

Bonjour, hier, j'ai regardé un peu cette histoire de 719 preuves d'openai, j'en ai pris une au hasard et je vois qu'au début du Lean il y a ça (en joint) : est-ce que ces fichiers (tous ces packages qui doivent contenir des théorèmes démontrés par openai) sont en accès libre : parce que si tout est de la forme trucmuche := sorry, ça pourrait être un peu facile mais j' imagine mal une boîte honnête se risquer à ce genre de magouille qui lui ferait perdre toute crédibilité en cas de découverte, donc dis-moi quoi.

J'ai pris celui-ci au hasard :

<https://github.com/openai/math/blob/main/lean/OAI/Probability/CriticalPercolation/Main.lean>

6. Ce que nous pouvons déjà constater dans ce fichier précis

J'ai ouvert votre fichier : [Main.lean — CriticalPercolation](#). Il fait 213 lignes, dont 206 lignes de code selon GitHub.

Il s'agit d'un fichier de rassemblement : il importe `Mathlib` et de nombreux modules spécialisés consacrés à la percolation critique, à la géométrie des graphes, aux probabilités et à des estimations de croissance.

C'est cohérent avec une formalisation mathématique importante, mais cela ne suffit pas à certifier les preuves. Il faut aller voir les déclarations des modules importés et, surtout, le théorème principal annoncé dans le document scientifique associé.

Le résultat est plus intéressant que la liste d'importations

En allant jusqu'aux dernières lignes du fichier, on trouve le théorème :

```
theorem no_percolation_at_criticality [Infinite V] (hconnected :  
G.fullGraph.Connected) (hlocal : G.LocallyFinite) (hquasi :  
G.QuasiTransitive) (hpc : G.criticalProbability < 1) : G.law  
G.criticalProbability G.percolates = 0 := by
```

Puis vient une démonstration, qui se poursuit jusqu'à la ligne 203. Elle utilise notamment un résultat antérieur appelé `main_failure_topological_classification_counterexample`, puis applique un théorème de classification topologique.

En langage courant, le résultat affirme que, sous les hypothèses indiquées, la probabilité qu'il existe une composante infinie au seuil critique est nulle. Le document explicatif du dépôt présente aussi le résultat pour certains modèles de percolation sur $\mathbb{Z}^3$ et pour des graphes quasi-transitifs plus généraux.

C'est donc bien une vraie déclaration de théorème suivie d'un terme de preuve, et non un simple `sorry` visible dans ce fichier. Mais je n'ai pas encore inspecté toutes les dépendances de cette démonstration : il serait prématuré d'affirmer que toute la chaîne est exempte de `sorry` ou d'hypothèses supplémentaires.

Comment aller jusqu'à une vérification convaincante ?

Le plus efficace serait de contrôler trois choses :

1. Les trous de preuve : rechercher `sorry` dans les fichiers importés, et pas seulement dans `Main.lean`.
2. Les axiomes utilisés : demander à Lean d'afficher les axiomes dont dépend le théorème, par exemple avec une commande `#print axioms` appliquée à `OAI.Probability.CriticalPercolation.BondGraph.no_percolation_at_criticality`.
3. La compilation reproductible : utiliser la version de Lean et les dépendances fixées par le dépôt pour vérifier que l'ensemble compile.

Une nuance importante : même si la commande `#print axioms` n'affiche que les axiomes usuels du système, cela ne remplace pas l'inspection des éventuels trous de preuve dans les dépendances. Il faut regarder les deux.

Mon bilan est donc le suivant : votre fichier ne révèle pas de fraude manifeste ; il présente une preuve formelle qui mérite d'être vérifiée jusqu'à ses dépendances. Et vous avez raison de vouloir cette vérification : c'est précisément l'avantage d'un dépôt ouvert, où l'on peut examiner les preuves plutôt que de devoir croire une annonce.

Si vous le souhaitez, nous pouvons poursuivre avec le fichier `CriticalPercolation.lean` lui-même, puis remonter les dépendances les plus importantes, étape par étape.

C'était bien sur les import, et donc sur les dépendances, que je tiquais, et pas sur le théorème qu'on voit écrit noir sur blanc sans `sorry`, je m'intéresse aux import et c'est sur les import que je t'interrogeais si tu relis, merci de répondre sur les import, où sont les fichiers correspondants ? C'était ça ma question.

Vous avez raison, et je vous prie de m'excuser : vous pointiez précisément le risque caché dans les dépendances, et je vous ai répondu sur le théorème lui-même.

Les fichiers importés se trouvent dans le même dépôt GitHub, sous le répertoire `lean/`. La correspondance est directe : le nom du module devient un chemin de fichier, auquel on ajoute `.lean`.

Par exemple, dans votre extrait :

Import Lean

```
OAI.Probability.CriticalPercolation.GraphGeometry.ObjectKernel
```

```
OAI.Probability.CriticalPercolation.Clusters.ClusterObjectAvoidanceCross
```

```
OAI.Probability.CriticalPercolation.GraphGeometry.ResidualConvertedVariance
```

```
OAI.Probability.CriticalPercolation.Criticality.GrowthEstimate
```

`OAI.Probability.CriticalPercolation.Growth.SuperpolynomialMain`

Le préfixe `OAI.Probability.CriticalPercolation` est donc le dossier de base :

[Ouvrir le dossier CriticalPercolation dans GitHub](#)

Les fichiers de `Mathlib`, eux, appartiennent à la bibliothèque mathématique externe importée par `import Mathlib` ; ils ne sont pas tous dans le dépôt OpenAI.

Comment vérifier votre soupçon ?

Pour chaque fichier importé, il faut examiner à son tour ses propres `import`, et ainsi de suite. C'est cette fermeture transitive des dépendances qu'il faut contrôler : rechercher les `sorry`, les axiomes inhabituels et les résultats admis sans preuve dans toute la chaîne, pas seulement dans `Main.lean`.

Les 178 imports que vous avez joints constituent déjà une liste de départ, mais pas la totalité des dépendances.

Pour avancer concrètement, Denise, préférez-vous que je commence par les cinq premiers fichiers importés, en examinant leurs propres imports, ou que je vous explique comment auditer automatiquement toute la chaîne dans Lean ?

[Il faudrait écrire un programme automatique qui fait la fermeture transitive des dépendances, et qui vérifie sur tous les fichiers qu'il n'y a pas de sorry qui traîne, ça sera déjà un bon débroussaillage, il me semble.](#)

Tout à fait. Et c'est même le bon premier audit à effectuer : partir de `Main.lean`, calculer automatiquement toute la fermeture transitive de ses imports, puis rechercher les `sorry` dans chacun des fichiers atteints.

Il faut toutefois être rigoureux sur trois points :

- Ne pas s'arrêter aux imports directs : si le fichier A importe B, et B importe C, il faut aussi examiner C.
- Ne pas se limiter aux fichiers `OAI` : les dépendances peuvent aussi venir de `Mathlib` et d'autres bibliothèques.
- Ne pas confondre « aucun `sorry` détecté » avec « preuve mathématique correcte » : ce premier programme constitue un filtre, pas un certificat définitif.

Je vous propose un script Python qui parcourt récursivement les imports, construit la liste des fichiers visités et signale les occurrences de `sorry`. Il indiquera également les imports qu'il n'a pas réussi à résoudre : c'est essentiel, car une dépendance non retrouvée ne doit pas disparaître silencieusement de l'audit.

J'ai préparé une première version du programme, que vous pouvez récupérer ici :

Télécharger le script Python `audit_lean_imports.py`

Comment l'utiliser

1. Clonez le dépôt OpenAI sur votre ordinateur et installez ses dépendances Lean en suivant son README.
2. Placez le script à la racine du dépôt `openai/math`.
3. Lancez, dans un terminal :

```
python3 audit_lean_imports.py
```

Le module de départ est alors `OAI.Probability.CriticalPercolation.Main`, celui que vous avez choisi.

Le programme crée un rapport `lean_dependency_audit.txt`, contenant :

- le nombre de fichiers parcourus ;
- les occurrences de `sorry` détectées, avec fichier et numéro de ligne ;
- les imports non résolus ;
- la liste des fichiers effectivement parcourus.

Vous pouvez aussi choisir un autre module de départ :

```
python3 audit_lean_imports.py  
OAI.Probability.CriticalPercolation.GraphGeometry.ObjectKernel
```

Comment interpréter le rapport

Résultat	Interprétation
Des <code>sorry</code> sont trouvés	Il faut examiner ces occurrences : certaines peuvent être dans du code inactif ou résulter de faux positifs.
Aucun <code>sorry</code> trouvé	Bon signe, mais pas une preuve de complétude.
Des imports non résolus	L'audit est incomplet : il faut résoudre ces dépendances avant de conclure.
Aucun <code>sorry</code> et aucun import non résolu	Le débroussaillage est encourageant, mais il reste à contrôler les axiomes et à compiler la chaîne exacte.

Une limite à garder en tête : cette version est un premier outil d'exploration. Elle recherche le mot `sorry` dans les fichiers sources accessibles, mais elle ne remplace pas le vérificateur de Lean et ne garantit pas encore une fermeture complète dans toutes les configurations de dépendances.

À mon avis, c'est une bonne première étape. Le résultat vraiment intéressant sera de voir combien de fichiers sont parcourus, combien d'imports restent introuvables et si des `sorry` apparaissent loin dans la chaîne. Ces trois informations nous permettront de décider comment approfondir l'audit.

`audit_lean_imports.py`

Attends, tu as été un peu trop rapide, je préférerais considérer que Mathlib est fiable, qu'en penses-tu ?

Je suis tout à fait favorable à cette simplification. Pour un premier audit, considérer Mathlib comme une bibliothèque de confiance et concentrer notre enquête sur les fichiers propres à OpenAI me paraît une stratégie raisonnable.

Mais je distinguerais deux décisions.

1. Ce que nous décidons de ne pas auditer

Nous pouvons exclure Mathlib de notre recherche de `sorry`, en partant de l'hypothèse de travail que ses résultats sont fiables.

C'est un choix pragmatique : Mathlib est une bibliothèque mathématique largement utilisée, entretenue collectivement, dont les contributions sont soumises à des vérifications et à des contrôles de qualité.

Cela ne signifie pas que Mathlib serait infallible. Cela signifie simplement que nous ne voulons pas refaire tout son audit pour examiner le travail d'OpenAI.

2. Ce que nous devons tout de même vérifier

Il faut examiner les fichiers OAI et leurs dépendances propres, sans nous arrêter aux imports directs.

Par exemple, si le fichier OpenAI `Main.lean` importe `A.lean`, lequel importe `B.lean`, nous devons contrôler les deux fichiers OpenAI, même si `B.lean` n'est pas importé directement par `Main.lean`.

Pour cette première passe, je propose donc les règles suivantes :

- Mathlib : bibliothèque considérée comme fiable, exclue de notre recherche de `sorry`.
- Fichiers OAI : fermeture transitive complète des imports, recherche de `sorry` dans tous les fichiers sources concernés.
- Imports non résolus : signalés explicitement, jamais ignorés.

- Axiomes supplémentaires : recensés dans une seconde passe, si la première ne révèle pas de problème majeur.

Il reste une subtilité importante : il ne faut pas confondre « fichier situé dans le dépôt OpenAI » et « fichier appartenant au namespace **OAI** ». Le programme doit suivre les chemins des fichiers réellement résolus, pas seulement leurs noms de modules.

Je modifierais donc le script dans ce sens avant de le considérer prêt à l'emploi. Ce sera plus simple, plus ciblé et surtout plus utile pour répondre à votre question initiale : les résultats annoncés par OpenAI reposent-ils sur des démonstrations complètes dans leurs propres fichiers, ou certains maillons de la chaîne contiennent-ils des preuves laissées en suspens ?

Chat en pause jusqu'à la réinitialisation du quota à 13:44

Vous avez épuisé le quota de chats avec fichiers ou images. Démarrez un nouveau chat (texte uniquement), ou changez de forfait pour continuer.