

Tutoriel

Algorithmique · Python · interface graphique

Résoudre le problème des N-Reines avec Python et une GUI

Construire un solveur de backtracking pour les N-Reines, le tester et visualiser ses solutions dans une interface Tkinter interactive.

Auteur **Youness Boutyour**



DURÉE

60 minutes

NIVEAU

Débutant avancé

FORMAT

Tutoriel pratique

Algorithmique · Python · interface graphique · publié le 2026-05-13 · local

Objectifs et livrables

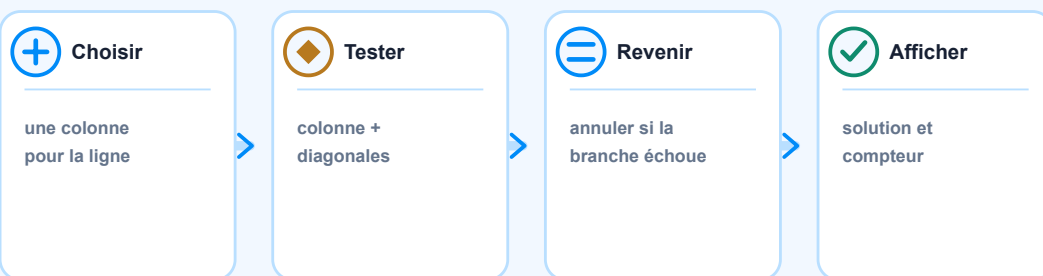
Le problème des N-Reines consiste à placer N reines sur un échiquier $N \times N$ sans qu'aucune paire de reines ne partage une ligne, une colonne ou une diagonale. L'énoncé paraît simple, mais le nombre de placements possibles grandit très vite : une recherche naïve essaierait jusqu'à N^N configurations. Nous allons transformer ce problème en une exploration progressive, où une branche impossible est abandonnée dès qu'un conflit apparaît.

Le tutoriel construit un programme complet en Python. La représentation du plateau, le test de validité et le backtracking seront d'abord exécutables dans le terminal. Nous ajouterons ensuite une fenêtre Tkinter : elle dessinera l'échiquier, affichera chaque solution trouvée et permettra de régler N, la vitesse d'animation et l'arrêt de la recherche. Un fil d'exécution séparé empêchera le calcul de bloquer les boutons de la fenêtre.

À la fin, l'archive contient le fichier exécutable, les tests unitaires et une commande de comptage. Pour $N=8$, l'exécution retrouve les 92 solutions connues et la GUI en affiche une après l'autre. Ce projet n'utilise ni LLM ni API générative : le fichier `llm_client.py` de l'archive documente explicitement que la démonstration est locale et déterministe. Il pourra servir de frontière si un agent doit un jour demander au solveur de calculer une taille donnée.

Explorer une solution sans bloquer la GUI

Schéma du backtracking des N-Reines : choisir une colonne, vérifier les conflits, revenir en arrière puis afficher la solution.



• une reine par ligne · élagage · animation Tkinter

n-reines-python-gui

Le solveur avance lorsqu'une colonne est sûre et restaure l'état précédent lorsqu'une branche échoue.

Quelques repères sur le problème

Une solution comme permutation

Nous plaçons exactement une reine par ligne. Le plateau peut donc être décrit par une liste `board` telle que `board[row]` soit la colonne choisie pour cette ligne. Pour $N=4$, `[1, 3, 0, 2]` signifie : ligne 0 en colonne 1, ligne 1 en colonne 3, ligne 2 en colonne 0 et ligne 3 en colonne 2.

Cette représentation garantit déjà l'absence de deux reines sur une même ligne. Les colonnes doivent être toutes différentes. Pour les diagonales, deux cases partagent la diagonale descendante si `row - column` est identique ; elles partagent l'autre diagonale si `row + column` est identique. Trois ensembles Python permettent de tester ces contraintes sans parcourir les reines déjà placées.

Pourquoi le backtracking suffit ici

À la ligne `r`, le solveur essaie chaque colonne encore disponible. Il ajoute la colonne et les deux diagonales aux ensembles, descend à la ligne suivante, puis retire ces valeurs au retour. Ce retrait est essentiel : il restaure l'état de la branche précédente avant d'essayer la colonne suivante.

Le pire cas reste exponentiel, mais l'élagage évite la plupart des placements inutiles. La mémoire utilisée par la recherche est en $O(N)$ pour le plateau et les ensembles, hors solutions déjà consommées par l'appelant. Le générateur renvoie les solutions une par une ; la GUI n'a donc pas besoin de stocker toute la liste.

Lire le flux avant de coder

Le schéma représente une seule branche de l'arbre de recherche. Le solveur choisit une case, vérifie les contraintes déjà rencontrées, puis descend d'un niveau seulement si la case est sûre. Le retour arrière n'est pas un échec global : il signifie que la branche courante ne peut plus produire de solution et que la recherche doit reprendre au dernier choix.

La séparation entre solveur et interface est essentielle. Le solveur produit des solutions sans connaître Tkinter ; la GUI consomme ces solutions et peut interrompre le parcours. Cette frontière permet de tester le comptage dans un terminal et de régler ensuite l'animation sans modifier la logique algorithmique.

1. Préparer l'environnement et le projet

Créez une arborescence courte. Le module `src/n_queens.py` contiendra à la fois le solveur et la GUI ; les tests n'importeront que les fonctions algorithmiques, ce qui permet de les exécuter même dans un environnement sans écran.

```
Terminal · Bash

mkdir -p n-reines-python-gui/{src,tests}
cd n-reines-python-gui
python3.11 -m venv .venv
source .venv/bin/activate      # Windows : .venv/Scripts/activate
python -m pip install --upgrade pip
python --version
python -c "import tkinter; print('Tkinter: disponible')"
```

Sur Ubuntu ou Debian, l'import peut échouer si le module graphique n'est pas installé avec Python :

```
Terminal · Bash

sudo apt-get install python3-tk
```

La résolution et les tests n'utilisent aucun paquet externe. Pour vérifier rapidement l'organisation du projet, créez aussi [requirements.txt](#) avec le commentaire suivant :

```
■ Sortie / exemple

# Dépendances Python externes : aucune.
# tkinter est une bibliothèque standard, parfois distribuée comme paquet système.
```

Sortie obtenue

```
■ Sortie / exemple

Python 3.11.x
Tkinter: disponible
```

Commande de vérification

L'interpréteur utilisé pour les tests et celui qui ouvre la fenêtre doivent être le même. Cette commande confirme le chemin réellement actif :

```
Terminal · Bash

python -c "import sys; print(sys.executable); print(sys.version.split()[0])"
```

Sortie / exemple

```
./.../n-reines-python-gui/.venv/bin/python  
3.11.x
```

2. Modéliser l'échiquier et écrire le solveur

Commencez par le cœur indépendant de toute interface. `iter_solutions` est un générateur : dès qu'une ligne complète est placée, il cède un tuple au code appelant. L'événement d'arrêt est optionnel ; il sera utilisé par le bouton « Arrêter » de la GUI.

Python

```
from collections.abc import Iterator
from threading import Event

def iter_solutions(n: int, stop_event: Event | None = None) → Iterator[tuple[int, ...]]:
    board = [-1] * n
    columns: set[int] = set()
    diagonals_down: set[int] = set() # row - column
    diagonals_up: set[int] = set() # row + column

    def visit(row: int) → Iterator[tuple[int, ...]]:
        if stop_event is not None and stop_event.is_set():
            return
        if row == n:
            yield tuple(board)
            return

        for column in range(n):
            down = row - column
            up = row + column
            if column in columns or down in diagonals_down or up in diagonals_up:
                continue
            board[row] = column
            columns.add(column)
            diagonals_down.add(down)
            diagonals_up.add(up)
            yield from visit(row + 1)
            columns.remove(column)
            diagonals_down.remove(down)
            diagonals_up.remove(up)
            board[row] = -1

    yield from visit(0)
```

Le `yield from` transmet les solutions découvertes dans les appels récursifs. Le bloc de nettoyage est exécuté après le retour de la récursion, y compris après l'exploration d'une branche qui ne mène à aucune solution. C'est ce qui permet de réutiliser exactement le même état pour le choix suivant.

Ajoutez une fonction de comptage et une validation de taille. La limite 12 est volontaire pour garder une animation interactive raisonnable ; le cœur algorithmique pourrait être optimisé autrement pour des tailles supérieures.

Python

```
MAX_N = 12

def validate_n(n: int) → None:
    if not isinstance(n, int) or not 1 ≤ n ≤ MAX_N:
        raise ValueError(f"n doit être un entier compris entre 1 et {MAX_N}")

def count_solutions(n: int) → int:
    validate_n(n)
    return sum(1 for _ in iter_solutions(n))
```

Sortie obtenue

Ajoutez temporairement un point d'entrée terminal, ou utilisez celui de l'archive finale :

Terminal · Bash

```
python -m src.n_queens --n 4 --count
python -m src.n_queens --n 8 --count
```

Sortie / exemple

```
N=4 : 2 solution(s) en 0.000 s
N=8 : 92 solution(s) en 0.0xx s
```

Le temps varie selon la machine. Le nombre de solutions, lui, doit rester identique.

Commande de vérification

Une solution est valide si ses colonnes, ses différences `row-column` et ses sommes `row+column` sont toutes distinctes :

Terminal · Bash

```
python - <<'PY'
from src.n_queens import count_solutions, iter_solutions

solutions = list(iter_solutions(4))
assert count_solutions(1) == 1
assert count_solutions(4) == 2
assert len(solutions) == 2
for solution in solutions:
    assert len(set(solution)) == 4
    assert len({row - col for row, col in enumerate(solution)}) == 4
    assert len({row + col for row, col in enumerate(solution)}) == 4
print("contraintes N=4: PASS")
PY
```

Sortie / exemple

contraintes N=4: PASS

3. Construire la fenêtre Tkinter

Une interface graphique doit rester réactive pendant la recherche. Le solveur tourne donc dans un thread de travail. Il ne touche jamais directement aux widgets : il place des événements dans une `queue.Queue`, puis la boucle Tkinter les lit régulièrement avec `after`. Cette règle évite les mises à jour concurrentes de la fenêtre.

Le dessin est volontairement simple. Une case alterne entre blanc et bleu clair ; une reine est un cercle bleu marqué d'une couronne. Le nombre de cases est adapté à la taille N, et la dernière solution reçue remplace le dessin précédent.

Python

```
events: queue.Queue[tuple] = queue.Queue()
stop_event = threading.Event()

def solve_in_background(n: int, delay: float) -> None:
    count = 0
    for solution in iter_solutions(n, stop_event):
        count += 1
        events.put(("solution", solution, count))
        if delay:
            time.sleep(delay)
    events.put(("done", count, stop_event.is_set()))
```

Dans le thread principal, `poll_events` récupère les messages et appelle `draw_board`. Le bouton d'arrêt ne tue pas brutalement le thread : il positionne un événement ; le générateur le consulte entre deux placements et s'arrête proprement.

```
def poll_events() → None:
    try:
        while True:
            event = events.get_nowait()
            if event[0] == "solution":
                _, solution, count = event
                draw_board(solution)
                status_var.set(f"Solution {count} trouvée")
            elif event[0] == "done":
                _, count, stopped = event
                status_var.set(f"{count} solution(s) trouvée(s)")
    except queue.Empty:
        pass
    root.after(40, poll_events)
```

Le code complet téléchargeable ajoute les boutons, la validation de N, le réglage de la pause, la gestion des erreurs et le dessin du symbole de reine. Il propose N de 1 à 12. Pour une première observation, choisissez N=4 avec une pause de 0,30 s ; pour voir le comptage classique, utilisez N=8 sans ouvrir de fenêtre avec `--count`.

Sortie obtenue

Après `python -m src.n_queens --n 4`, une fenêtre intitulée « Résoudre le problème des N-Reines » apparaît. Le statut évolue ainsi :

```
Sortie / exemple

Recherche en cours pour N=4...
Solution 1 trouvée pour N=4
Solution 2 trouvée pour N=4
2 solution(s) trouvée(s) en 0.61 s.
```

Le plateau affiche une reine par ligne et par colonne. Cliquer sur « Arrêter » pendant une recherche plus longue change le statut en « arrêt demandé », puis réactive le bouton « Lancer » lorsque le thread a terminé.

Commande de vérification

Lancez la GUI avec la taille minimale puis vérifiez manuellement trois points :

 Terminal · Bash

```
python -m src.n_queens --n 1
```

 Sortie / exemple

Une fenêtre avec une seule case et une seule reine est affichée.

La commande ouvre une fenêtre et ne rend pas le terminal tant que celle-ci est ouverte. Dans un environnement sans serveur graphique, utilisez `--count` et les tests de la section suivante ; le solveur ne dépend pas de l'écran.

4. Ajouter les tests et le point d'entrée CLI

Les tests vérifient deux propriétés différentes : les valeurs de référence confirment que l'exploration ne perd ni ne duplique les solutions, tandis que les contraintes contrôlent chaque tuple produit. Les nombres de référence sont petits pour rester rapides dans un environnement de formation.

 Python

```
import unittest

from src.n_queens import count_solutions, iter_solutions

class NQueensTests(unittest.TestCase):
    def test_known_solution_counts(self) → None:
        self.assertEqual(count_solutions(1), 1)
        self.assertEqual(count_solutions(4), 2)
        self.assertEqual(count_solutions(5), 10)

    def test_every_solution_has_distinct_columns(self) → None:
        for solution in iter_solutions(5):
            self.assertEqual(len(set(solution)), 5)
            self.assertEqual(len({r - c for r, c in enumerate(solution)}), 5)
            self.assertEqual(len({r + c for r, c in enumerate(solution)}), 5)
```

Le point d'entrée `argparse` accepte `--n` et `--count`. Sans `--count`, il ouvre la GUI ; avec cette option, il imprime un résultat exploitable dans un script ou un notebook. Cette séparation est pratique pour comparer le résultat à un agent, un benchmark ou une future interface web sans mélanger le calcul et le rendu graphique.

Sortie obtenue

Terminal · Bash

```
python -m unittest discover -s tests -v
```

Sortie / exemple

```
test_every_solution_has_distinct_columns ... ok
test_known_solution_counts ... ok
```

```
-----
Ran 2 tests in 0.0xxs
```

```
OK
```

Commande de vérification

Exécutez aussi plusieurs tailles dans le terminal :

Terminal · Bash

```
for n in 1 4 5 8; do python -m src.n_queens --n "$n" --count; done
```

Sortie / exemple

```
N=1 : 1 solution(s) en 0.000 s
N=4 : 2 solution(s) en 0.000 s
N=5 : 10 solution(s) en 0.000 s
N=8 : 92 solution(s) en 0.0xx s
```

Les durées sont indicatives ; les quantités sont les valeurs à contrôler.

5. Exécuter le projet complet

Téléchargez l'archive ou copiez les fichiers dans le répertoire créé, puis exécutez les tests avant d'ouvrir la fenêtre. Une séquence de travail fiable consiste à valider le calcul sans interface, tester l'interface avec N=4, puis augmenter la taille.

 Terminal · Bash

```
python -m unittest discover -s tests -v
python -m src.n_queens --n 8 --count
python -m src.n_queens --n 4
```

Pour faire apparaître les deux solutions N=4 plus lentement, indiquez une pause dans la fenêtre. La pause ne modifie jamais le résultat ; elle contrôle uniquement la vitesse à laquelle les événements sont envoyés à l'interface.

Sortie obtenue

Le scénario complet produit une sortie terminal comparable à celle-ci :

 Sortie / exemple

```
OK
N=8 : 92 solution(s) en 0.0xx s
```

Dans la fenêtre, le dernier statut indique le nombre total de solutions et le temps mesuré depuis le lancement. Le plateau final est une configuration valide, mais ce n'est pas nécessairement la même représentation que celle obtenue sur une autre machine si l'ordre de parcours est modifié.

Commande de vérification

Vérifiez que les fichiers de l'archive sont présents et qu'aucun cache Python ne se glisse dans le livrable :

 Terminal · Bash

```
find src tests -type f -maxdepth 2 -print
find . -type d -name __pycache__ -prune -print
```

 Sortie / exemple

```
src/n_queens.py
src/llm_client.py
tests/test_n_queens.py
```

Le second `find` doit rester vide dans l'archive publiée ; les caches créés localement peuvent être supprimés avant un partage du projet.

Exemples de sorties

Voici les configurations renvoyées par le solveur pour N=4, sous forme de colonnes par ligne :

Sortie / exemple

```
(1, 3, 0, 2)
(2, 0, 3, 1)
```

Pour `(1, 3, 0, 2)`, les reines se trouvent en `(0,1)`, `(1,3)`, `(2,0)` et `(3,2)`. Les colonnes sont toutes différentes, les valeurs `row-column` valent `-1, -2, 2, 1` et les valeurs `row+column` valent `1, 4, 2, 5`. Chaque ensemble contient donc quatre valeurs distinctes.

Dépannage et diagnostics

`ModuleNotFoundError: No module named tkinter` — installez le paquet système correspondant à votre distribution (`python3-tk` sur Debian/Ubuntu), puis relancez la commande avec le même interpréteur que celui du terminal.

La fenêtre ne répond pas — vérifiez que vous utilisez le fichier complet de l'archive : le thread de recherche et `root.after` sont nécessaires. Ne placez pas un appel direct à `count_solutions(12)` dans le callback d'un bouton ; il bloquerait la boucle Tkinter jusqu'à la fin du calcul.

Le bouton Arrêter semble tarder — l'arrêt est coopératif. Le thread termine le petit appel récursif courant avant de publier l'événement `done`. Une pause de 0,05 à 0,10 s rend l'animation lisible ; pour mesurer la performance, utilisez `--count` sans délai.

Le nombre de solutions paraît différent — contrôlez la valeur de N et l'unité utilisée. Le programme compte les solutions orientées : les rotations et réflexions sont comptées séparément. Pour N=8, le résultat attendu est 92, et non le nombre de classes modulo symétries.

Vérification finale

Le tutoriel est terminé lorsque les quatre commandes suivantes réussissent :

Terminal · Bash

```
python -m unittest discover -s tests -v
python -m src.n_queens --n 4 --count
python -m src.n_queens --n 8 --count
python -m compileall -q src tests
```

■ Sortie / exemple

```
OK
N=4 : 2 solution(s) en 0.000 s
N=8 : 92 solution(s) en 0.0xx s
```

Vous pouvez ensuite ouvrir `python -m src.n_queens --n 8`, observer les solutions et arrêter l'animation. Le livrable est reproductible, ne dépend pas d'un service distant et conserve une séparation claire entre calcul, rendu et tests.

Limites et prolongements

Le backtracking présenté est idéal pour comprendre les contraintes, mais son temps de calcul devient important lorsque N augmente. Pour aller plus loin, mesurez le nombre de nœuds visités, ajoutez une heuristique de choix de colonne, exploitez les symétries ou comparez une version bitmask à la version utilisant des ensembles.

La GUI affiche les solutions mais ne visualise pas encore les retours arrière. Un mode pas-à-pas pourrait envoyer des événements `place`, `reject` et `remove`, puis afficher la profondeur de la récursion. Une interface web ou un agent pourrait appeler `count_solutions` comme un outil local, à condition de borner N, le temps CPU et le nombre de résultats. Dans ce cas, la validation du serveur doit rester indépendante du prompt ou du modèle qui demande le calcul.

Références

- [Python — documentation de tkinter](#)
- [Python — threading et événements](#)
- [Python — unittest](#)