**Designing an n-Queens Puzzle Game: From Random Solutions to Playable Levels**

**Introduction**

The classic **n-Queens problem**—placing n queens on an n×n chessboard so that no two queens attack each other—has long fascinated computer scientists and puzzle lovers alike. Even LinkedIn featured a Queens puzzle where users tried to place queens in 60 seconds!

Many articles and posts have given ideas and codes for solving puzzles. However, building a *puzzle game* around n-Queens introduces a twist. It’s not just about solving the arrangement once. Instead, we need to **generate distinct, playable “levels”** for users, each with a unique solution or a limited number of solutions, plus additional constraints to make it fun and challenging.

This blog post will walk through **how** I’ve tackled that challenge:

* We’ll still solve the classic n-Queens puzzle—but *randomly*, to produce variety.
* We’ll then add **colored regions** so that each row, column, *and* colored region must have exactly one queen.
* Finally, we’ll show how to ensure each puzzle has an interesting structure and is solvable in a unique or near-unique way.

**Note:** This article does *not* focus on the front-end or Django aspects. We’ll dive into the **procedural generation algorithms** behind the scenes, using Python, Cython, and a healthy dose of backtracking, flood fill, and multiprocessing.

**Game Rules**

**The Board Layout**

1. **One Queen Per Row:** Each row on the chessboard must contain exactly one queen.
2. **One Queen Per Column:** Each column must contain exactly one queen.
3. **One Queen Per Color Region:** A board is divided into distinct colored regions, each of which may only contain one queen.
4. **No Attacks Allowed:** Queens cannot attack each other—meaning no two queens share the same row, column, or diagonal.

Below is an example of a **randomly generated valid 5×5 queens solution** (before we add color constraints). Each queen is denoted with a red “Q.”

📌 **(*instruction*) Insert the first image showing an n=5 Queens solution**

This basic arrangement is the familiar n-Queens problem. The difference comes next, when we break the board into color regions.

**Procedural Puzzle Generation Algorithm**

Let’s explore the puzzle generation in four main steps:

**Step 1: Generate a Random Valid n-Queens Solution**

Even though you might know a deterministic algorithm for n-Queens, it’s crucial to **introduce randomness** so that each puzzle is different. A straightforward way to do this is via **randomized backtracking**.

Below is a **Cython-optimized** snippet that shows the high-level idea (abridged from genertor\_cy.pyx):

cython

复制编辑

cdef bint \_backtrack(int row, int n, list solution,

object cols, object diag1, object diag2, list sol):

if row == n:

sol[0] = solution.copy()

return True

# Shuffle the columns for random exploration

cdef list cols\_list = list(range(n))

random.shuffle(cols\_list)

for col in cols\_list:

# Check conflicts

if col in cols or (row - col) in diag1 or (row + col) in diag2:

continue

# Place the queen

solution[row] = col

cols.add(col)

diag1.add(row - col)

diag2.add(row + col)

# Recurse

if \_backtrack(row + 1, n, solution, cols, diag1, diag2, sol):

return True

# Backtrack

cols.remove(col)

diag1.remove(row - col)

diag2.remove(row + col)

return False

cpdef list random\_n\_queens(int n):

cdef list solution = [None] \* n

...

\_backtrack(0, n, solution, cols, diag1, diag2, sol)

return sol[0]

* We maintain three sets (cols, diag1, and diag2) to quickly check if a spot conflicts with existing queens.
* For each row, we **shuffle** the columns. Randomizing column order speeds up “finding a random solution” because we effectively sample from the large solution space.

Once we have this valid arrangement, we proceed to **color regions**.

**Step 2: Generate Color Regions (Region Partitioning)**

The next challenge: ensure each *colored region* on the board also contains exactly one queen. We do this by:

1. **Seeding** each region with exactly one queen’s position.
2. **Flood filling** outward from each seed to color the rest of the board.

Below, we have an image that shows the board seeds: each queen’s cell is assigned a unique color (the rest are white).

📌 **(*instruction*) Insert the second image showing seeded regions**

We store region IDs in a 2D list, initially None except where the queen seeds are placed.

cython

复制编辑

cpdef list generate\_regions(int n, list queen\_solution):

cdef list regions = [[None for j in range(n)] for i in range(n)]

# Shuffle region IDs so each puzzle is distinct

cdef list region\_ids = list(range(n))

random.shuffle(region\_ids)

# 1) Assign each queen’s row/column as an initial seed

for i in range(n):

rid = region\_ids[i]

r\_seed = i

c\_seed = queen\_solution[i]

regions[r\_seed][c\_seed] = rid

...

return regions

At this point, we have *n* seeds—one for each queen—and *n* region IDs.

**Step 3: Expand Regions Using Flood Fill**

Now we **grow** these seeded regions using a multi-source flood fill. Each region attempts to expand into adjacent, uncolored cells, with a *random probability* so that the final shapes vary.

📌 **(*instruction*) Insert the third image showing the Flood Fill process**

Here is the relevant code snippet (in Cython, from the same file):

cython

复制编辑

# Step 2: Each region chooses a random expansion probability in [0.3, 0.5]

for rid in region\_seeds:

expansion\_prob[rid] = 0.3 + 0.2 \* random.random()

# Multi-source queue approach

for rid, seeds in region\_seeds.items():

for pos in seeds:

queue.append((pos[0], pos[1], rid))

while queue:

i, j, current\_rid = queue.popleft()

# Check adjacent cells

for (di, dj) in [(-1,0),(1,0),(0,-1),(0,1)]:

ni, nj = i+di, j+dj

if 0 <= ni < n and 0 <= nj < n:

if regions[ni][nj] is None:

if random.random() < expansion\_prob[current\_rid]:

regions[ni][nj] = current\_rid

queue.append((ni, nj, current\_rid))

* We store a random expansion\_prob[rid] for each region ID in [0.3, 0.5].
* In BFS fashion, each region tries to “capture” neighboring cells.

The result is a **colorful mosaic** where each region is connected, but the region sizes and boundaries vary from puzzle to puzzle.

Finally, we fill any leftover None cells by borrowing from adjacent region IDs, ensuring no cell remains uncolored.

**Step 4: Validate the Puzzle for Uniqueness and Difficulty**

At this point, we have an **n-Queens arrangement** *and* a **color partition** that demands exactly one queen per region. But how do we ensure the puzzle has a single solution or at least a limited number of solutions?

We do so by:

1. **Checking region connectivity** (no isolated pockets).
2. **Running a specialized solver** to count solutions up to some threshold (e.g., 1 or 2). If more solutions are found, we discard or re-generate.

Below is the final board with *one queen per row, column, and region*. Notice each color region has exactly one queen.

📌 **(*instruction*) Insert the fourth image showing the completed puzzle board**

We rely on a **Cython + NumPy** backtracking solver (abridged from solver\_cy.pyx) to verify solution counts:

cython

复制编辑

def count\_valid\_solutions(object color\_grid, int threshold):

cdef int n = len(color\_grid)

cdef np.ndarray[DTYPE\_t, ndim=2] arr = np.array(color\_grid, dtype=np.intc)

...

with nogil:

result = \_backtrack(0, n, 0, -100, grid\_arr, threshold, used\_blocks)

return result

* We remove the Python GIL (with nogil) to maximize parallel performance.
* If result exceeds threshold, we stop exploring. This ensures we don’t waste time enumerating all solutions once we know the puzzle is “too easy.”

If the puzzle passes validation (region connectivity, limited solution count), we deem it a **playable puzzle**.

**Performance Challenges and Optimizations**

Generating smaller puzzles (e.g., 4×4 up to 8×8) is straightforward. However, once you aim for **n ≥ 14**, it can take *days* to find enough unique puzzles that pass the single-solution test. Here’s how we tackle performance bottlenecks:

1. **Cython Speedups**
   * We compiled our backtracking and flood-fill code in Cython.
   * We disabled GIL (nogil) in CPU-intensive loops to run them efficiently in C.
2. **NumPy Integration**
   * By converting color grids to NumPy arrays internally, we reduce overhead in checking adjacency, region membership, etc.
3. **Multiprocessing**
   * Puzzle generation can happen in parallel. We spawn multiple processes, each attempts to build a puzzle, and we combine unique ones that meet the difficulty criteria.
4. **Pruning**
   * Our solver stops the search early if it exceeds a solution threshold.
   * The flood fill uses randomness, which helps us avoid exploring large swaths of similar puzzles.

Despite these optimizations, **n=14** and above remains computationally expensive. It’s still possible—but be prepared for *long* generation times!

**Code Examples**

Below are a few concise examples showing how we integrated Python and Cython:

**1. Flood Fill in Cython**

cython

复制编辑

from collections import deque

cpdef list generate\_regions(int n, list queen\_solution):

cdef list regions = [[None for j in range(n)] for i in range(n)]

...

queue = deque()

# ... Multi-source BFS in Cython ...

return regions

**2. Backtracking with Cython “nogil”**

cython

复制编辑

cdef int \_backtrack(int row, int n, uint32\_t used\_cols, ... ) nogil:

# Main logic for row-wise queen placement

# 1) If row == n, return 1 solution

# 2) For each col, check adjacency, region usage, etc.

# 3) Recurse and prune if solutions exceed threshold

...

**3. Parallel Generation**

python

复制编辑

import multiprocessing

def worker\_generate\_map(n):

new\_map = generate\_map(n)

# Solve to check the puzzle’s solution count <= threshold

solution = solve(new\_map["colorGrid"], new\_map["name"], threshold=1)

if solution["number\_solution"] <= 1:

return new\_map

return None

if \_\_name\_\_ == '\_\_main\_\_':

pool = multiprocessing.Pool()

results = pool.map(worker\_generate\_map, [n] \* batch\_size)

...

**Python vs. Cython Performance**

* **Pure Python** backtracking is often *tens to hundreds of times* slower for large n.
* **Cython** with nogil can drastically improve throughput, especially when combined with multiprocessing.
* Memory layouts (e.g., contiguous NumPy arrays) also reduce overhead.

**Real-World Applications in Game Design**

The technique of **randomly generating a valid puzzle, then layering constraints** is widely used in puzzle and game design. Notable examples include:

* **Sudoku Generation** – Randomly assign digits, then prune puzzles to ensure they have a unique solution.
* **Maze Generation** – Carve passages via random DFS or randomized Kruskal’s algorithm, then ensure the maze structure is solvable.
* **AI-Driven Puzzle Creation** – Use search or machine learning to produce “fun” puzzle variants at scale.

**Future Directions**

* **Machine Learning for Difficulty Tuning:** Train a model that predicts puzzle difficulty from partial constraints, then “reject” easy or overly hard puzzles mid-generation.
* **“Trap Puzzles”**: Generate boards that appear solvable but have no valid solutions once certain constraints are applied—players must discover a *contradiction.*

**Conclusion**

Designing a puzzle game from the n-Queens problem requires more than just finding *a* solution—it demands a robust **procedural generation** pipeline that ensures uniqueness, difficulty control, and an engaging experience for players. Our approach incorporates:

* **Randomized n-Queens backtracking** to diversify solutions.
* **Region-based constraints** enforced by flood fill.
* **Cython + NumPy + multiprocessing** for performance.
* **Validation** to ensure each puzzle is solvable (but not too solvable!).

If you’re intrigued by how AI or advanced heuristics might push puzzle-generation boundaries, or if you have feedback on these algorithms, **let’s connect**. This is an exciting space where puzzle-solving meets code optimization and game design innovation.

**Thank you for reading**, and I look forward to any discussions, pull requests, or collaborations!