Université Paris Nanterre

Licence MIASHS deuxième année

Rapport de projet informatique

Mini Graffiti

Projet réalisé de novembre 2025 à janvier 2026

Membres du groupe

Mahjoub Omaïma – N° étudiant : 44004229
Cherfaoui Abdelkader – N° étudiant : 43014163

Dépôt GitHub : `https://github.com/Omaima05/mini-graffiti.git`

Table des matières

1 Introduction

Le projet Mini-Graffiti a été inspirée de la théorie des graphes et du logiciel Graffiti. L'objectif de ce projet est d'étudier automatiquement des relations possibles entre différents invariants de graphes sous la forme de conjectures. Ces conjectures sont ensuite testées sur un ensemble de graphes afin de vérifier si elles sont compatibles avec les données expérimentales ou si elles peuvent être réfutées par un contre-exemple.

Le logiciel Graffiti provient des travaux du mathématicien Siemion Fajtlowicz à la fin des années 1980. L'idée principale de Graffiti est d'utiliser un programme informatique pour générer automatiquement des conjectures en théorie des graphes à partir d'invariants numériques tels que le nombre de sommets, les degrés, le diamètre ou le rayon. Le logiciel explore un grand nombre de relations possibles entre ces invariants et propose des conjectures sous forme d'inégalités. Ces conjectures ne sont pas démontrées automatiquement, mais servent de point de départ aux recherches mathématiques. Certaines ont ensuite été étudiées, démontrées ou réfutées par des mathématiciens.

Mini-Graffiti est une version simplifiée et pédagogique de ce dernier. Il en conserve l'idée fondamental, mais de manière plus accessible. Le programme se base sur un ensemble fini de graphes, appelé zoo de graphes, sur un nombre limité d'invariants, ainsi que sur une validation expérimentale par tests automatiques et la recherche de contre-exemples. L'objectif n'est pas de produire de nouvelles théories mathématiques, mais de comprendre la démarche scientifique expérimentale consistant à formuler des hypothèses, à les tester et à analyser leurs limites.

Une particularité importante du projet est l'intégration d'une intelligence artificielle utilisée comme assistant de génération d'hypothèses. L'IA propose des conjectures, mais ne fournit aucune preuve et n'est jamais considérée comme fiable par défaut.

Le projet met ainsi en avant les limites de l'IA, en montrant que de nombreuses conjectures générées sont fausses ou approximatives et qu'elles nécessitent une validation rigoureuse par le programme. Cette approche montre l'importance de vérifier.

2 Présentation générale du programme

Notre programme a pour objectif de reproduire l'idée de Mini-Graffiti qui est de proposer des conjectures sur des graphes, puis les tester automatiquement sur un ensemble de graphes afin de voir si elles tiennent, si elles sont solides ou si elles sont fausses, et de leur trouver un contre-exemple si ces dernières sont fausse.

Le fonctionnement général est le suivant :

Tout d'abord, le programme construit des graphes, c'est-à-dire une liste de différents types de graphes par exemple : chemins, cycles, graphes complets, arbres, bipartis, graphes aléatoires. Le fichier `zoo.py` sert de base de test. Il permet d'avoir plusieurs formes de graphes pour vérifier les conjectures sur des exemples différents. Ensuite, le programme calcule plusieurs invariants pour chaque graphe du zoo. Ce sont des valeurs numériques qui décrivent le graphe, comme le nombre de sommets, le nombre d'arêtes, les degrés (minimum, maximum, moyen), la densité, le diamètre, le rayon et la complexité cyclomatique. Puis, une conjecture est générée sous forme d'une inégalité. Dans notre projet, cette conjecture est proposée par une intelligence artificielle. L'IA sert

uniquement à proposer une idée qui est une relation entre invariants, mais elle peut se tromper. C'est donc pour cela que le programme doit toujours vérifier la conjecture. Après cela, le programme vérifie la conjecture sur tous les graphes du zoo. Deux cas sont possibles, si la conjecture est vraie pour tous les graphes, on dit qu'elle est validée sur le zoo (cela ne veut pas dire que c'est une preuve mathématique, mais seulement que ça marche sur nos tests). Ou alors, si la conjecture est fausse pour au moins un graphe, alors elle est réfutée. Le graphe qui fait échouer la conjecture est appelé un contre-exemple. Ce contre- exemple est important car il montre concrètement que la conjecture n'est pas toujours vraie. Lorsqu'il y a un contre-exemple, le programme sauvegarde une image du graphe, ce qui permet de mieux voir sa structure car le graphe contre-exemple constitue une preuve expérimentale. Dans les deux cas (conjecture validée ou réfutée), le programme enregistre les résultats dans une base de données SQLite pour garder un historique : la conjecture, le résultat, la date, et éventuellement des informations sur le contre- exemple.

Enfin, le projet est organisé en plusieurs fichiers Python (modules) pour que ce soit plus clair :

— un fichier pour le zoo,
— un fichier pour les invariants,
— un fichier pour l'IA,
— un fichier pour les tests,
— un fichier pour la visualisation,
— un fichier pour la base de données,
— et un fichier principal qui lance tout.

Cette organisation rend le programme plus simple à comprendre et plus facile à améliorer.

3 Construction du zoo de graphes

Dans notre projet, le zoo de graphes est une partie essentielle. Il s'agit d'un ensemble de graphes que nous construisons au début du programme. Ces graphes servent ensuite à tester les conjectures proposées. L'idée est simple : une conjecture n'a d'intérêt que si on peut la vérifier sur plusieurs graphes différents. Le zoo joue donc le rôle de base de test expérimentale.

3.1 Rôle du zoo de graphes

Le zoo est nécessaire car notre programme doit vérifier si une conjecture est vraie ou fausse. Pour cela, il faut la tester sur un ensemble de graphes différents. Si on testait seulement sur un ou deux graphes, le résultat n'aurait aucune valeur car la conjecture pourrait être vraie sur ces exemples, mais fausse sur d'autres graphes. Donc il faut essayer sur le plus de graphs différents possibles. Le zoo permet donc de tester une conjecture sur plusieurs cas différents, de détecter rapidement si une conjecture est fausse et de trouver un contre-exemple si la conjecture ne tient pas. Ainsi, il rend le test automatique possible et utile.

Mais il possède une limite. Il contient un nombre fini de graphes. Cela veut dire que si une conjecture est validée sur le zoo, cela ne signifie pas qu'elle est forcément vraie

mais seulement qu'elle est vraie sur les graphes que nous avons testés.

Par consequent, une conjecture validée est seulement validée expérimentalement, ce n'est pas une preuve mathématique. Une conjecture peut très bien être vraie sur tous les graphes du zoo mais devenir fausse sur un graphe plus grand ou d'une autre forme qui n'est pas dans notre zoo. Cela montre que nous comprenons bien la démarche scientifique du projet.

3.2 Types de graphes utilisés

Zoo doit contenir des graghes de formes différentes afin qu'il soit intérréssant. Nous avons donc choisi plusieurs familles de graphes. Chaque famille apporte des caractéristiques particulières, ce qui rend le test plus solide. Voila les différents types de graphes qu'on a utilisé :

Un graphe chemin est une structure très simple : les sommets sont reliés en ligne. Ces graphes ont peu d'arêtes, et un diamètre souvent grand. Ils sont utiles pour tester des conjectures liées aux distances.

Les cycles ressemblent aux chemins, mais avec une arête supplémentaire qui ferme la boucle. Ils permettent de tester des conjectures sur la présence de cycles et sur la structure régulière du graphe.

Dans un graphe complet, chaque sommet est relié à tous les autres. Ce sont des graphes très denses, avec des degrés élevés et un diamètre très petit (souvent 1). Ils sont très utiles pour tester des conjectures liées à la densité ou aux degrés.

Les arbres sont des graphes connexes sans cycles. Ils sont importants car ils représentent des graphes "minimaux" en nombre d'arêtes tout en restant connexes. Ils permettent de tester des conjectures sur la structure et la complexité cyclomatique.

Les graphes bipartis sont des graphes dont les sommets peuvent être séparés en deux groupes, avec des arêtes uniquement entre les deux groupes. Ils ont des propriétés particulières (pas de cycle impair, structure différente). Ils ajoutent de la diversité.

Les graphes aléatoires servent à créer de la diversité et à éviter un zoo trop "prévisible". Ils permettent de générer des formes variées, parfois proches de cas réels. Ils sont utiles pour tester des conjectures sur des graphes moins réguliers.

Grâce à ces différents choix, le zoo contient des graphes très variés : certains sont très denses, d'autres beaucoup moins, certains ont beaucoup de cycles tandis que d'autres n'en ont pas du tout. Cette variété est importante car elle permet de tester le programme sur des cas très variés et donc de vérifier qu'il fonctionne correctement dans toutes les situations.

3.3 Choix techniques

Nous avons choisi de tester principalement des graphes connexes. Des graphes connexes sont des graphes où tous les sommets sont reliés entre eux. On a fait ce

choix car certains invariants comme le diamètre et le rayon sont plus simples à utiliser sur des graphes connexes. Si le graphe n'est pas connexe, ces valeurs peuvent devenir difficiles à interpréter ou nécessiter une gestion spécifique. Cela, nous permet de simplifier l'analyse et d'obtenir des résultats cohérents.

Nous avons limité la taille des graphes à n<=10 pour diverses raisons. Une des raisons principales est que pour le programme reste rapide et exécutable sur un ordinateur classique. Une autre raison est que pour certains calculs (comme le diamètre) deviennent plus lourds sur de grands graphes. De plus, le but du projet est surtout de comprendre la démarche expérimentale et non traiter des graphes géants. Cette limite permet donc d'obtenir des tests rapides tout en gardant une diversité suffisante.

Un autre choix important est la reproductibilité. Pour les graphes aléatoires, nous utilisons une seed (graine) fixe. Cela signifie que si on relance le programme, on obtient les mêmes graphes aléatoires et les mêmes résultats. Cela est important dans un projet expérimental car cela permet de refaire exactement la même expérience, de vérifier et comparer les résultats et d'éviter des résultats qui changent à chaque exécution. Ce point est important dans une démarche expérimentale, car il garantit que les résultats ne dépendent pas du hasard mais bien des choix effectués dans le programme.

En résumé, le zoo de graphes est une base de test indispensable. Il est construit de manière variée avec plusieurs familles de graphes, limité en taille pour garder de bonnes performances, et rendu reproductible grâce à une seed fixe. Cette partie est essentielle, car la qualité du zoo influence directement la qualité des tests de conjectures et la pertinence des contre-exemples trouvés

4 Invariants de graphes utilisés

Les invariants de graphes jouent un rôle central dans le projet. Un invariant est une valeur numérique qui décrit une propriété d'un graphe et qui ne dépend pas de la façon dont le graphe est dessiné. Ces invariants permettent de comparer différents graphes et de formuler des conjectures sous forme de relations mathématiques.

Le programme calcule automatiquement plusieurs invariants pour chaque graphe du zoo. Ces valeurs sont ensuite utilisées par l'intelligence artificielle pour proposer des conjectures, puis par le programme pour vérifier si ces conjectures sont vraies ou fausses.

4.1 Les différents invariants utilisés

Le nombre de sommets correspond au nombre de nœuds du graphe. C'est l'invariant le plus simple, mais l'un des plus importants, car il donne la taille du graphe. Plusieurs de propriétés d'un graphe dépendent de ce nombre.

Le nombre d'arêtes correspond au nombre de connexions qui existent entre les sommets. Il permet de savoir si un graphe est peu connecté ou au contraire très dense. Cet invariant est souvent comparé au nombre de sommets dans les conjectures.

Le degré d'un sommet correspond au nombre d'arêtes qui lui sont reliées. Le degré minimum donne le sommet le moins connecté du graphe. Le degré maximum donne le sommet le plus connecté. Le degré moyen donne une idée globale du niveau de connexion du graphe. Ces valeurs sont très utiles pour décrire la structure du graphe et sont souvent utilisées dans les conjectures sur la connectivité.

La densité mesure le rapport entre le nombre d'arêtes présentes et le nombre maximum d'arêtes possibles. Elle permet de comparer des graphes de tailles différentes. Un graphe complet a une densité proche de 1, tandis qu'un graphe très peu connecté a une densité proche de 0.

Le diamètre correspond à la plus grande distance entre deux sommets du graphe. Il mesure la taille du graphe du point de vue des distances. Les graphes très étendus, comme les chemins, ont un grand diamètre, tandis que les graphes très denses, comme les graphes complets, ont un diamètre très petit.

Le rayon est lié au diamètre. Il correspond à la plus petite distance maximale entre un sommet et tous les autres sommets. Il permet d'identifier la précense d'un sommet "central" dans le graphe.

La complexité cyclomatique mesure la présence de cycles dans le graphe. Elle permet de faire la différence entre les graphes sans cycle (comme les arbres) et les graphes plus complexes contenant plusieurs cycles.

4.2 Rôle des invariants dans les conjectures

Les invariants sont utilisés pour formuler les conjectures. Une conjecture est généralement une relation mathématique reliant plusieurs invariants. Par exemple, une conjecture peut comparer le diamètre et le rayon, ou relier le nombre d'arêtes au nombre de sommets et aux degrés.

Les invariants ont donc 2 rôles qui sont permettre de décrire quantitativement les graphes du zoo et servir de variables dans les conjectures générées et testées. En combinant plusieurs invariants, le programme peut proposer des conjectures plus ou moins complexes. Certaines sont rapidement réfutées par un contre-exemple, alors que d'autres restent valides sur l'ensemble du zoo. L'utilisation des invariants montre que le projet ne se limite pas à manipuler des graphes de manière générale, mais cherche à analyser et comparer leurs propriétés de façon rigoureuse et structurée.

C'est donc pour cela que le choix et l'utilisation de ces invariants sont essentiels pour comprendre le fonctionnement du programme et vérifier la validité des conjectures testées.

5 Génération et test des conjectures

Cette partie est la partie centrale du projet. Elle explique comment les conjectures sont créées et comment elles sont testées automatiquement sur les graphes du zoo.

5.1 Forme des conjectures

Une conjecture correspond à une hypothèse sur les graphes. Elle est écrite sous la forme d'une inégalité entre plusieurs invariants, par exemple le diamètre, le rayon ou le nombre de sommets. Cela permet de garder des conjectures simples à comprendre et faciles à tester par le programme.

Par exemple, une conjecture peut dire que le diamètre d'un graphe est toujours inférieur à une certaine expression utilisant d'autres invariants. Ces conjectures sont proposées automatiquement, mais elles ne sont pas forcément vraies. Elles servent de point de départ pour les tests.

5.2 Test automatique des conjectures

Une fois la conjecture générée, le programme la teste sur tous les graphes du zoo. Pour chaque graphe, les invariants sont calculés, puis la conjecture est évaluée avec ces valeurs. Si la conjecture fonctionne pour tous les graphes testés, on considère qu'elle est validée. Cela ne signifie pas qu'elle est prouvée, mais simplement qu'aucun contre-exemple n'a été trouvé parmi les graphes utilisés. Mais si la conjecture est fausse pour au moins un graphe, elle est directement réfutée. Alors, le programme s'arrête et passe à l'analyse de ce graphe.

5.3 Détection des contre-exemples

Un contre-exemple est un graphe pour lequel la conjecture ne fonctionne pas. Il permet de montrer que la conjecture est fausse en général. Dans notre projet, le graphe contre-exemple est automatiquement identifié et sauvegardé. Il est représenté sous forme d'image, ce qui permet de mieux visualiser sa structure et de comprendre pourquoi la conjecture échoue.

Les contre-exemples sont très importants, car ils montrent les limites des conjectures proposées. Même si une conjecture fonctionne pour beaucoup de graphes, un seul contre-exemple suffit à la rendre incorrecte.

Cette partie montre bien le concept de Mini-Graffiti qui consiste de formuler des hypothèses, les tester de manière systématique et utiliser les contre-exemples pour analyser et comprendre les erreurs, plutôt que de supposer qu'une conjecture est vraie.

6 Visualisation et conservation des résultats

Il est important de tester des conjectures, mais aussi de comprendre les résultats obtenus et de les conserver. C'est pour cela que nous avons choisi de visualiser certains graphes et d'enregistrer les résultats dans une base de données.

6.1 Visualisation graphique des contre-exemples

Nous avons choisi de visualiser uniquement les contre-exemples, les graphes pour lesquels une conjecture est fausse. Quand une conjecture est validée sur le zoo, il n'y a pas de graphe particulier à montrer, car tous les graphes respectent la règle. Afficher un

graphe dans ce cas n'apporte pas d'information utile. Cependant, lorsqu'une conjecture est réfutée, il existe un graphe précis qui ne vérifie pas la relation proposée. Visualiser ce graphe nous permet de mieux comprendre pourquoi la conjecture ne fonctionne pas. L'image rend le résultat plus facile à expliquer.

Quand un contre-exemple est trouvé, le programme crée automatiquement une image du graphe correspondant et la sauvegarde dans un dossier nommé rapport/image. Cela permet de garder une preuve visuelle des erreurs trouvées, sans avoir besoin de relancer le programme. On retrouvera ces images dans l'annexe.

6.2 Archivage des données via SQLite

Pour ne pas perdre les résultats, nous avons utilisé une base de données SQLite. Elle conserve toutes les conjectures testées et leurs résultats, même après la fin du programme. Ce qui rend le projet plus organisé et plus fiable.

Pour chaque conjecture, la base de données enregistre la conjecture testée, le résultat si elle est validée ou réfutée, la date de test, et lorsqu'il y a un contre-exemple, le lien vers l'image correspondant à sa conjecture. Le stockage dans une base de données permet d'avoir une vue d'ensemble des résultats. On peut par exemple savoir combien de conjectures ont été validées ou réfutées, et observer le comportement général des conjectures proposées par l'IA. Cela rend le projet plus analytique et en lien avec l'informatique et la gestion.

Cette partie montre que notre projet ne s'arrête pas au test des conjectures, mais cherche aussi à rendre les résultats compréhensibles et exploitables.

7 Utilisation de l'intelligence artificielle

L'intelligence artificielle a joué un rôle important dans l'aboutissement de notre projet. Elle a été utilisée tout au long du développement comme un outil d'assistance, aussi bien pour la conception que pour la résolution de problèmes techniques. Elle permet d'automatiser la génération des conjectures, tout en montrant clairement que l'IA ne peut pas remplacer une analyse rigoureuse. Dans notre projet, l'IA est utilisée de manière encadrée et critique, ce qui correspond à l'esprit de Mini-Graffiti.

7.1 Rôle de l'IA

L'intelligence artificielle a été utilisée principalement comme un outil d'assistance et comme un générateur d'hypothèses. Elle a joué un rôle important tout au long du développement, mais sans jamais remplacer le raisonnement humain ni les tests automatiques. D'une part, l'IA a servi de générateur de conjectures. Son rôle est de proposer automatiquement des relations possibles entre plusieurs invariants de graphes, sous forme d'inégalités mathématiques. Ces conjectures servent uniquement de point de départ pour les tests réalisés par le programme.

Il est important de préciser que l'IA n'est pas considérée comme fiable par défaut. Les conjectures qu'elle propose peuvent être vraies, fausses ou approximatives. Le programme ne lui fait donc jamais confiance directement : chaque conjecture générée est

systématiquement testée sur le zoo de graphes.

D'autre part, l'IA a également été utilisée comme assistant de développement. Elle nous a aidés lorsque nous rencontrions des difficultés dans l'écriture du code, dans la compréhension de certaines erreurs ou lors de l'exécution du programme. Par exemple, elle nous a permis de mieux comprendre l'origine de bugs, de problèmes de lenteur ou d'erreurs liées à l'utilisation de bibliothèques comme NetworkX ou Matplotlib.

L'IA a aussi joué un rôle de conseiller et d'outil d'orientation. Elle nous a aidés à structurer notre démarche, à organiser le projet en plusieurs modules clairs et à faire des choix techniques adaptés, comme la gestion des erreurs, l'optimisation du code ou la mise en place d'un fonctionnement reproductible. Enfin, elle a été utilisée comme un outil de formation, nous aidant à mieux comprendre les concepts du projet (graphes, invariants, conjectures, contre-exemples), et pas seulement à produire du code.

Ainsi, l'IA a surtout servi d'assistant, capable de proposer des idées, d'aider à résoudre des problèmes techniques et d'accompagner notre apprentissage.

7.2 Fonctionnement de l'IA utilisée

Pour intégrer l'IA dans le programme, nous avons utilisé Ollama, qui permet d'exécuter des modèles de langage directement en local sur l'ordinateur. Cela évite l'utilisation de services en ligne et permet de mieux contrôler les échanges avec le modèle. Le modèle utilisé est Mistral, un modèle de langage capable de produire des textes courts et structurés. Dans notre cas, il est utilisé pour générer des conjectures sous forme d'inégalités mathématiques simples.

Le fonctionnement est le suivant. Tout d'abord, le programme construit un prompt, c'est-à-dire un texte de consignes envoyé à l'IA . Ensuite, ce prompt précise les invariants autorisés, le format attendu qui est une seule inégalité, et parfois des exemples à éviter. Enfin, l'IA génère une réponse à partir de ce prompt. Comme l'IA peut parfois donner des réponses fausses ou mal présenté, le programme les nettoie et les vérifie. Cela permet de supprimer les caractères inutiles, de vérifier que la réponse correspond bien à une expression exploitable, et d'éviter que le programme ne plante lors du test.

7.3 Limites de l'IA

L'utilisation de l'IA nous a permis de voir plusieurs limites importantes. Tout d'abord, l'IA génère souvent des conjectures fausses. Cela montre qu'elle ne "comprend" pas réellement la théorie des graphes, mais qu'elle se base uniquement sur des modèles statistiques. Ces conjectures fausses sont toutefois intéressantes, car elles permettent de produire rapidement des contre-exemples.

Ensuite, l'IA a tendance à proposer des conjectures répétitives ou trop simples, en particulier lorsqu'elle cherche des relations sûres. Pour limiter ce problème, nous avons essayé de varier les prompts et d'indiquer les conjectures déjà testées, mais ce comportement reste une limite du modèle.

Enfin, ces limites montrent la nécessité du test automatique. Sans vérification systématique sur le zoo de graphes, il serait impossible de savoir si une conjecture proposée par l'IA est correcte. Le projet montre donc clairement que l'IA ne doit pas être utilisée seule, mais toujours accompagnée de mécanismes de contrôle et de validation.

Cela montre que l'IA est utilisée de manière réfléchie. Elle assiste le projet en générant des idées, mais elle ne remplace jamais l'analyse automatique et critique des résultats.

8 Difficultés rencontrées et solutions

Le développement de ce projet ne s'est pas fait de manière linéaire. Nous avons rencontré plusieurs difficultés, aussi bien techniques que conceptuelles. Ces problèmes nous ont obligés à tester différentes solutions, à corriger notre code et à mieux comprendre le fonctionnement global du programme. Cette section présente les principaux blocages rencontrés et les solutions mises en place.

8.1 Problèmes techniques

On a constaté une difficulté importante lors de l'exécution du programme en mode multi-conjectures. Le programme était censé tester jusqu'à 20 tentatives, numérotées de 1 à 20. Cependant, après plusieurs essais, nous avons constaté que l'exécution s'arrêtait souvent avant la fin, généralement autour de la 3 ou 4 tentative.
Le programme semblait alors rester bloqué, sans afficher de message d'erreur clair. Cela donnait l'impression que le programme "réfléchissait" indéfiniment, sans produire de résultat supplémentaire.
Ce problème était à cause du temps de réponse de l'IA et à des appels bloquants. Pour améliorer cela, nous avons augmenté les délais d'attente (timeout). Nous avons aussi mieux géré les erreurs réseau et accepté le fait qu'un modèle local peut parfois ralentir ou bloquer, ce qui constitue une limite du projet. Cette difficulté montre que l'utilisation d'une IA locale peut entraîner des problèmes de stabilité lors d'exécutions longues.

Un autre problème qu'on a rencontré était la répétition des conjectures générées par l'IA. Dans plusieurs exécutions, l'IA proposait presque toujours la même conjecture, utilisant les mêmes invariants. Lorsque cette conjecture était fausse, le programme trouvait logiquement le même contre-exemple. Cela entraînait donc que les résultats devenaient peu variés et que l'image affichée était souvent la même, ce qui donnait l'impression que le programme ne fonctionnait pas correctement.
Pour remédier à cela, nous avons amélioré le prompt envoyé à l'IA afin de l'encourager à varier les invariants utilisés et nous avons ajouté une mémoire des conjectures déjà testées pour limiter les répétitions. Enfin, nous avons compris que ce comportement est une limite du modèle lui-même, qui a tendance à proposer des réponses "sûres" et répétitives.

Au début du projet, nous avions tenté d'utiliser une API en ligne Gemini pour générer les conjectures. Cependant, cette API présentait plusieurs limitations : restrictions d'accès, clés qui expirent, erreurs fréquentes et manque de stabilité due aux réseaux. Il était difficile de l'utiliser de manière répétée et fiable.
Nous avons remplacé cette API par une solution locale en utilisant Ollama avec le

modèle Mistral. Cela nous a permis d'éviter les limites imposées par les API en ligne, de mieux contrôler les échanges avec l'IA et de travailler hors ligne. Ce choix a rendu le projet plus stable, même si elle a introduit d'autres contraintes comme le temps de calcul.

Nous avons également rencontré des erreurs liées aux bibliothèques Python, notamment NetworkX. Certaines fonctions avaient changé selon les versions, ce qui provoquait des erreurs à l'exécution. Nous avons alors consulté la documentation officielle. Ensuite, nous avons remplacé les fonctions obsolètes par leurs équivalents récents. Enfin, nous avons vérifié la compatibilité avec notre version de Python.

Un autre point important a été le choix du langage. Au début on voulait utiliser le langage C, mais on a remarqué qu'il était peu adapté à ce projet. La gestion manuelle de la mémoire, l'absence de bibliothèques simples pour les graphes et la difficulté d'intégrer une IA auraient rendu le projet beaucoup plus complexe.
Donc, nous avons choisi d'utiliser Python, car il permet une écriture plus simple et plus lisible, l'utilisation de bibliothèques adaptées (NetworkX, Matplotlib, SQLite), et surtout une intégration plus facile de l'intelligence artificielle.

8.2 Problèmes conceptuels

Une difficulté conceptuelle importante a été de comprendre la différence entre une preuve mathématique et une validation expérimentale. Une conjecture validée sur le zoo n'est pas forcément vraie en général, car le zoo est un ensemble fini de graphes. Nous avons clairement distingué ces deux notions dans notre analyse. Nous avons mis en avant le rôle central des contre-exemples, qui suffisent à réfuter une conjecture.

Ces difficultés nous ont permis de mieux comprendre les limites techniques et conceptuelles du projet. Elles nous ont aussi appris à analyser des bugs réels, améliorer la stabilité du programme, adopter une démarche expérimentale rigoureuse et utiliser l'IA de manière critique. Ce qui montre que le projet n'est pas seulement un code fonctionnel, mais aussi un travail de réflexion, d'adaptation et d'apprentissage progressif.

9 Synthèse et analyse des résultats

À la fin des différentes exécutions du programme, nous avons obtenu un ensemble de résultats qui permettent d'avoir une vision globale du comportement des conjectures générées et testées.

Lors de chaque session, le programme a testé un certain nombre de conjectures par exemple 20 par session d'exécution. Parmi ces conjectures, certaines ont été validées sur le zoo de graphes, tandis que d'autres ont été réfutées par la présence d'un contre-exemple. De manière générale, nous avons observé qu'une grande partie des conjectures est fausse et que seules quelques conjectures restent valides sur l'ensemble du zoo. Ce résultat n'est pas surprenant car l'intelligence artificielle génère des conjectures sans preuve et sans connaissance exacte des propriétés mathématiques des graphes. Elle propose donc souvent des relations incorrectes ou trop générales. Le rôle du programme est justement de tester systématiquement ces conjectures et de détecter rapidement celles

qui ne sont pas valides grâce à des tests automatiques. Les conjectures réfutées sont particulièrement intéressantes, car elles produisent des contre-exemples concrets. Ces graphes montrent précisément pourquoi la conjecture ne fonctionne pas. À l'inverse, les conjectures validées doivent être interprétées avec prudence car elles sont compatibles avec les graphes testés, mais elles ne sont pas prouvées.

L'analyse globale des résultats montre donc que le programme remplit bien son objectif qu'il ne fait pas confiance à l'IA, il teste systématiquement chaque conjecture et il met en évidence les erreurs à l'aide de contre-exemples. Cela confirme que l'approche expérimentale de Mini-Graffiti est efficace pour explorer des hypothèses et comprendre leurs limites.

10 Conclusion

Le projet Mini-Graffiti nous a permis de mettre en place une démarche expérimentale complète autour de la théorie des graphes. Nous avons développé un programme capable de générer automatiquement des conjectures, de les tester sur un ensemble de graphes et de produire des contre-exemples lorsque ces conjectures sont fausses.

Le projet regroupe plusieurs aspects importants de l'informatique comme la programmation, l'utilisation de bibliothèques spécialisées, la gestion de données, la visualisation et l'intégration d'une intelligence artificielle. Il met également en avant une démarche scientifique basée sur l'expérimentation et l'analyse critique.

Tout au long de ce projet, nous avons appris à manipuler des graphes et leurs invariants avec Python, structurer un programme en plusieurs modules clairs, gérer des erreurs techniques et des problèmes de performance, utiliser une intelligence artificielle de manière critique et encadrée et distinguer une validation expérimentale d'une preuve mathématique. Mais, nous avons aussi appris que le développement d'un projet ne se contente pas à écrire du code, mais il faut aussi faire des choix techniques, des tests, des corrections et une réflexion constante.

Cependant, le projet présente certaines limites. La principale concerne le zoo de graphes utilisé, qui est de taille finie et composé majoritairement de graphes de petite taille. Ainsi, une conjecture validée sur ce zoo peut être fausse pour des graphes plus grands ou pour des structures qui ne sont pas représentées dans l'ensemble testé. Les résultats obtenus doivent donc être interprétés avec prudence et ne peuvent pas être considérés comme des preuves mathématiques.

Pour prolonger ce travail, on pourrait envisagées plusieurs améliorations pour prolonger ce travail comme augmenter la taille et la diversité du zoo de graphes, ajouter de nouveaux invariants pour enrichir les conjectures, améliorer la génération des conjectures afin d'obtenir des hypothèses plus variées, analyser statistiquement les résultats sur un plus grand nombre de conjectures et optimiser encore les performances pour tester davantage de graphes.

A Annexes

A.1 Capture d'exécution du terminal

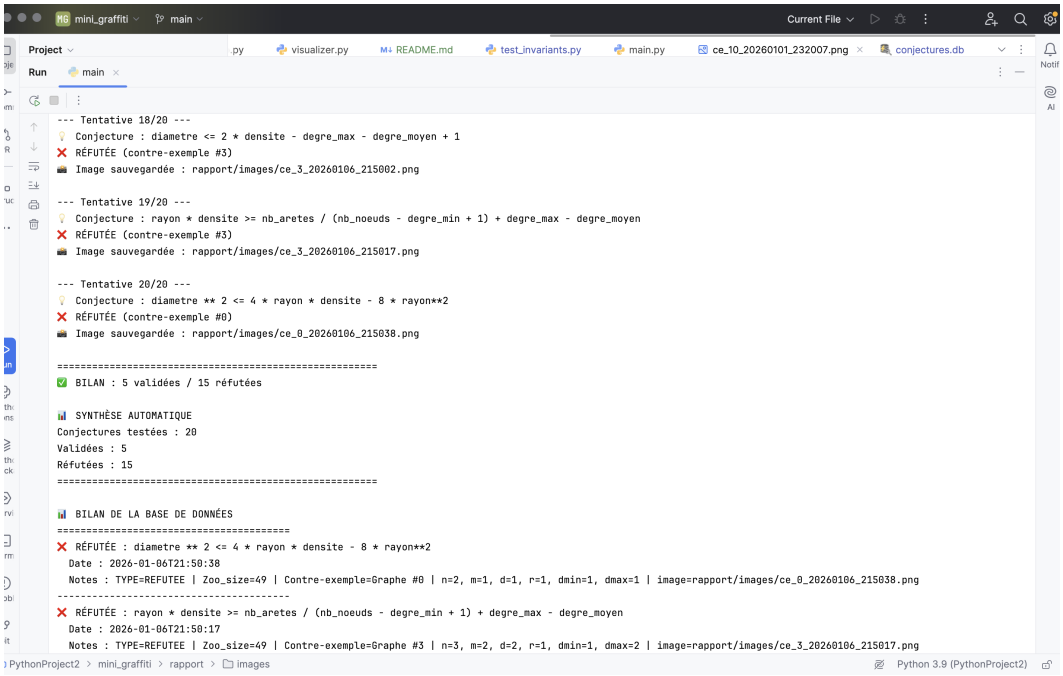

FIGURE 1 – Exemple d'exécution du programme

La figure suivante montre un exemple d'exécution du programme. À chaque tentative, une conjecture est générée puis testée automatiquement sur le zoo de graphes. Lorsqu'une conjecture est fausse, elle est réfutée et un graphe contre-exemple est identifié et sauvegardé sous forme d'image. À la fin de l'exécution, un bilan récapitulatif indique le nombre de conjectures validées et réfutées, ainsi qu'une synthèse automatique des résultats.

A.2 Image d'un graphe contre-exemple

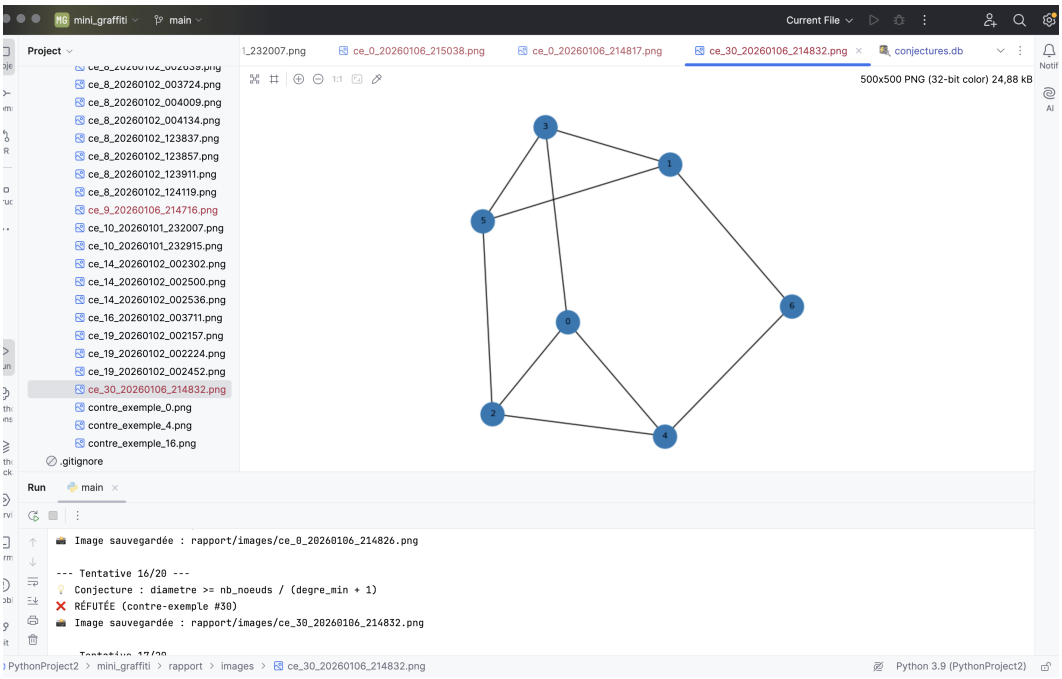

FIGURE 2 – Réfutation d'une conjecture et affichage du contre-exemple

Cette figure représente un graphe contre-exemple pour une conjecture générée par l'IA. Ce graphe montre concrètement que la conjecture ne se vérifie pas pour tous les graphes du zoo.

A.3 Visualisation de la base SQLite

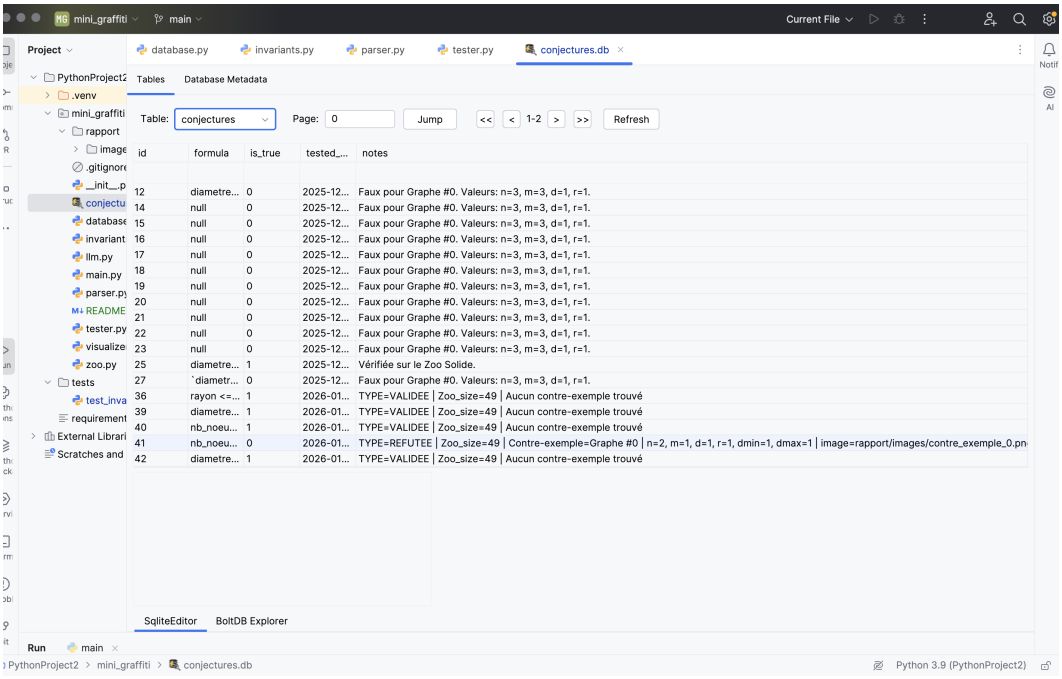

FIGURE 3 – Historique des conjectures sauvegardées en base SQLite

Cette capture montre un extrait de la base de données SQLite utilisée pour stocker les résultats des conjectures testées.

Remerciements

Nous souhaitons exprimer notre reconnaissance pour le travail d'équipe que nous avons mené tout au long de ce projet.

Un grand merci à nous-mêmes, pour notre implication, notre sérieux et notre bonne humeur. Chacune d'entre nous a contribué de manière essentielle à la réussite de ce travail, que ce soit dans la collecte des données, le traitement des articles ou la mise en place des visualisations.

Notre collaboration a été non seulement efficace, mais aussi agréable, et a permis de surmonter ensemble les différentes difficultés rencontrées.

Enfin, nous sommes fières du chemin parcouru et de ce que nous avons pu apprendre en équipe à travers ce projet.

Omaïma Mahjoub, Cherfaoui Abdelkader