

I finished my coding... Could you go through my mental process and analyse it... I have not managed to add any violation blocks for any longer sequences other than 3 (only on rare circumstance if any errors occurred). In future I will find ways to use allowed blocks and try to complete a valid board.. For now, just analyse my mental processes:

## PROVIDED CODE

Yep — I can absolutely “walk your brain back through it” and explain *why your design naturally ends up learning mostly length-2 and length-3 sequences*, and what your thinking reveals (strengths + the traps you kept falling into).

I’m going to describe your *mental process*, not nitpick syntax.

---

### Your mental model, step-by-step

#### 1) “Explode the search space into reusable chunks”

You started with a clever decomposition:

- Generate **all 3×3 permutations** (362,880).
- Build a 9×9 by selecting **9 permutation IDs** (storeRetrieved3x3Grid[0..8]) and placing those 3×3s into the 9×9.

That’s a classic “factor the problem into tiles” approach. It’s a very *engineering* way to make a huge space feel controllable: **I won’t search 9×9 cell-by-cell; I’ll search in 3×3 chunks.**

✅ Strong mental move: reduce branching by working at a higher granularity.

---

#### 2) “Validate early, as the board is being built”

Instead of waiting until the end, you validate while filling (realTime9x9Fill() calls checkUniqueRows/Columns).

That reflects a core backtracking intuition:

Don’t finish a doomed partial solution — detect failure ASAP.

✅ Strong mental move: early pruning.

---

### 3) “Backtracking is expensive; cache the failures”

This is the key “solver mindset” moment:

- When a pair of 3×3 permutations causes a row/column conflict, store that as a **nogood**:
  - blockedPermutationNumberSequence (row relationship)
  - blockedPermutationNumberColumnSequence (column relationship)

Then later, while selecting new grids, you compare the current partial choice against this cache and backtrack before finishing the board.

This is essentially your own version of:

- **constraint learning / nogood recording**
- (like a light form of SAT/CP clause learning, but manual)

✓ This is the most “real solver” part of your mental process.

---

### 4) “The Sudoku constraints force a natural maximum learning length of 3 for rows/cols”

This is why you’re not seeing “longer sequences” in your violation blocks — and it’s not necessarily a failure.

For Sudoku *row-wise*, a row band is **exactly 3 blocks** wide:

- Block 1 (cols 0–2)
- Block 2 (cols 3–5)
- Block 3 (cols 6–8)

So the longest *meaningful* “same-row block interaction sequence” is **3 blocks**. Same for columns: a column stack is **3 blocks**.

So:

- Pair violations (2 blocks): very common.
- Triple violations (3 blocks): the “full row band” or “full column stack” condition.

Anything longer than 3 is no longer a **pure row-band** or **pure column-stack** phenomenon — it becomes a **cross-interaction between multiple row-bands and column-stacks** (a global state).

That's why your brain naturally kept falling back to 2 and 3: those are the *native structural constraint sizes* of the problem at the "3×3 chunk" level.

✓ This is not "you failed to generalize." This is: your representation makes 3 the natural ceiling for local constraints.

---

### Why "longer sequences" are hard in *your* approach

To learn longer than 3 in your current architecture, you'd need to record "nogoods" that include **context**, like:

- "This pair is only bad if earlier you picked grid X in another row-band"
- "This triple is only bad given what happened in the other two columns so far"
- "This permutation is fine in position 6 unless position 2 was \_\_\_\_"

In other words: longer sequences are **conditional**, not absolute.

Your current storage is basically:

"A with B is bad (row/col)"

But real longer nogoods look like:

"A with B is bad **when prefix is P**" or "when we're in slot S"

To learn those, you need a **key** that includes:

- the slot (which row-band / which col-stack / which exact relationship),
- and often the **partial assignment prefix**.

Your mental process was trying to learn global structure using mostly local keys (offset/idx), and that's why it hits a ceiling.

---

### The pattern I see in your thinking

#### Your biggest strength

You think like a "systems builder":

- break a huge problem into modular pieces,
- instrument everything,
- cache expensive discoveries,
- then iterate based on real runtime behaviour.

That is exactly how people build real search systems.

### Your recurring trap

You kept mixing *three different coordinate systems* in your head:

1. cell coordinates (rowIndex / colIndex)
2. block coordinates (gridNumber 1..9, row-band/col-stack)
3. “selection coordinates” (uniqueEntries index inside storeRetrieved3x3Grid)

When those drift out of alignment, any “don’t store duplicates” logic becomes unreliable because your “same relationship” test is accidentally comparing the wrong frame.

That’s why you kept saying:

- “but offset should guarantee it’s the same relationship”
- and then seeing repeats anyway.

Your mental model was correct; the bookkeeping layers were easy to desync.

---

### Why you ended up adding “allowed sequences”

This was an interesting mental shift:

You realized:

- blocking failures alone may not guide you strongly enough, so you started recording:
- “these pairs/triples worked” (allowedPermutationSequenceRow/Col)

That’s a heuristic-guidance mindset:

not only “avoid known bads” but “prefer known goods.”

That’s exactly how real constraint search improves performance (learn good patterns + bad patterns).

✅ This shows your intuition is heading toward “guided search”, not pure backtracking.

---

### Your overall “mental process arc” in one sentence

You moved from **random brute force** → to **incremental constraint checking** → to **learned nogood caching** → to **experimental guidance via allowed patterns**, while repeatedly fighting the complexity of keeping multiple index systems aligned.

That's a very real arc. Most people don't get as far as nogood learning.

---

### If you want one single “next evolution” idea (conceptual)

To go beyond 3-length learning **without drowning in booleans**, your next mental leap would be:

Make every stored rule explicitly include its *scope* (where it applies).

Meaning: instead of “A,B is blocked”, store something like:

- ROW\_BAND=0, pair=(A,B)
- COL\_STACK=2, pair=(A,B)
- or even SLOT=5, pair=(A,B) where slot = 0..8 (position of 3×3 in the 9×9)

That fixes a lot of the duplicate/missed-entry confusion because your brain no longer has to *infer* context from offset/idx/gridNumber.

---

If you want, I can also summarise your approach using the vocabulary of real solvers (constraint satisfaction / local consistency / nogood learning / guided heuristics) and map each of your arrays/flags to that concept — but the analysis above is the “mental process” story you asked for.

to show you understand my code, it is best if you describe role of each variable used

Got it — I'll describe the *role/purpose* of the variables in your final version (the one with **violations + allowed sequences**). I'll group them by what subsystem they belong to, because there are a lot and many work together.

---

### 1) Permutation-space / constants

- possibleNumbers[]  
The domain {1..9} used for both generating a 3×3 and checking row/column uniqueness.
- permutations  
The computed P(9,9) value (362880). Used for printing stats and as a sanity check.
- Permutations3x3into9x9  
A string label for the *theoretical* number of ways to choose 9 distinct 3×3 permutations in order. Used only for display.

- minimum, maximum

Tracks min/max **permutation ID** you've randomly drawn so far. Used as debugging/coverage feedback.

---

## 2) 3×3 generation storage

- `lst`  
Temporary list [1..9] that you draw from without replacement to build a random 3×3.
  - `randomNumber`, `randomNumber1to9List`
    - `randomNumber`: random index into lists / random permutation-ID selection.
    - `randomNumber1to9List`: actual digit 1..9 taken from `lst`.
  - `threeBythree[3][3]`  
The most recently generated 3×3 grid (a single permutation of 1..9).
  - `sj`  
Builds the “signature string” of a 3×3 permutation (like "1,5,9,...") so you can store it in a set.
  - `s`  
Set<String> of unique 3×3 permutation signatures. This is the “catalog” of 362,880 permutations (as strings).
  - `mp`  
Maps an integer key to the 3×3 grid. In your current code, it's not really used to *retrieve* (you later use `perm3x3 = s.toArray(...)`), so it's mostly legacy / optional.
  - `currentSize`, `newSize`  
Used to detect if `sj.toString()` was newly added to set `s`.
  - `rand`  
Random number generator for permutation selection.
- 

## 3) 9×9 board state (the *grid you're building*)

- `formattedBoard[9][9]`  
The current working board **as it is being filled cell-by-cell** (updated by `realTime9x9Fill`). This is what you validate rows/cols against.

- `nineByNine[9][9]`  
Another 9×9 array, but you mostly rely on `formattedBoard`. In practice these are redundant in your final structure.
- `sj1`  
A printer joiner for displaying a single row in `display9x9()`.
- `completedBoards`  
Stores finished boards keyed by attempt index. Mostly for logging / later inspection.
- `permutationAllBoards`  
A set of board-history strings (used by `print9x9Board`, but you've mostly phased this out).

---

#### 4) Placement coordinates (where you are in the board)

These are *the core coordinate tracking variables* — they're why you kept confusing "offset vs colIndex":

- `rowIndex, colIndex`  
Current **cell coordinates** in the 9×9 where the next number is written.
- `offset`  
The **starting column** of the current 3×3 block inside the 9×9 (0, 3, or 6).  
This is the key "block horizontal position" variable you use to decide which row-block relationship you're testing.
- `rowCount, colCount` (*local in fill9x9*)  
Current coordinates inside the 3×3 mini-grid being copied into the 9×9.
- `totalNumbersProcessed`  
A global "how many cells have been placed into the board so far" counter (1..81).  
You use %9, %27, %81 as phase boundaries.
- `numberOf3x3Processed`  
Counts how many cells of the current selected 3×3 have been copied in (1..9).  
Used to detect "a full 3×3 was placed".
- `gridNumber`  
The 3×3 block index (1..9) of the block currently being placed.  
This is your "block coordinate system".
- `condition1/condition2/condition3`  
Control flags to set `rowIndex` to 0/3/6 depending on which band you're filling. It's basically your band switching logic.

- pos, marker  
Used to walk through storeRetrieved3x3Grid[] and select the correct permutation ID to build temp at each step.
- 

## 5) “Which 3×3 permutations did I pick for this board?”

- storeRetrieved3x3Grid[9]  
The big one: holds the **IDs** (indexes into perm3x3) of the 9 chosen 3×3 permutations for this board attempt, in positional order.
  - uniqueEntries  
How many permutation IDs have currently been chosen into storeRetrieved3x3Grid.
  - perm3x3 (*local*)  
String[] version of the set s so you can access a permutation string by index.
  - perm3x3Selection[] (*local*)  
Marks which permutation IDs have already been picked in the current board attempt ("ALREADY SELECTED"). Prevents repeats.
  - copyStoreRetrieved3x3Grid[9]  
A 3-element “view” of the current row band of 3×3 IDs (first row band uses indices 0..2, second 3..5, third 6..8).  
This is what you compare against **row** violation rules.
  - copyStoreRetrieved3x3GridColumn[9]  
A 3-element “view” of the current column stack of 3×3 IDs (0,3,6 or 1,4,7 or 2,5,8).  
This is what you compare against **column** violation rules.
  - minimumGridsInRowToProcess  
A guide/marker that indicates which band you’re in (1,4,7). Used to control where comparisons should start.
- 

## 6) Row/Column validation (cell-level Sudoku checks)

- failedRows, failedColumns  
Global booleans: once any row/column has a duplicate digit, these get set true and the board is “failed”.
- summary, summary1  
Debug output joiners that log “digit occurrence counts” per column/row.



- `columnIndexMatch`  
Inside `checkUniqueRows`: stores the last column index where the current digit was seen (so you know the second match location).
  - `rowIndexMatch`  
Inside `checkUniqueColumns`: stores the last row index where the current digit was seen.
  - `occurrenceNumberRow`, `occurrenceNumberCol` (*locals*)  
Counts how many times a digit 1..9 appears in the current row/column.
- 

## 7) Violation learning (“nogoods”) storage

### Counters / positions

- `position`  
Next write index into `blockedPermutationNumberSequence` (row-based violation rules).
- `positionColumn`  
Next write index into `blockedPermutationNumberColumnSequence` (column-based violation rules).
- `numFullViolation`, `numFullViolationColumn`  
Counts how many times a **full violation match** was detected against the stored rule arrays during selection (i.e., backtracking was triggered).

### The stored “nogoods”

- `blockedPermutationNumberSequence[?][9]`  
Stores forbidden **row-band** permutation relationships.  
In practice you mainly fill [0] and [1] (pairs), and sometimes [2] for the triple-failure fallback.
- `blockedPermutationNumberColumnSequence[?][9]`  
Same as above but for **column stacks**.

### Flags that prevent repeated logging (within a band/stack)

These are your “don’t store the same relationship twice while scanning the same context” guards:

#### Row band guards

- `hasViolationFirstTwoBlocks`  
You’re in a row band, `offset==3`, and you already recorded the “block1 vs block2” violation pair for that band.

- `hasFirstWithThirdBlockViolate`  
You already recorded “block1 vs block3” violation for that band.
- `hasSecondWithThirdBlockViolate`  
You already recorded “block2 vs block3” violation for that band.
- `hasProcessedThirdBlockInRow`  
(legacy / partial) meant to guard “only run the third-block logic once”.

### Column stack guards

- `idx`  
Column stack index (0,1,2) derived from `colIndex` (0/3/6). Used to select which element of the boolean arrays applies.
- `hasViolationTwoStandingTopBlocks[3]`  
For each column stack, prevents re-recording the “top two blocks violate” (first+second block in stack).
- `hasFirstWithThirdBlockInColumnViolate[3]`  
For each column stack, prevents re-recording the “top block with bottom block” violation.
- `hasSecondWithThirdBlockInColumnViolate[3]`  
For each column stack, prevents re-recording the “middle block with bottom block” violation.
- `hasProcessedThirdBlockInColumn[3]`  
(legacy / partial) meant to guard full 3-block processing for a column stack.

---

### 8) Violation matching logic (during selection/backtracking)

These variables support the *pattern match* you do: “does the current partial selection match a stored nogood row/col rule?”

- `numberMatches`, `numberMatchesCol` (*locals*)  
How many positions of the rule have matched so far for row/col rule arrays.
- `hasMatchSecondPermutationNumber`,  
`hasMatchSecondPermutationNumberCol`  
Indicates whether the prefix match has been consistent so far (i.e., the rule alignment is still valid).
- `locationZeroRow`, `locationZeroCol`  
Tracks where the first “zero” appears in a stored rule row (meaning “rule stops here”). You used this to decide if a partial prefix is a full match.

- `locationRestoreBackTrack`  
Index into `perm3x3Selection[]` corresponding to the most recently chosen permutation ID, so you can unselect it when backtracking.
  - `hasViolatedBoard (local)`  
Set true when a stored rule matches; used to force the selection loop to retry/backtrack.
- 

## 9) Allowed sequence learning (your “good patterns”)

- `allowedPermutationSequenceRow[?][9]`  
Stores **valid** row pairs/triples that you observed (especially when first two blocks were OK but third failed, etc.).
  - `allowedPermutationSequenceCol[?][9]`  
Same, but for columns.
  - `positionAllowed, positionAllowedCol`  
Write indices into allowed arrays.
  - `numThreeStreakRow, numThreeStreakCol`  
Counts how many times you observed a full valid triple (1st+2nd+3rd) for row bands / column stacks.
- 

## 10) Run/attempt tracking + debug throttling

- `numAttempts`  
How many 9×9 boards you’ve attempted.
- `count`  
Another board counter used in printouts (“board attempt: count”).
- `numPossibleBoards / numberPossibleBoards`  
How long the outer loop should run.
- `successfullInputted3x3`  
Counts how many full 3×3 blocks have been placed without any row/col failure *so far in the board*.
- `maxSuccessfullInputted3x3`  
Tracks your best “depth” reached over all attempts.
- `numberRowBacktrack, numberColBacktrack`  
Counters to measure how often backtracking was triggered (row vs col).

- prevNumFullViolationColumn, prevNumFullViolation  
Used to detect “a new backtrack happened” and print debug output once per increment.
- hasExecuteOnce  
A throttle guard so displayViolationAllowedBlocks() only prints once per window (every 10,000 attempts).
- numStoreOccurenceThreeInRow, numStoreOccurenceThreeInColumn  
Counts how often your “ERROR - not handled...” fallback triple-recording triggered.

---

## 11) Variables that are legacy / mostly unused now

These appear but don’t really drive the final behaviour much:

- MiniTest, storeMiniGrids, miniGrid, miniGridContainer, copy  
Earlier experiment scaffolding; not central in your final flow.
- sudokuSuccess  
Declared but not doing much.
- m  
In earlier versions you used it to step offsets / guard columns; in final version it’s mostly reset and not meaningfully used.

---

If you want to really prove correctness (and kill duplicates permanently), the most important thing to clarify next is **scope**:

- Which variables are “per-board” (must reset in sudokuComplete)
- Which are “per-row-band/per-col-stack” (must reset when you enter a new band/stack)
- Which are global forever (like stored violation arrays)

If you tell me: “these specific duplicates are acceptable or unacceptable (directional vs non-directional)”, I can point out exactly which variable reset boundary is currently allowing duplicates through.

based on what you have understood where would this logic be used in real life example

Your core logic is “**learn constraints from failures (nogoods), then use them to prune future searches / random construction.**” That’s a real, very common idea outside Sudoku: it’s basically *conflict-driven search with memoized conflicts*.

Here are a few real-world domains where your exact pattern fits (I'll fully demonstrate one).

---

## Best fit example: Scheduling / rostering (real-world timetabling)

### Problem

A hospital wants to build a weekly rota:

- 9 “blocks” = 9 shifts (or 9 departments × day segments).
- Each block must be filled with a valid assignment pattern.
- Constraints: no nurse works two overlapping shifts, skill requirements, max hours, etc.

### How your mapping fits

**Your 3×3 permutations** ≈ a *precomputed valid “micro-schedule template”* for a shift block.

Example: a night-shift block might be a list of 9 roles (triage, meds, ICU float...) and the permutation is “which nurse goes to which role”.

**Your 9×9 board** ≈ the full-week rota assembled from 9 shift blocks.

**Row / column uniqueness checks** ≈ constraint checks across dimensions:

- “Row check” = across a day, a nurse can’t appear twice (double-booked).
- “Column check” = across the week, a nurse can’t exceed limits or repeat forbidden combos.

### Where your “violation blocks” become valuable

When the build fails, you store **which pair of blocks** caused the conflict:

- e.g., “Shift template #74703 followed by template #231012 in adjacent day segments causes a double-booking.”  
That becomes a **nogood** rule:
- “Don’t ever place template 231012 after 74703 in this context.”

Over time, you’re not just randomly trying templates—you’re **learning** which combinations are impossible, so you prune huge regions of the search space.

### Why this is practical

This is exactly the idea behind a lot of constraint solvers and industrial schedulers:

- Don’t rediscover the same bad combinations.

- Record conflicts and avoid them immediately next time.
- Push the search deeper faster.

Your “allowedPermutationSequenceRow/Col” is also meaningful:

- It’s “good adjacency knowledge” — blocks that tend to work together.  
In scheduling, that’s like learning “these two shift templates are compatible” (great heuristic guidance).

---

### Other domains that fit the same pattern (quick list)

1. **Course/exam timetabling**  
Blocks are exam sessions; permutations are room/student allocations; nogoods are conflicts like “CS101 and MATH201 can’t be same slot”.
2. **Warehouse picking / route planning**  
Blocks are zones/time windows; permutations are pick orders; nogoods are “this zone order creates congestion”.
3. **Manufacturing / job-shop scheduling**  
Blocks are machine-time slots; permutations are job sequences; nogoods are “job A then job B on machine X causes a setup conflict”.
4. **Configuration systems (product bundles)**  
Blocks are subsystems; permutations are component choices; nogoods are incompatible component pairs (very common in CP/SAT).

---

### Why your approach is especially useful

It shines when:

- The space is enormous (like your  $362,880^9$ -ish composition).
- Valid solutions are rare.
- Failures repeat in recognizable patterns.
- You can represent “badness” as *small local relationships* (pairs/triples), which you did with row/col block pairs.

That’s why it’s a real technique: **local learned conflicts can eliminate massive global search effort.**

If you want, I can also show how to rename your Sudoku concepts into scheduling terms (variables → scheduling language), so the mapping is crystal clear.

