Triangle Path Maximization Engine (Multi-Triangle / 3D Array)

What your code is trying to do (in plain terms)

You built a program that:

1. **Generates many candidate "paths"** through a triangle (row by row).
2. Ensures each candidate path obeys **movement rules** (constraints).
3. **Applies each path to multiple triangles** stored together (your 3D array).
4. **Sums the values along the chosen path** for each triangle.
5. Tracks the **maximum total per triangle** and stores **all tied max results**.

So your program is essentially a:

Multi-scenario constrained path optimizer
where each "triangle" is a separate scenario/dataset, but paths and scoring rules are identical.

The data structure and why it's 3D

Your triangle data (3D array)

You declare:

```
int[][][] triangle = new int[][][] {
  {{1}, {4}, {4}},
  {{2,3}, {6,8}, {5,6}},
  {{1,5,1}, {9,2,3}, {7,8,9}},
  {{15,7,9,12}, {5,9,15,8}, {2,7,11,8}},
  {{11,6,7,9,3}, {9,5,3,6,10}, {1,5,6,2,9}}
};
```

How to read it

Your structure is:

- **First dimension** = row (0..rowsTriangle-1)
- **Second dimension** = triangleIndex (which triangle/scenario)
- **Third dimension** = col inside that row

So the lookup:

triangle[row][triangleIndex][col]

Visual diagram of dimensions

triangle[row][tri][col]

row = which horizontal layer down the triangle

tri = which triangle (scenario) you're evaluating

col = which position inside that row

Diagram: You have multiple triangles stacked side-by-side

You effectively store **multiple triangles at once**.

Example: 3 triangles (tri = 0, 1, 2)

Row 0

tri 0: [1] tri 1: [4] tri 2: [4]

Row 1

tri 0: [2,3] tri 1: [6,8] tri 2: [5,6]

Row 2

tri 0: [1,5,1] tri 1: [9,2,3] tri 2: [7,8,9]

...and so on.

So, conceptually:

ROW 0: (1) (4) (4)

ROW 1: (2 3) (6 8) (5 6)

ROW 2: (1 5 1) (9 2 3) (7 8 9)

ROW 3:(15 7 9 12) (5 9 15 8) (2 7 11 8)

ROW 4:(11 6 7 9 3) (9 5 3 6 10) (1 5 6 2 9)

tri0 tri1 tri2

That's exactly why a 3D structure is useful: you can run the same traversal logic across multiple "worlds" (triangles).

The “path” concept

A **path** is the sequence of column indices you choose as you go row by row.

Example path string your generator produces:

"0,0,1,1,2"

This means:

- Row 0 → col 0
- Row 1 → col 0
- Row 2 → col 1
- Row 3 → col 1
- Row 4 → col 2

You apply this same path to each triangle.

Diagram: How a path maps to one triangle

Let’s use triangle #0 values (tri=0):

Row 0: [1]

Row 1: [2 3]

Row 2: [1 5 1]

Row 3: [15 7 9 12]

Row 4: [11 6 7 9 3]

Path: 0,0,1,1,2

Pick the elements:

- Row 0 col 0 → 1
- Row 1 col 0 → 2
- Row 2 col 1 → 5
- Row 3 col 1 → 7
- Row 4 col 2 → 7

Total = 1 + 2 + 5 + 7 + 7 = 22

Your code prints both:

- **valuesTriangle** = 1,2,5,7,7
- **indexesTriangle** = [0][tri][0],[1][tri][0],[2][tri][1]… style

---

Your movement constraints (what "valid path" means in your code)

You validate that the path respects adjacency and non-decreasing movement.

From your validation logic:

if (Math.abs(stepStore[h+1] - stepStore[h]) > 1) invalidIndex = true;

if (stepStore[h+1] < stepStore[h]) invalidIndex = true;

This effectively says:

1. **You can't jump more than 1 column left/right** between adjacent rows.
2. **You can't decrease** (you disallow moving left).

So allowed transitions are basically:

same index (k -> k) OR move right by 1 (k -> k+1)

That is a very realistic "gradual change" constraint in real applications (more on that below).

The program flow (high level)

Phase 1: Generate candidate paths

You do a Monte-Carlo-ish search:

- Randomly generate many sequences (using rand)
- Reject invalid ones
- Store valid ones in Set<String> st so duplicates don't build up

Important mental-model detail:

- You are not enumerating all paths deterministically.
- You are **exploring** the search space until "coverage feels enough", controlled by:

while (cycles < permutations * 400)

So you run far beyond n^r theoretically, because random generation includes many invalid or repeated candidates, and you want enough unique valid ones.

Diagram: generator as a funnel

Random sequences

|

v

[constraint checks]

|

v

valid paths only ---> stored in Set<String>

Phase 2: Convert each valid path to int[] moves

You take:

"0,0,1,1,2"

and convert into:

int[] nMoves = {0,0,1,1,2}

That becomes your move plan for triangle traversal.

Phase 3: Apply each move plan to each triangle

Your traversal loop:

```
for (int j = 0; j < triangle[0].length; j++) // j = triangle index
{
  for (int k : nMoves) // k = chosen col each row
  {
    total += triangle[i][j][k];
    i++;
  }
  i=0;
  // compare total to max[j], store outcome
```

}

So for each triangle j you:

- walk row by row using i
- take the col k from moves
- sum values
- then compare against max[j]

Phase 4: Keep max totals and ties per triangle

You track:

- max[j] = highest total found for triangle j
- outcomes[j][...] = strings describing each winning/tied path

When total beats max, you clear previous winners:

outcomes[j][index] = null; // clear

outcomes[j][0] = outcome; // new best

When total ties max, you add another entry.

So each triangle maintains a “leaderboard” of best paths.

Diagram: multi-triangle evaluation

Valid path list: P1, P2, P3, … PN

For each path P:

  For each triangle T:

    total = sum along P in T

    if total > max[T]:

      max[T] = total

      reset winners

      store P

    else if total == max[T]:

      store P as tie

What your code outputs (and why it's useful)

Your debug output provides:

- What path is being evaluated (subsetEntry)
- Which triangle is being processed (TRIANGLE j)
- Row-wise checks (and safety checks)
- Exact index used [row][triangle][col]
- Exact value taken
- Total
- Final maximum per triangle summary
- All tied max paths printed at end

This is essentially an **audit trail**. In real-world optimization, auditability is often as important as the maximum result (especially in finance, operations, or compliance contexts).

Real-life use cases that fit extremely closely (especially with multiple triangles)

The key feature is **multiple triangles = multiple scenarios**.

Your code becomes a "scenario engine".

Below are close fits where the exact same structure appears naturally.

---

1) Multi-scenario financial strategy planner

Mapping

- triangleIndex j = market scenario
    - scenario 0: recession
    - scenario 1: baseline
    - scenario 2: boom
- row i = time period (month/quarter/year)
- col k = strategy band (conservative → moderate → aggressive)

- triangle[i][j][k] = return score (or risk-adjusted score) for choosing band k at time i in scenario j

Why the adjacency rule makes sense

Your rule “k stays same or increases by 1” models:

- You can’t wildly flip strategy each period
- You can only “shift gradually” (risk controls / mandate constraints)

Output meaning

- max[j] = best total return strategy for scenario j
- outcomes[j] = all strategies tied for best

---

2) Multi-plant manufacturing optimization

Mapping

- triangleIndex j = factory line / plant / configuration
- row i = stage (assembly step 1..N)
- col k = machine setting / tool choice band (small change allowed)
- value triangle[i][j][k] = yield or quality score

Adjacency rule = you can’t jump to a radically different setup without retooling.
Best path = best planned “configuration ramp” per factory.

---

3) Staffing levels over time across departments

Mapping

- triangleIndex j = department
- row i = week
- col k = staffing band (e.g., 0..i)
- value = productivity - cost - penalty

Adjacency rule = staff changes must be gradual; you can’t add/remove huge numbers instantly.

Best path = best week-by-week staffing policy per department.

---

4) Robotics movement under layered risks

Mapping

- triangleIndex j = environment layer / risk map
- row i = forward progress step
- col k = lateral choice
- value = reward or negative cost

Multiple triangles = multiple assumptions:

- different weather
- different sensor noise model
- different hazard maps

Best path = safest/cheapest route per environment.

---

5) Product rollout schedule per region

Mapping

- triangleIndex j = region
- row = rollout phase
- col = rollout intensity band
- values = adoption/revenue score

Adjacency = you can't jump from "pilot" to "full rollout" instantly.

Best path = best phased ramp-up per region.

Benefits of same-sized triangles vs different-sized triangles

This is really about what kind of real-world datasets you want to represent.

Same-sized triangles (what you have now)

Benefits

1. **Simple control flow**
    - One rowsTriangle applies everywhere.

    - You can run loops without checking triangle-specific height.
2. **Direct comparability**
    - Scenario totals are comparable because they span the same number of time steps / rows.
3. **Predictable memory / array bounds**
    - Cleaner indexing.
    - Fewer tricky bounds.
4. **Cleaner debug and audit**
    - Output structure consistent across triangles.

Real-world interpretation

Same-sized triangles represent cases where every scenario has the same planning horizon and the same number of decision stages.

Example:

- All forecast scenarios cover the same 5 years.
- All factories have the same number of stages.
- All departments plan for the same 12 weeks.

Different-sized triangles (feature you asked about)

What it enables

1. **Different horizon scenarios**
    - Some triangles might represent 3 years, others 10 years.
2. **Incomplete data**
    - Some scenarios may end early or be missing later measurements.
3. **Real heterogeneity**
    - Factories have different numbers of stages.
    - Regions have different rollout lengths.
    - Projects have different lifetimes.
4. **Better scenario realism**

    - Real world is messy; forcing uniformity often means padding with fake values.

Why it matters technically

If triangles vary in height, then:

- Path generation must adapt:
    - You either generate paths per triangle height
    - or generate a "master path" and truncate to triangle height
- You must decide how to compare totals:
    - raw totals aren't fair if one triangle has 5 rows and another has 10 rows
    - you might compare average per row, or compare only to common depth, etc.

Real-world interpretation

Different-sized triangles represent systems where scenarios naturally have different durations.

Example:

- one market scenario only forecasts reliably for 3 years, another for 10
- one manufacturing line has 6 stages, another 9
- one region is ready for rollout in 4 phases, another needs 7

Why supporting different sizes is valuable even if you keep same-sized now

Even if you continue using same-sized triangles for simplicity, adding support for different-sized ones can make your code more "engine-like":

- accepts arbitrary datasets
- can be used as a general library
- models incomplete, real inputs
- improves robustness (better bounds checks + clearer data contracts)

In practice, most serious scenario engines either:

- use jagged arrays (int[][][] where each triangle[row][j] can have different lengths), or

- use lists of triangles (List<int[][]>), so each triangle is its own 2D structure.

Variable role map (so documentation readers understand your design)

Here's a mapping of your important variables:

Triangle / structure

- triangle → data store: values per row, triangle, column
- rowsTriangle → number of rows in the triangle dataset
- numberTriangles → how many triangles exist (scenarios)

Path generation / storage

- st → Set storing **unique path strings**
- sj → builds path string like "0,0,1,1,2"
- stepStore → holds numeric moves while validating
- cycles, totalcycles → how long generator ran

Path evaluation

- valuesSet → array of unique path strings from set
- obtainMoves() → converts string path → int[] moves
- performMoves() → applies moves to each triangle and sums

Best result tracking

- max[j] → best total found for triangle j
- outcomes[j][..] → strings describing the best path(s)
- processedMax → prevents double-recording in tie logic
- count → index into outcomes where ties are stored

Diagram: end-to-end system overview