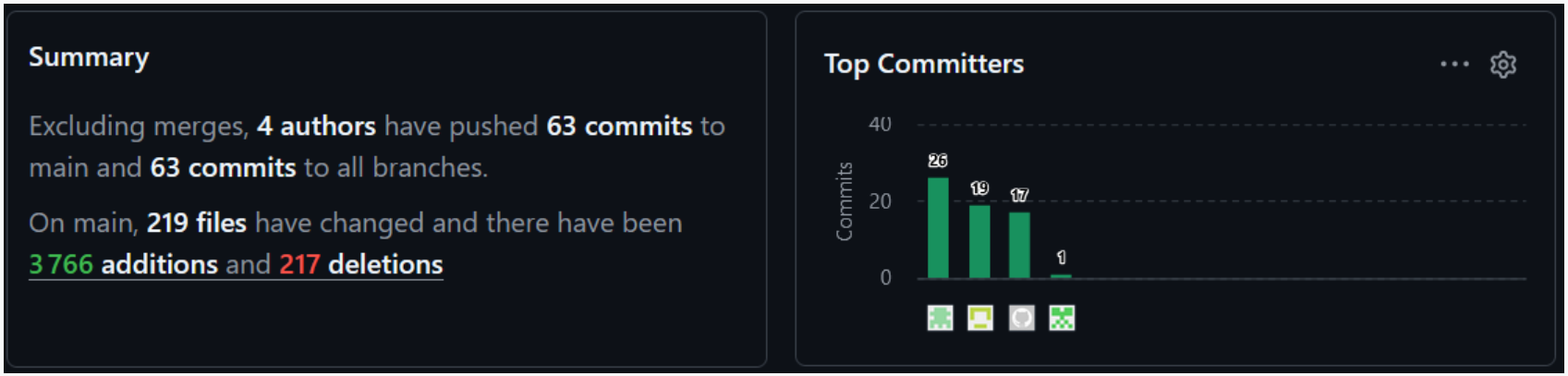


Figure: Commits sur le mois dernier

Merci de votre attention !