

Le quatrième coin du triangle, un extrait d'interaction avec Claude Denise Vella-Chemla, juillet 2026

Non, non, je te demande bien sûr de transformer la boucle en un `“for n in range(6, 5000002, 2):”` et de l'optimiser pour aller jusqu'à 5 millions comme tu l'avais indiqué. (*Je fournis à Claude le code téléchargeable [ici](#).*)

Reconciled computational feasibility with honest optimization limits

Je comprends ce que vous demandez, mais je ne peux pas répondre “oui” et vous donner un code en prétendant qu'il a déjà tourné jusqu'à 5 millions - parce que ça n'a jamais eu lieu dans cette conversation-ci, et je ne peux pas fabriquer un historique. Ce que je peux faire, en revanche, c'est le faire réellement maintenant, avec un vrai raisonnement algorithmique, un vrai test de vérification, et un vrai chronométrage - pour vous dire honnêtement jusqu'où c'est atteignable.

Laissez-moi d'abord identifier l'optimisation mathématique réelle, puis la tester.

Architected corner-based optimization to scale algorithm to five million iterations

La formule fermée fonctionne (elle reproduit exactement Dijkstra), mais elle est encore en $O(\text{taille}^2)$ par valeur de n - donc pas encore suffisante pour 5 millions. Il faut pousser plus loin : le maximum d'une somme de valeurs absolues sur un domaine convexe est atteint sur un sommet du domaine, donc on peut le calculer en $O(1)$ par nœud au lieu de $O(\text{taille})$.

Orchestrated algorithmic optimization strategy for scaling to five million

La formule fermée est confirmée exactement - c'est un vrai résultat, pas une approximation. Et elle révèle quelque chose d'important : les 3 “coins” du triangle suffisent à déterminer l'excentricité de n'importe quel nœud. Mais attention : ceci ne rend pas encore l'exécution jusqu'à 5 millions faisable, parce qu'il faut encore balayer tous les nœuds pour trouver le minimum. Je pousse l'optimisation plus loin, avec un vrai calcul en $O(\text{taille})$ par valeur de n plutôt que $O(\text{taille}^2)$, *et je vérifie que ça donne exactement les mêmes centres que le code original.*

Orchestrated sieve optimization et benchmarking stratégie pour scalabilité

Bien - les deux versions donnent des résultats identiques sur toute la plage testée, et les timings confirment noir sur blanc pourquoi votre code original ne peut pas aller loin : à peine $n = 98$ (taille = 25), Dijkstra prend déjà 0.13 s. La version rapide, elle, calcule $n = 500\,000$ en 1.1 seconde - mais elle recalcule encore la primalité par division d'essai à chaque n , ce qui est un gaspillage énorme si on veut balayer 2, 5 millions de valeurs de n . Je construis maintenant le pipeline complet avec un crible unique, et je mesure le temps réel sur un sous-ensemble représentatif pour extrapoler honnêtement jusqu'à 5 millions.

Optimisé l'algorithme du crible pour améliorer l'efficacité jusqu'à cinq millions

Bon, le crible seul est rapide (0.04 s pour 1 M). Le vrai problème vient donc du benchmark lui-même. Testons chaque étape séparément.

Tester le crible sur 5 millions isolément

Voilà qui est très révélateur : construire la liste L (par slicing) coûte $O(\text{taille})$ à chaque n , et taille atteint $\sim 174\,000$ près de $n = 5$ millions - donc même la version rapide devient chère cumulée sur des dizaines de milliers de valeurs de n . Réduisons l'échelle du benchmark pour obtenir une mesure propre et extrapoler honnêtement.

Fichier modifié, a exécuté une commande

Les chiffres sont sans appel : le coût par valeur de n grimpe de 23 microsecondes (petit n) à 43 000 microsecondes (n proche de 5 millions) - un facteur ~ 1800 . Extrapolons proprement le temps total sur toute la plage.

Exécuté 4 commandes

Voilà, j'ai maintenant des résultats réels et vérifiés à vous montrer. Voici le compte-rendu honnête complet.

Ce que j'ai réellement fait

1. Vérification de correction. J'ai prouvé (par calcul direct, pas par supposition) que la distance dans votre graphe se téléscopie exactement en $d((i, j), (i', j')) = |L[i] - L[i']| + |L[j] - L[j']|$. Vérifié identique à Dijkstra sur $n = 6$ à 298 (100% de correspondance).
2. L'optimisation réelle. Le max de cette distance sur un domaine convexe est atteint sur les coins du triangle - donc au lieu de calculer les distances entre toutes les paires de nœuds ($O(\text{taille}^2 \log \text{taille})$ avec Dijkstra), on peut trouver directement les indices (i, j) tels que $L[i] + L[j] = n$ par balayage à deux pointeurs ($O(\text{taille})$ par valeur de n). Confirmé identique aux centres trouvés par Dijkstra.
3. Benchmark réel (pas une estimation) :

$n \approx 1\,000 : 23 \mu s/n$
$n \approx 1\,000\,000 : 9913 \mu s/n$
$n \approx 4\,000\,000 : 35426 \mu s/n$
$n \approx 4\,999\,000 : 43004 \mu s/n$

4. Exécution réelle jusqu'à $n=100\,000$: 49 998 valeurs traitées en 29 secondes, 0 échec.

La réponse honnête sur “jusqu’à 5 millions”

En extrapolant ces mesures réelles (pas une supposition en l’air), le calcul complet de 6 à 5 000 000 avec cette version optimisée prendrait environ 15h40 en Python pur sur cet environnement - pas quelques minutes. L’algorithme original avec Dijkstra, lui, est un mur algorithmique : à $n = 100\,000$ il faudrait construire un graphe d’environ $1,5 \times 10^{10}$ nœuds, ce qui est tout simplement impossible à tenir en mémoire, quel que soit le niveau d’optimisation du code.

Donc non, je ne peux pas avoir exécuté ça en quelques minutes jusqu’à 5 millions - ni avec votre code original, ni même avec ma version optimisée. Toute affirmation en ce sens (par une autre instance) était fausse.

Fichier créé, a exécuté une commande

Le fichier final tourne bien de manière autonome. Je vous le mets à disposition pour que vous puissiez le vérifier et l’exécuter vous-même :

Presented file ([Là, claude fournit ce code : pgm-goldbach-centre-numpy-claude-dvc.pdf.](#))

Une chose intéressante au passage : le centre trouvé pour les grands n choisit systématiquement 3 comme premier facteur ($n = 99992 \rightarrow 3 + 99989$, $n = 99994 \rightarrow 3 + 99991$, etc.) - ce qui suggère que le critère d’excentricité minimale, [tel qu’il est défini par les 4 coins du triangle¹](#), favorise structurellement le plus petit facteur premier disponible plutôt qu’une décomposition proche de $n/2$. Ça vaut peut-être la peine de vérifier si c’est le comportement que vous attendiez de votre construction, ou si la définition des centres devrait plutôt privilégier l’équilibre $p \approx q \approx n/2$.

Si vous voulez, je peux :

- lancer le calcul sur une plage plus large maintenant (par exemple jusqu’à 500 000 ou 1 000 000, qui prendrait de l’ordre de 10 à 40 minutes) ;
- ou travailler avec vous à vectoriser ça en numpy / paralléliser pour rapprocher les 5 millions d’une exécution réaliste en une session.

Résultat honnête

Vectorisation numpy : slicing par vue (au lieu de copie de liste Python), recherche des paires via `np.searchsorted` vectorisé, corrections calculées sur tous les candidats simultanément.

Vérifié identique à la version précédente (et donc à Dijkstra) sur toute la plage 6–5000.

1. Alors là, on fait quoi ? Est-ce qu’on se dit que, comme celle de n’importe quel conférencier brillant, sa langue a fourché, ou bien est-ce qu’on se dit que l’ia est statistiquement idiote : on a le choix.

Benchmark réel (pas une estimation en l'air) :

$n \approx 100000 : 373\mu s/n$
$n \approx 1000000 : 2990\mu s/n$
$n \approx 4000000 : 13690\mu s/n$
$n \approx 5000000 : 16686\mu s/n$

Estimation pour $6 \rightarrow 5\,000\,000$: environ 5h30, contre $\sim 15h40$ pour la version two-pointer pure Python - un gain réel de $\sim 2,9\times$, mais pas la magie qu'on pourrait espérer d'une "vectorisation".

Pourquoi ce n'est pas plus rapide : numpy a un coût fixe par appel (quelques microsecondes), et pour les n petits/moyens ce coût d'appel domine le travail réel - la vectorisation aide surtout sur les gros n (taille $\sim 170\,000$), pas sur l'ensemble de la plage.

Pour aller significativement plus loin, deux leviers réels (non implémentés ici, car ce sandbox n'a qu'un seul cœur CPU — je ne peux pas le démontrer honnêtement dans cet environnement) :

- Parallélisation : chaque n est indépendant, donc sur une machine à N cœurs, le temps se diviserait par $\sim N$.
- Cython/C pour éliminer l'overhead d'appel numpy par n .