Constraint-Aware 3D Container Loading via Recursive Space Partitioning and Annealed Permutation Search

Onur Gumus
3DPack.ing / GPTPacker.com
April 2026

Abstract

We describe a production orthogonal 3D container loading system developed over several years that combines three techniques which, in our review, we did not find described together in prior container loading work: (1) a recursive space-partition placement that subdivides containers through nested sub-container narrowing, preserving parent-region dimensions for constraint checking, (2) a beam-augmented annealed search over item permutations, where a deterministic placer enforces all constraints internally so that every candidate ordering decodes to a valid placed/unplaced partition with all enforced constraints satisfied for placed items, and (3) a deterministic compaction post-pass that reclaims gravity and slide gaps while preserving a >75% bottom-face support rule for elevated items. On heterogeneous mid-size item sets, cost-function rebalancing improved packing compactness from 0.45 to 0.72 (a 62% gain) with no changes to the placement routine. On a production 40-foot container load of 730 items across two box types, the system achieves 82% volume utilization. The system is deployed at 3DPack.ing and GPTPacker.com.

1. Introduction

1.1 The Standard Formulation

The three-dimensional bin packing problem (3D-BPP) asks: given a rectangular container of dimensions W x H x L and a set of n rectangular items, each with dimensions w_i x h_i x l_i, find a non-overlapping axis-aligned placement of items inside the container that maximizes packed volume. We consider orthogonal 3D container loading with axis-aligned boxes. The problem is NP-hard by reduction from 1D bin packing.

Our work is closer to the *constrained 3D container loading problem* than generic 3D-BPP: items have orientation restrictions, stacking rules, spacing requirements, and placement priorities that interact with the geometric packing.

1.2 The Problem in Practice

The gap between the academic formulation and a real freight loading problem is substantial. A logistics operator loading a 40-foot container faces:

- **Orientation constraints.** Appliances must remain upright (the height axis is fixed). This restricts rotation to horizontal swaps only.
- **Directional placement.** Heavy pallets must sit on the container floor. Fragile items cannot bear weight on top.
- **Stacking limits.** A given item may support at most k items above it, or at most W kg. These limits are per-item.
- **Spacing requirements.** Certain item families need a minimum gap to adjacent items. When two items with gaps are placed next to each other, the effective spacing should be the maximum of their individual requirements, not the sum — a property we call *order-dependent gap collapsing* because the collapsed gap depends on placement sequence.

- **Per-item rotation permissions.** Some items rotate freely; others cannot rotate at all; others rotate horizontally but cannot tilt.
- **Physical stability.** An elevated item must have a sufficient fraction of its bottom face supported by items below. Our system uses a >75% contact area threshold.
- **Priority ordering.** Higher-priority items are placed first and never sacrificed for lower-priority ones when space is limited.

These constraints interact: a floor-pinned item consumes floor area that a no-stacking item also needs (since nothing can be placed above a no-stacking item, it requires its own vertical column). A stacking limit of 2 interacts with rotation permissions to determine which orientations leave room for exactly 2 items above. Gap collapsing interacts with placement order because the nearest-neighbor search that determines gap offsets depends on what has already been placed.

We did not find prior work that models these constraints jointly or analyzes their interactions in deployed freight-loading systems.

1.3 Coordinate System

Throughout this paper:

- **X** = width (horizontal, perpendicular to container length)
- **Y** = height (vertical, gravity axis)
- **Z** = length (depth, along the container's long axis)

The container origin is at (0, 0, 0). Items are axis-aligned rectangular boxes. *MinimizeLength* mode optimizes along Z; *MinimizeHeight* mode optimizes along Y.

1.4 Definitions

We distinguish three packing quality metrics:

- **Utilization** = total placed item volume / container volume. Measures how full the container is.
- **Compactness** = total placed item volume / bounding-box volume of placed items. Measures how dense the packed region is, independent of container size. A value of 1.0 means zero internal gaps.
- **Blank space** = 1 - compactness. The fraction of the packed region's bounding box that is empty.

These are related but independent. A packing can have low utilization (container mostly empty) but high compactness (the packed items are tightly arranged), or vice versa.

1.5 Contributions

1. **Recursive space-partition placement** (Section 3.1). A deterministic placement strategy that maintains nested sub-containers rather than enumerating candidate positions. Each placement narrows a sub-container along the width axis, and later placements in that sub-container inherit the parent's full height and length. This preserves constraint-checking context that disjoint partitioning methods would lose.

2. **Beam-augmented annealed permutation search** (Section 3.2). A search that mutates item orderings rather than item-to-position mappings, augmented with eight concurrent mutation strategies per temperature step. Constraints are enforced inside the deterministic placer; every candidate ordering decodes to a valid placed/unplaced result, so no repair operators are required.

3. **Grounding-aware deterministic compaction** (Section 3.3). A post-pass that drops and slides items toward the container origin, validating each move against a >75% bottom-face support threshold. The pass is monotone (coordinates only decrease) and in our regression suite

substantially reduced run-to-run variance, often converging nearby search outputs to the same final arrangement.

2. Related Work

2.1 Placement Strategies

Corner-point methods (Martello, Pisinger, and Vigo, 2000) enumerate candidate positions at corners formed by placed items and container walls. **Extreme-point methods** (Crainic, Perboli, and Tadei, 2008) refine these by projecting candidates onto existing surfaces. Both generate multiple candidate positions per item and select the best by a scoring function. **Maximal-rectangles methods** maintain a set of maximal empty spaces.

These candidate-enumeration methods are especially effective on unconstrained or lightly constrained instances, where the scoring function can evaluate candidates independently. They become more complex under rich constraint sets, because each candidate must be checked against directional placement rules, stacking limits, rotation permissions, and spacing — and infeasible candidates must be filtered or repaired before scoring.

Our approach generates one candidate position per sub-container (the sub-container's origin) but compensates through search diversity: many item orderings are evaluated, and each ordering produces different sub-container decompositions. This is a practical tradeoff — fewer candidates per item, but simpler constraint enforcement per candidate — discussed in Section 3.1.2.

Our recursive narrowing is related in spirit to guillotine-style partitioning (Lodi, Martello, Vigo, 2002), but differs in topology: the maintained sub-containers are nested rather than disjoint, and later placements retain inherited height and length from the parent region. This distinction helps preserve constraint-checking context (Section 3.1.2).

2.2 Search Strategies

In many meta-heuristic approaches to 3D loading, the search state includes explicit placement decisions — positions, orientations, or local relocations. These approaches can directly optimize spatial relationships and are well-suited to unconstrained or moderately constrained settings. Under richer constraint sets, however, they must maintain feasibility during search, often requiring repair operators or penalty functions when mutations produce constraint violations.

Our search operates primarily in permutation space, closer to the decode-and-evaluate paradigm used in genetic algorithms for scheduling (Bierwirth, 1995). The search mutates item orderings; the deterministic placer handles feasibility. This separation is the key architectural decision that enables the constraint set described in Section 4.

The nested beam search (eight candidates per temperature step, evaluated in parallel) combines elements of beam search (Ow and Morton, 1988) with annealing-style acceptance. We did not find this specific combination in the 3D loading literature, though parallel neighborhood evaluation is standard in meta-heuristic frameworks for other combinatorial problems.

2.3 Post-Processing

Post-placement compaction has been explored for 2D packing (Imahori and Yagiura, 2010) but is uncommon in 3D container loading, partly because defining "compaction" in 3D requires a stability model. We are not aware of prior 3D post-compaction work that couples each move to an explicit support-area stability test of the kind used here.

3. Approach

3.1 Recursive Space-Partition Placement

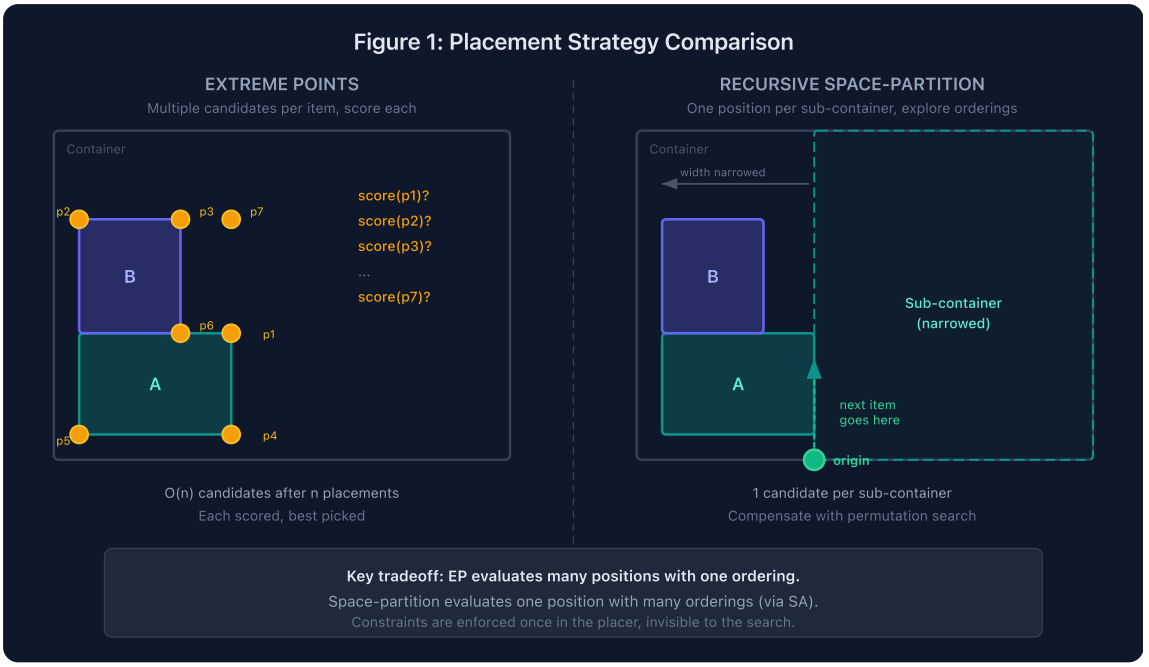
3.1.1 Mechanism

The placement state is a list of *sub-containers*, each defined by an origin (X, Y, Z) and dimensions (W, H, L). Initially, the list contains a single entry: the root container. Items are processed sequentially in the order determined by the search.

For each item, the routine scans sub-containers in a mode-dependent order (Z-ascending for MinimizeLength, Y-ascending for MinimizeHeight) and attempts placement at each sub-container's origin. If the item fits (after applying gap offsets, rotation selection, and constraint checks), it is placed, and the sub-container is *narrowed*: its width dimension W is reduced by the item's effective width, and its X-coordinate is shifted accordingly. The narrowed sub-container replaces the original in the list.

After each placement, the sub-container list is merged (adjacent sub-containers with compatible dimensions are combined) and re-sorted.

```
PLACEMENT(sub_containers, items):
  for each item in items:
    placed = false
    for each sub in sub_containers (sorted by mode):
      for each rotation in allowed_rotations(item):
        rotated_item = apply(rotation, item)
        if rotated_item fits in sub (dimensions, gap offsets, constraints):
          place rotated_item at sub.origin + gap_offsets
          narrow sub: sub.W -= rotated_item.effective_width
                      sub.X += rotated_item.effective_width
          merge and re-sort sub_containers
          placed = true
          break
      if placed: break
    if not placed: item goes to unplaced list
```



3.1.2 Comparison with Extreme-Point Placement

In extreme-point placement, each placed item generates up to three new candidate positions (at its exposed corners). The next item is scored at each candidate and the best is selected. This produces a flat set of independent candidates — any can be chosen.

In our approach, each placement modifies a single sub-container rather than generating new candidates. The narrowed sub-container is strictly contained within the original. The consequence: later items placed in the same sub-container inherit the parent region's full remaining height and length for constraint checking.

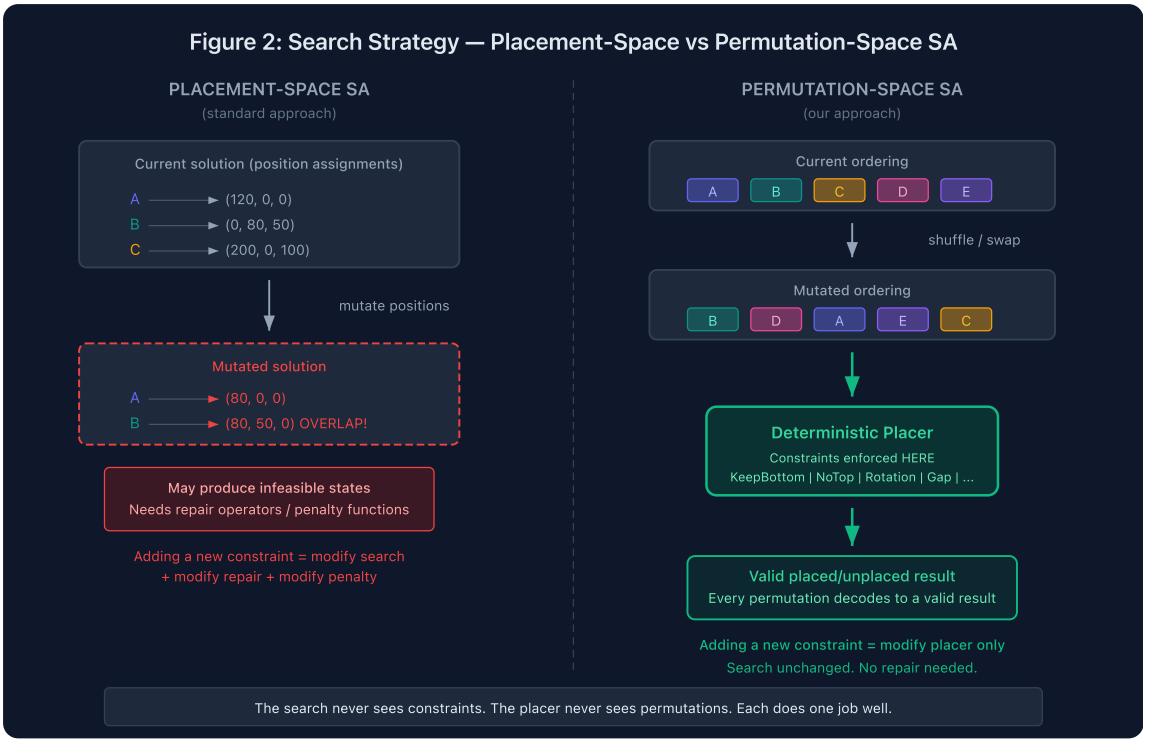
Why this helps with constraints. Consider a floor-pinned item (KeepBottom) followed by a no-stacking item (NoTop). The NoTop item needs to verify that the vertical space above it is available. In our scheme, the sub-container's inherited height reflects the true container height. In a disjoint partition scheme, the sub-container's height might have been trimmed by an earlier horizontal cut, increasing the chance that a constraint appears infeasible only because a previous partition prematurely restricted the search region. The maintained sub-container list is equivalent to a traversal over an implicit recursive partition tree, where each node inherits spatial context from its parent.

The disadvantage is candidate count: we evaluate one position per sub-container versus O(n) extreme points for n placed items. We compensate through search diversity — many orderings are evaluated, each producing different sub-container decompositions and therefore different implicit candidate positions.

3.1.3 Transposition

Before the search begins, the system evaluates items in both the original container orientation and a transposed orientation (X and Z axes swapped). This doubles the effective starting positions at the cost of two initial evaluations.

3.2 Beam-Augmented Annealed Permutation Search



We use an annealing-inspired cooling schedule over a parallel multi-neighborhood search. At each temperature, eight neighbor orderings are generated and evaluated by running the full deterministic placement routine. The best neighbor is selected and tested using an annealing-style acceptance rule. This is closer to *beam-augmented simulated annealing* than standard SA, as the beam width (eight) injects greediness at each temperature step.

3.2.1 Search State

The primary search state is an ordered list of items. The placement routine (Section 3.1) maps this ordering to a packing deterministically. The search additionally applies branch-specific orientation perturbations (Section 3.2.2), so the full state is an item ordering augmented by a small set of orientation hints.

3.2.2 The Eight-Branch Evaluation

At each temperature step, eight candidate orderings are generated in parallel:

- **Branches 1-4:** Apply orientation perturbations (rotateX, rotateY, rotateZ, identity) to the first third of the item list, respecting per-item rotation permissions — items with fixed orientation are left unchanged. These branches explore "what if the frontmost items had different orientations?"
- **Branches 5-8:** Apply random pairwise swaps to copies of the current best ordering, with swap count proportional to item count. Standard neighborhood exploration.

All eight are evaluated by running the full placement routine. The lowest-cost result is selected for the acceptance step.

3.2.3 Acceptance and Cooling

The acceptance rule follows the standard Metropolis criterion used in simulated annealing:

- If the best neighbor's cost C_new < C_current: accept unconditionally.
- Otherwise: accept with probability exp(-(C_new - C_current) / T).

The temperature T decreases geometrically: T <- T * alpha at each outer iteration, where alpha is typically 0.95. The search terminates when T drops below a minimum threshold T_min or a wall-clock cap is reached.

The wall-clock cap is mode-dependent:

Speed	Per-pass cap	Use case
VeryFast	1.5 s	Interactive preview
Fast	4.5 s	Standard web UI
Slow	9 s	Batch processing, complex loads

End-to-end runtime on large instances exceeds the per-pass cap because the system runs multiple passes: transposed evaluation, the main search, an optional back-fill pass, and the compaction post-pass.

3.2.4 Cost Function

The cost of a packing is:

```
C = p1 + p2 + p3 + p4
```

where:

p1 (unfit penalty): `10 * (sum of unplaced item volumes / container volume)`. Normalized by container volume to make the term dimensionless and unit-invariant. Dominates all other terms by design: any solution that places all items has p1 = 0 and is preferred over any solution that leaves items unplaced.

p2 (log extent sum): A logarithmic sum of each item's end-coordinate along the optimization axis, normalized by the log of the container dimension on that axis. Scale ~0.001-0.05. Retained as a weak secondary preference to distinguish equal p1/p3/p4 states; too small to drive the search meaningfully on its own.

p3 (linear extent): The maximum end-coordinate on the optimization axis, divided by the container dimension on that axis. Range [0, 5]. Directly rewards shorter packings: a 10% reduction in extent produces a 10% reduction in p3.

p4 (blank-space): `5 * (1 - compactness)`, where compactness is placed volume divided by bounding-box volume of placed items. Range [0, 5]. This is the only term that detects internal gaps. Unlike utilization (which measures how full the container is), compactness penalizes voids *within the packed region*.

Scale balance. p3 and p4 both range [0, 5], giving the search a landscape where minimizing the optimization-axis extent and maximizing internal density are co-equal objectives. In earlier versions, the extent penalty was logarithmic at scale ~0.001 — a factor of 5000x smaller than p4's operating range. At that scale, the search was blind to internal gaps when all items fit (p1 = 0). Rebalancing p3 and p4 to the same scale produced a 62% improvement in compactness on heterogeneous item sets, with no changes to the placement routine (Section 5.2).

3.3 Deterministic Compaction Post-Pass

3.3.1 Motivation

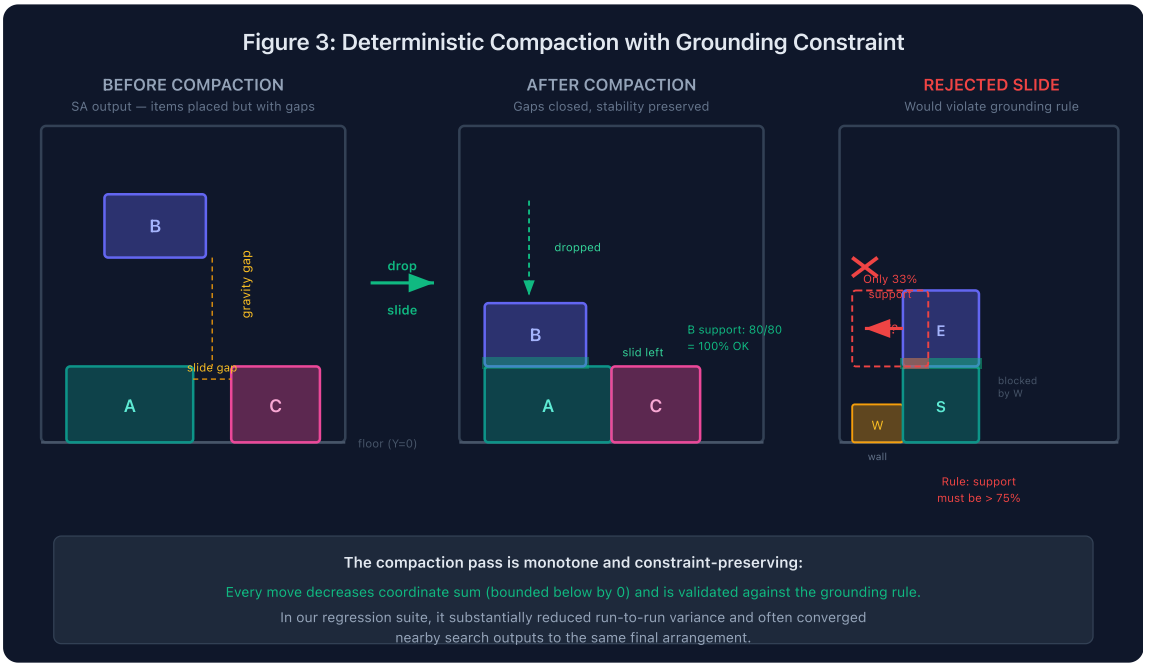
The search optimizes over orderings; the placer maps each ordering to a packing. The result is locally optimal in permutation space but not necessarily geometrically tight. Items may sit higher than necessary (because their sub-container had a high Y-origin) or further from the origin than necessary (because the sub-container's X-offset was set by an earlier narrowing).

3.3.2 Algorithm

```
COMPACT(items, container):
  repeat until no item moves:
    // DROP: process Y-ascending
    for each item (sorted by Y ascending):
      candidate_Y = highest top of items below that overlap in X-Z
      if candidate_Y < item.Y AND grounded(item at candidate_Y):
        item.Y = candidate_Y

    // SLIDE Z: process Z-ascending
    for each item (sorted by Z ascending):
      candidate_Z = rightmost Z-edge of items behind that overlap in X-Y
      if candidate_Z < item.Z AND grounded(item at (item.X, item.Y, candidate_Z)):
        item.Z = candidate_Z

    // SLIDE X: process X-ascending
    for each item (sorted by X ascending):
      candidate_X = rightmost X-edge of items left that overlap in Y-Z
      if candidate_X < item.X AND grounded(item at (candidate_X, item.Y, item.Z)):
        item.X = candidate_X
```



3.3.3 Grounding Constraint

The grounding check was not part of the original compaction design. The initial implementation moved items freely as long as no overlap occurred. In production, this caused items to slide into positions where they were geometrically legal but physically unstable.

The failure mode: a "wall" item at low X prevents a floor-level supporter from sliding, but nothing at the elevated Y-level blocks the elevated item. The elevated item slides past its supporter and ends up with insufficient contact area.

The correction applies the same grounding test during compaction that the placement routine applies during placement: the sum of overlap area between the item's bottom face and the top faces of items at exactly its Y-level must be >75% of the item's bottom-face area. Floor items (Y = 0) are exempt.

This illustrates a general lesson: any post-pass that modifies item positions must re-validate every constraint that the original placement enforced. Missing even one constraint (in this case, physical stability) introduces violations that the placement routine would have prevented.

3.3.4 Properties

The compaction pass is monotone: each accepted move strictly decreases the item's coordinate, and coordinates are bounded below by zero. Convergence occurs within 2-4 sweeps in practice.

In our 15-scenario regression suite, the compaction pass substantially reduced run-to-run variance. On the S4 scenario (heterogeneous mid-size items), pre-compaction runs produced compactness values ranging from 0.41 to 0.57 across runs due to the search's time-bounded stochastic nature. After compaction, all tested reruns of S4 converged to compactness 0.724 with identical final coordinates.

4. Constraint Interaction Analysis

Individual constraints are straightforward. The difficulty is in their composition. We describe interactions discovered through production deployment.

4.1 Floor-Pinned + No-Stacking + Floor Saturation

A floor-pinned item (KeepBottom) occupies floor area at Y=0. A no-stacking item (NoTop) cannot support weight above it, effectively requiring its own vertical column. When both appear in the same item set, they compete for floor area. In a container with limited floor area relative to the combined footprints, the placer must interleave these items carefully.

Within feasibility-preserving ordering logic, floor-pinned items are favored as a heuristic because they irreversibly consume floor area — placing them late risks finding no valid floor position.

4.2 Order-Dependent Gap Collapsing

When two items with gap requirements g1 and g2 are placed adjacent, the effective gap is max(g1, g2), not g1 + g2. This prevents gap inflation in dense packings.

However, the collapsed gap depends on which neighbor is already placed when the new item arrives. The placement routine computes the nearest neighbor among items already in the container and applies the collapsing formula. Different item orderings produce different total gap allocations and different packing densities.

Our permutation-based search reaches these different gap allocations naturally because placement order directly affects the decoded geometry. This is an example of how permutation-space search captures degrees of freedom that would require explicit gap-assignment variables in a placement-space formulation.

4.3 Stack Limits + Rotation + Container Height

A stack capacity of k means at most k items above. The number that fits depends on their heights, which depend on their rotations. An item of 100x200x50 contributes 200 units of height in one orientation and 50 in another. The rotation choice is therefore coupled to the stacking feasibility check: the placement routine evaluates rotations within the context of remaining height and stack capacity at the candidate position.

4.4 Grounding + Compaction

The grounding constraint interacts with the compaction post-pass in a way not anticipated during design (Section 3.3.3). This is an interaction between an invariant (physical stability) and a system-level optimization (geometric compaction) — a category that constraint-handling literature typically does not address because most formulations do not include post-processing passes.

5. Experimental Results

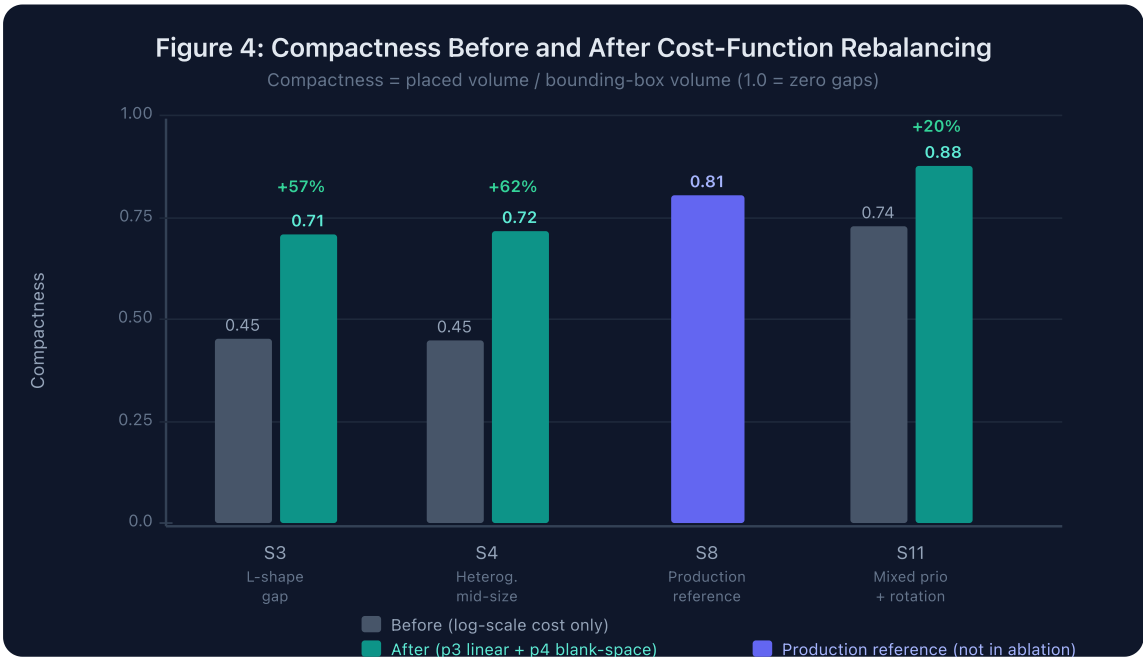
5.1 Methodology

We evaluate using a suite of 15 scenarios checked automatically after every algorithm change. Each scenario defines a container, an item list, and hard invariants:

- No overlap between any pair of placed items
- Every item fully inside the container
- Item dimensions are a permutation of original dimensions (rotation only)
- Conservation of item IDs across placed + unplaced sets
- Floor-pinned items at Y = 0
- Every elevated item has >75% bottom-face support

Additionally, each scenario has golden baselines: compactness floors, extent ceilings, and minimum placement counts. An algorithm change that drops below a baseline on any scenario is rejected.

Scope. The evaluation is intended primarily for regression safety and comparative change detection within the production system, rather than as a comprehensive benchmark study against public baselines. Reported values are from single runs at fixed seed (seed = 42) per scenario, with stability verified across 3-5 repeated runs to confirm low variance. We do not currently compare against an external extreme-point or greedy baseline implementation; the ablation in Section 5.2 isolates the contribution of each system component against the system's own prior configuration.



5.2 Cost-Function Rebalancing (Ablation)

The most significant recent improvement came from rebalancing the cost function, not from changing the placement routine. We isolate the contribution of each change:

Configuration	S3 compact	S4 compact	S11 compact
Base placer + log-scale cost (p1, p2, old p3)	0.454	0.448	0.736
+ linear extent p3 (scale 0-5)	0.454	0.484	0.736
+ blank-space p4 (scale 0-5)	0.714	0.570	0.736
+ p3 linear AND p4 together	0.688	0.644	0.736
+ compaction post-pass	0.688	0.724	0.736
+ safety-net classic-sort comparison	0.688	0.724	0.883

The *safety-net classic-sort comparison* evaluates a deterministic baseline ordering (items sorted by priority, then by volume descending) alongside the annealed search output and keeps the better result when the search underperforms on any of: placed count, optimization-axis extent, or compactness.

S3 (16 mixed-size items, gap-prone L-shape): compactness improved 0.454 to 0.688 (+51%). **S4** (20 heterogeneous mid-size items): compactness improved 0.448 to 0.724 (+62%). Maximum Z-extent dropped from 5300mm to 3435mm (35% shorter along the optimization axis). **S11** (24 mixed-priority items with rotation): compactness improved 0.736 to 0.883 (+20%). The safety-net detected that the annealed search result was worse than the deterministic classic-sort baseline on this scenario and swapped in the better result.

5.3 Production Deployment

The system handles a production load of 730 items across two box types (300 units of 850x440x310mm, 430 units of 855x390x195mm) in a 40-foot container (12035x2352x2695mm):

- **82% utilization** (62.7 m^3 placed / 76.3 m^3 container), close to the volume upper bound implied by total item volume (82.2%)
- **All 730 items placed** (zero unplaced)
- **All constraints satisfied** (orientation, support, priority)
- **End-to-end runtime:** under 60 seconds on commodity hardware at Slow speed, including transposed evaluation, search, back-fill, compaction, and validation passes

5.4 Compaction Effect on Variance

Before compaction, the S4 scenario produced compactness values ranging from 0.41 to 0.57 across 5 runs (a 39% spread). After compaction, all 5 runs converged to 0.724. In our regression suite, the compaction pass often collapsed nearby search outputs to the same final arrangement, making results reproducible regardless of how many search iterations completed within the time budget.

The compaction pass's primary value is variance elimination and correctness (preventing grounding violations), not raw compactness improvement. On scenarios where the cost-function rebalancing already pushes the search toward compact solutions, the post-pass improves compactness by only 1-3% absolute.

6. Limitations

Single candidate per sub-container. This is the main architectural tradeoff in the current system. The placement routine evaluates one position per sub-container (the origin). Extreme-point methods evaluate O(n) candidates per item. There exist item configurations where no permutation of our scheme produces the optimal placement that an extreme-point method would find directly.

S5 regression. The cost-function rebalancing that improved S3/S4/S11 caused a regression on one scenario (KeepBottom + NoTop mix): compactness dropped from 0.71 to 0.57, and the MinimizeHeight extent increased from 160 to 200 (25% worse). Investigation confirmed the regression is structural: the single-candidate-per-sub-container placement cannot reach the 160-extent solution under the rebalanced priority transformation. This is a limitation that richer candidate enumeration would address.

O(n^2) per compaction sweep. Each sweep checks each item against all others. For 730 items, this is approximately 2 million comparisons per sweep (2-4 sweeps typical). This is fast on modern hardware but scales quadratically. A spatial index would reduce this to O(n log n).

Constraint completeness. The constraint set was developed in response to customer requirements, not systematically enumerated. Constraints not currently modeled include: graduated fragility levels (vs. binary NoTop), orientation-dependent effective dimensions, load-bearing wall specifications, and multi-container optimization with cross-container constraints.

7. Conclusion

We have described a production container loading system combining recursive space-partition placement, beam-augmented annealed permutation search, and grounding-aware compaction. The system handles orientation restrictions, stacking limits, gap collapsing, per-item rotation control, physical stability, and priority ordering — constraints that interact in ways we did not find analyzed in prior work.

Two insights from several years of production deployment:

First, packing quality under constraints is a two-dimensional optimization: the system must minimize extent along the optimization axis *and* maximize density within the packed region. Cost functions that measure only extent leave the search blind to internal gaps. Adding a blank-space penalty on the same scale as the extent penalty produced the largest single improvement in our system's history (62% compactness gain on heterogeneous inputs) with no changes to the placement routine.

Second, any post-pass that modifies item positions must re-validate every constraint the original placement enforced. The grounding violation we discovered in production was caused by exactly this oversight: the compaction pass moved items freely, and one geometrically valid move was physically invalid (insufficient bottom-face support). The general lesson is that post-processing and constraint enforcement are not independent concerns.

Availability

The system is available at 3DPack.ing for interactive container loading optimization and at GPTPacker.com for AI-assisted freight loading. Technical inquiries: contact via 3DPack.ing.

References

1. Bierwirth, C. (1995). A generalized permutation approach to job shop scheduling with genetic algorithms. *OR Spectrum*, 17(2-3), 87-92.
2. Crainic, T. G., Perboli, G., & Tadei, R. (2008). Extreme point-based heuristics for three-dimensional bin packing. *INFORMS Journal on Computing*, 20(3), 368-384.

3. Imahori, S., & Yagiura, M. (2010). The best-fit heuristic for the rectangular strip packing problem. *European Journal of Operational Research*, 206(2), 248-252.
4. Lodi, A., Martello, S., & Vigo, D. (2002). Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123(1-3), 379-396.
5. Martello, S., Pisinger, D., & Vigo, D. (2000). The three-dimensional bin packing problem. *Operations Research*, 48(2), 256-267.
6. Ow, P. S., & Morton, T. E. (1988). Filtered beam search in scheduling. *International Journal of Production Research*, 26(1), 35-62.