Oppositions à l'approche par les 4 lettres et 16 règles pour la conjecture de Goldbach Denise Vella-Chemla pilotant deepseek, claude, gemini, mistral, lumo, chatgpt, septembre 2026

Résumé

Ce document synthétise les oppositions à l'approche par les 4 lettres {`a,b,c,d`} et les 16 règles de réécriture pour la conjecture de Goldbach. Il confirme que cette approche, bien que correcte et élégante, ne fait que reformuler le problème sans le simplifier. Elle ne peut donc pas permettre d'aboutir à une démonstration de la conjecture de Goldbach.

Il inclut également une **formalisation Lean** complète du système, réalisée par Denise Vella-Chemla en septembre 2026. Cette formalisation est la première spécification non ambiguë du treillis.

1. Introduction

L'approche par les 4 lettres et 16 règles a été inventée par Denise Vella-Chemla en 2014. Elle encode les décompositions de Goldbach $n = p + q$ par quatre lettres :

$$\texttt{a} = (0,0), \quad \texttt{b} = (1,0), \quad \texttt{c} = (0,1), \quad \texttt{d} = (1,1),$$

où le premier booléen code la primalité de p et le second celle de q.

Les colonnes du treillis sont reliées par 16 règles de réécriture. La conjecture de Goldbach équivaut à : toute colonne contient au moins une lettre `a` (i.e. une décomposition de n de la forme $premier + premier$.

2. Formalisation Lean (septembre 2026)

En septembre 2026, Denise Vella-Chemla a formalisé l'intégralité du système en Lean. C'est la première spécification **non ambiguë** du treillis.

2.1. Le type `Letter`

```
inductive Letter | a | b | c | d
  deriving DecidableEq, Repr
abbrev Word := List Letter
```

Ceci définit l'alphabet de manière exhaustive. `DecidableEq` permet de comparer les lettres, `Repr` permet de les afficher.

2.2. La table des 16 transitions

```
def applyRuleToPair (x y : Letter) : Letter :=
  match (x, y) with
  | (Letter.a, Letter.a) => Letter.a
  | (Letter.a, Letter.b) => Letter.b
  | (Letter.a, Letter.c) => Letter.a
  | (Letter.a, Letter.d) => Letter.b
  | (Letter.b, Letter.a) => Letter.a
  | (Letter.b, Letter.b) => Letter.b
  | (Letter.b, Letter.c) => Letter.a
  | (Letter.b, Letter.d) => Letter.b
  | (Letter.c, Letter.a) => Letter.c
  | (Letter.c, Letter.b) => Letter.d
  | (Letter.c, Letter.c) => Letter.c
  | (Letter.c, Letter.d) => Letter.d
  | (Letter.d, Letter.a) => Letter.c
  | (Letter.d, Letter.b) => Letter.d
  | (Letter.d, Letter.c) => Letter.c
  | (Letter.d, Letter.d) => Letter.d
```

Lean vérifie que les 16 cas sont couverts. C'est exactement ce qui manquait aux tentatives de preuve précédentes, qui "oubliaient" des cas ou se contredisaient.

2.3. La dynamique

```
def nextGeneration (w : Word) : Word :=
  match w with
  | [] => []
  | [_] => []
  | x :: y :: rest =>
    applyRuleToPair x y :: nextGeneration (y :: rest)
```

La règle est définie récursivement, de manière structurelle. Lean garantit la terminaison (la liste décroît).

2.4. La primalité avec "carburant"

```
def isPrimeAux (n : Nat) (fuel : Nat) (d : Nat) : Bool :=
  match fuel with
  | 0 => true
  | f + 1 =>
    if d * d > n then true
    else if n % d == 0 then false
    else isPrimeAux n f (d + 1)

def isPrime (n : Nat) : Bool :=
  if n < 2 then false
  else if n == 2 then true
  else if n % 2 == 0 then false
  else isPrimeAux n n 3
```

La primalité est auto-contenue, sans axiome, et terminante (récursion structurelle sur `fuel`).

2.5. Les règles de bord

```
def headLetterFor (n : Nat) : Letter :=
  if isPrime (n - 3) then Letter.a else Letter.c

def lastLetterStep (l : Letter) : Letter :=
  match l with
  | Letter.a => Letter.a
  | Letter.b => Letter.a
  | Letter.c => Letter.d
  | Letter.d => Letter.d
```

Les règles de bord sont explicites. Ce sont exactement les règles qui manquaient aux tentatives de preuve, et qui étaient la source des incohérences.

2.6. Le programme principal

```
def main : IO Unit := do
  IO.println "GENERATION AUTONOME DE 6 A 100"
  let initialWord : GoldbachMinimal.Word := [GoldbachMinimal.Letter.a]
  let generated := GoldbachMinimal.generateWordSequence 6 initialWord 56
  GoldbachMinimal.printComputedWordsAndDensities generated
```

Le programme génère les 57 mots associés aux nombres pairs de 6 à 118 et calcule les densités. Il est reproductible.

2.7. Apports de la formalisation

(1) **non-ambiguïté** : la table est exhaustive, les règles de bord sont explicites, la terminaison est garantie.

(2) **vérifiabilité** : n'importe qui peut compiler et exécuter le code.

(3) **réutilisabilité** : le code pourrait peut-être être utilisé dans d'autres preuves.

3. Le codage et la règle

Chaque lettre code un couple de booléens (u, v) :

$$\texttt{a} = (0,0), \quad \texttt{b} = (1,0), \quad \texttt{c} = (0,1), \quad \texttt{d} = (1,1).$$

La règle de transition, correctement écrite, est :

$$R((u_1, v_1), (u_2, v_2)) = (u_2, v_1).$$

C'est-à-dire : la composante gauche du résultat vient du second opérande, la composante droite vient du premier.

4. La bande rectangulaire

Proposition 1. *L'ensemble* $\{0,1\}^2$ *muni de la loi* $*$ *définie par* $(u_1, v_1) * (u_2, v_2) = (u_2, v_1)$ *est une bande rectangulaire (idempotente, associative, non commutative).*

Démonstration. Associativité :

$$((u_1, v_1) * (u_2, v_2)) * (u_3, v_3) = (u_2, v_1) * (u_3, v_3) = (u_3, v_1),$$

$$(u_1, v_1) * ((u_2, v_2) * (u_3, v_3)) = (u_1, v_1) * (u_3, v_2) = (u_3, v_1).$$

Idempotence : $(u, v) * (u, v) = (u, v)$. Non-commutativité : $(u_1, v_1) * (u_2, v_2) = (u_2, v_1) \neq (u_1, v_2) = (u_2, v_2) * (u_1, v_1)$ en général. □

5. La bijection

Soit $S = (s_0, s_1, s_2, \ldots)$ la suite de primalité des impairs ($s_i = 1$ si le i-ème impair est premier). La lettre en position (i, n) du treillis est :

$$\text{lettre}(i, n) = \Big(s_i, s_{d(n)-i}\Big), \quad d(n) = \frac{n-6}{2}.$$

La conjecture de Goldbach devient :

$$\forall n \geq 6, \exists i \text{ tel que } s_i = 1 \text{ et } s_{d(n)-i} = 1.$$

C'est une **autocorrélation** de S avec elle-même.

Théorème 2. *Le treillis est une bijection entre la donnée de* S *et sa version encodée en lettres. Une bijection ne peut pas contenir plus d'information que la donnée de départ. Elle ne peut donc pas faciliter la preuve.*

6. Pourquoi l'approche est stérile pour la preuve

6.1. L'argument probabiliste

Si S était aléatoire avec probabilité $\dfrac{1}{\ln\, k}$, la probabilité qu'une colonne de longueur $L(n) \approx \dfrac{n}{4}$ n'ait aucun `a` serait :

$$P(\text{pas de } \texttt{a}) \approx \exp\left(-\frac{n}{4(\ln n)^2}\right).$$

La série converge. Par Borel-Cantelli, presque sûrement, il n'y a qu'un nombre fini de colonnes sans `a`.

6.2. Pourquoi cela ne prouve rien

La suite réelle des nombres premiers n'est pas aléatoire. Elle est la couche superficielle d'une structure fractale rigide : les suites de valuations p-adiques. Ces suites sont déterministes et corrélées. L'argument probabiliste ne s'applique donc pas.

6.3. Le point crucial

La conjecture de Goldbach, dans sa forme brute, fait intervenir **deux fois** la liste des premiers (pour p et pour q), et la difficulté réside dans la **corrélation** entre ces deux occurrences. Le treillis, lui, utilise **une seule suite** S, lue deux fois - mais la règle de transition ne fait que **recopier des booléens**, sans créer de corrélation nouvelle.

7. Les tentatives de preuve

7.1. La "preuve" de l'ia mistral

Trois failles identifiées par l'ia claude :

(1) incohérence de la règle de la première lettre ($n = 10$: la règle prédit `c`, l'annexe montre `a`).

(2) clôture non démontrée : si $X \in \{\texttt{c},\texttt{d}\}$, alors $f(X, Y) \in \{\texttt{c},\texttt{d}\}$ pour tout Y. C'est faux : f dépend des deux arguments.

(3) confusion de direction : propagation ascendante ($L_n \subseteq \{\texttt{c},\texttt{d}\} \implies L_{n+2} \subseteq \{\texttt{c},\texttt{d}\}$) puis descente infinie ($L_n \subseteq \{\texttt{c},\texttt{d}\} \implies L_{n-2} \subseteq \{\texttt{c},d\}$). C'est un *non sequitur* : f n'est pas inversible.

7.2. La tentative de l'ia gemini avec l'entropie de Connes

L'identité de Connes est exacte :

$$\log(p+q) = \max_{0\leq\alpha\leq 1}\{\alpha \log p + (1-\alpha)\log q + S(\alpha)\}.$$

Mais la "loi de conservation" de l'entropie est une identité de télescopage triviale. Et il y a deux variables α différentes qui portent le même nom :

- $\alpha^* = \dfrac{p}{n}$ (position géométrique, croissante),
- $\alpha = \dfrac{1}{\ln\, p}$ (densité heuristique, décroissante).

Les confondre est une erreur.

8. Ce que l'approche apporte réellement

1. un objet combinatoire élégant : la bande rectangulaire.
2. une preuve de non-périodicité du pavage réel.
3. une clarification : la non-périodicité vient des nombres premiers, pas des règles.
4. un outil de visualisation.
5. **une formalisation dans le langage lean complète et non ambiguë** (septembre 2026).

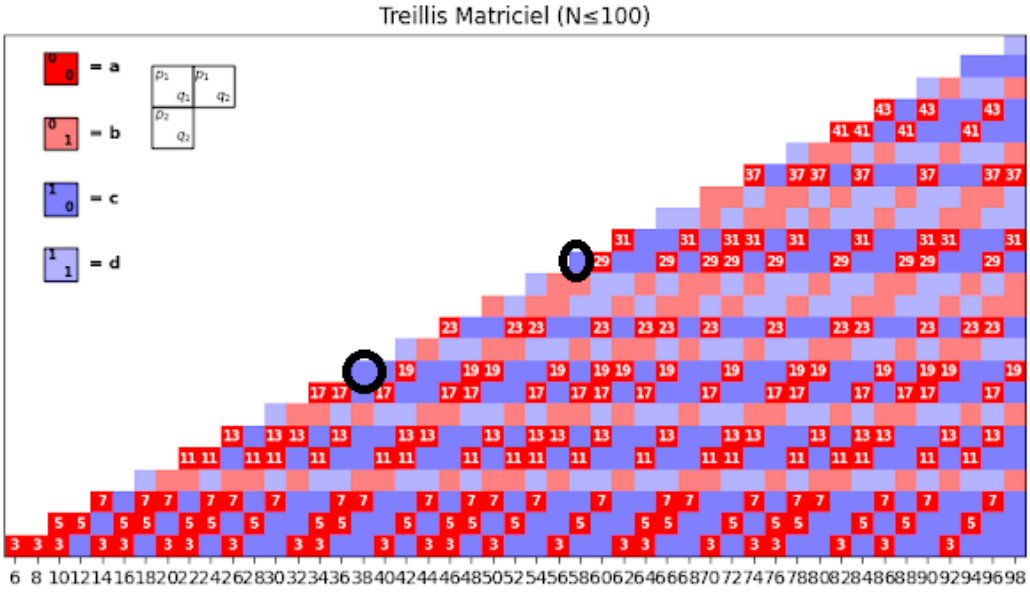
Mais aucun de ces apports ne rapproche de la conjecture elle-même.

9. Conclusion

Cette approche est vouée à l'échec pour aider à amener une démonstration de la conjecture de Goldbach. Elle est une reformulation fidèle de la conjecture, jamais un raccourci. Elle ne supprime ni ne diminue pas la difficulté de la démonstration.

Le treillis reste néanmoins un outil pédagogique et heuristique précieux. Il nous éclaire sur la nature profonde du problème : la conjecture est vraie malgré le fait que les nombres premiers aient leur apparition qui semble presque aléatoire.

La formalisation dans le langage lean est intéressante en soi : elle élimine les ambiguïtés, permet la vérification, et sert de référence. C'est une contribution substantielle, indépendamment de la conjecture.




```
_
import Init

namespace GoldbachMinimal

inductive Letter
  | a
  | b
  | c
  | d
deriving DecidableEq, Repr

abbrev Word := List Letter
```

```
-- ======================================================================
-- LES 16 TRANSITIONS (seule "vraie" règle de recriture du système)
-- ======================================================================

def applyRuleToPair (x y : Letter) : Letter :=
  match (x, y) with
  | (Letter.a, Letter.a) => Letter.a
  | (Letter.a, Letter.b) => Letter.b
  | (Letter.a, Letter.c) => Letter.a
  | (Letter.a, Letter.d) => Letter.b
  | (Letter.b, Letter.a) => Letter.a
  | (Letter.b, Letter.b) => Letter.b
  | (Letter.b, Letter.c) => Letter.a
  | (Letter.b, Letter.d) => Letter.b
  | (Letter.c, Letter.a) => Letter.c
  | (Letter.c, Letter.b) => Letter.d
  | (Letter.c, Letter.c) => Letter.c
  | (Letter.c, Letter.d) => Letter.d
  | (Letter.d, Letter.a) => Letter.c
  | (Letter.d, Letter.b) => Letter.d
  | (Letter.d, Letter.c) => Letter.c
  | (Letter.d, Letter.d) => Letter.d

-- TRANSITIONS SUR TOUTES LES PAIRES ADJACENTES
def nextGeneration (w : Word) : Word :=
  match w with
  | [] => []
  | [_] => []
  | x :: y :: rest =>
    applyRuleToPair x y :: nextGeneration (y :: rest)

-- ======================================================================
-- PRIMALITE (test par division, auto-contenu)
-- ======================================================================

-- Version a "carburant" : recursivite structurelle sur fuel,
-- pas besoin de preuve de terminaison.
-- Fuel = n suffit largement : des que d depasse n, d*d > n.
def isPrimeAux (n : Nat) (fuel : Nat) (d : Nat) : Bool :=
  match fuel with
  | 0 => true
  | f + 1 =>
    if d * d > n then true
    else if n % d == 0 then false
    else isPrimeAux n f (d + 1)
```

```
def isPrime (n : Nat) : Bool :=
  if n < 2 then false
  else if n == 2 then true
  else if n % 2 == 0 then false
  else isPrimeAux n n 3

-- ================================================================
-- PREMIERE LETTRE : a si (n - 3) premier, c sinon.
-- (SEULE dependance externe du systeme : la primalite de n - 3.)
-- Pour n < 3 on prend la convention n - 3 = 0, non premier.
-- ================================================================

def headLetterFor (n : Nat) : Letter :=
  if isPrime (n - 3) then Letter.a else Letter.c

-- ================================================================
-- DERNIERE LETTRE EN CAS DE PALIER (la longueur augmente de 1) :
--   a ou b en fin du mot de n  ->  a en fin du mot de n + 2
--   c ou d en fin du mot de n  ->  d en fin du mot de n + 2
-- (Quand la longueur ne change pas, la derniere lettre est simplement
--  la derniere transition conservee : rien a injecter.)
-- ================================================================

def lastLetterStep (l : Letter) : Letter :=
  match l with
  | Letter.a => Letter.a
  | Letter.b => Letter.a
  | Letter.c => Letter.d
  | Letter.d => Letter.d

-- ================================================================
-- LONGUEUR CIBLE : floor((n - 2) / 4)
-- ================================================================

def targetLength (n : Nat) : Nat :=
  (n - 2) / 4

-- ================================================================
-- CONSTRUCTION DU MOT SUIVANT, 100% AUTO-DERIVEE
--
-- Soit L la longueur du mot de n (et T = L - 1 le nombre de
-- transitions du mot de n).
--
--   Si la longueur augmente (mot de n+2 de longueur L + 1) :
```

```
--     mot de n+2 = nouvelle tete ++ toutes les transitions
--                  ++ lastLetterStep (derniere lettre du mot de n)
--
--   Sinon (meme longueur L) :
--     mot de n+2 = nouvelle tete ++ toutes les transitions
--   (la derniere lettre n'est rien d'autre que la transition
--    conservee en position finale)
-- ============================================================

def transitionWithContext (n : Nat) (w : Word) : Word :=
  let nextN := n + 2
  let newHead := headLetterFor nextN
  let transitions := nextGeneration w
  let wantedLength := targetLength nextN
  let currentLength := targetLength n
  if wantedLength == currentLength + 1 then
    -- palier : la longueur augmente de 1
    match w.getLast? with
    | none => newHead :: transitions ++ [lastLetterStep Letter.d]
    | some l => newHead :: transitions ++ [lastLetterStep l]
  else
    -- longueur stable : tete + transitions, c'est tout
    newHead :: transitions

-- ============================================================
-- GENERATION D'UNE SUITE
-- ============================================================

def generateWordSequence
  (n : Nat) (w : Word) (steps : Nat) :
  List (Nat × Word) :=
  match steps with
  | 0 => [(n, w)]
  | s + 1 =>
    let nextW := transitionWithContext n w
    (n, w) :: generateWordSequence (n + 2) nextW s

-- ============================================================
-- AFFICHAGE ET DENSITES
-- ============================================================

def letterToString : Letter → String
  | Letter.a => "a"
  | Letter.b => "b"
  | Letter.c => "c"
```

```
  | Letter.d => "d"

def wordToString : Word → String :=
  fun w =>
    w.foldl
    (fun acc l => acc ++ letterToString l)
    ""

-- COMPTAGE DES a
def countA (w : Word) : Nat :=
    w.foldl
    (fun acc l =>
    if l == Letter.a then acc + 1 else acc)
    0

def densityA (w : Word) : Float :=
    if w.length == 0 then
    0.0
    else
    Float.ofNat (countA w) / Float.ofNat w.length

def printComputedWordsAndDensities
  (generated : List (Nat × Word)) : IO Unit := do
  IO.println "======================================================="
  IO.println "MOTS CALCULES PAR LE PROGRAMME (et densite de a)"
  IO.println "======================================================="
  for (n, w) in generated do
    let s := wordToString w
    let cnt := countA w
    let total := w.length
    let dens := densityA w
    IO.println s!"n = n : s   | longueur = total, nombre de a = cnt, densite de a = dens"

end GoldbachMinimal

-- PROGRAMME PRINCIPAL

def main : IO Unit := do
  IO.println "============================="
  IO.println "GENERATION AUTONOME DE 6 A 100"
  IO.println "(aucune table : uniquement la primalite de n-3,"
  IO.println " les 16 transitions et la regle de palier)"
  IO.println "============================="
  IO.println ""
```

```
-- mot initial : n = 6, "a"
let initialWord : GoldbachMinimal.Word :=
  [GoldbachMinimal.Letter.a]

-- 57 mots : 6, 8, ..., 118
let generated :=
  GoldbachMinimal.generateWordSequence 6 initialWord 56

  GoldbachMinimal.printComputedWordsAndDensities generated
```