AlgoNest Solver — how it nests, what it yields, and how we know

Technical whitepaper, version 1.0, 13 September 2026. AluPro sp. z o.o.

Summary

AlgoNest Solver is our own nesting engine for plasma cutting: it lays parts on a strip of sheet of a given height and minimises the length used. On a set of 48 shop-floor jobs (four profiles, three spacings, 30–150 parts) it reaches an average material utilisation of **65.25 %**, measured on the true outlines with their holes, against **67.36 %** for the strongest academic solver we know (sparrow, KU Leuven) in the same order of time (~100 s), and **7–13 points above Deepnest** on the shared subset. On series of identical parts and on parts nested inside other parts it beats both: squares **92.4 % vs 85.5 %**, part-in-part **70.0 % vs 53.5 %**. The solver also plans lead-ins: it chooses the entry point and the lead length so the lead fits in the scrap, never enters a neighbour or the parent part, and — on request — starts inside the kerf of a part already cut.

1. What we measure, and why we do it ourselves

Every solver reports utilisation its own way: one counts part area *with the spacing pressed into it*, another *without holes*, a third against a rectangle it chose for itself. The differences reach 7–15 points and say nothing about the sheet. So every number in this document comes from **one judge**: it takes a finished layout (ours, sparrow's from its SVG, Deepnest's from its DXF), drops the *true* outline with holes into every position, measures the width as the rightmost point and computes

utilisation = Σ part area / (strip height × width used).

The judge also checks that pairs of parts keep the required spacing, that nothing sticks out of the strip and that quantities match. A layout that fails the audit does not enter the table — and a layout from our solver never leaves it without passing.

2. The test set

48 instances generated from 41 shape families: 11 shop shapes with holes (flanges, brackets, slotted plates, channels, discs, frames, rings), 8 hard ones (gears, blades, hooks, stars, crescents, combs, zigzags, darts) and 22 **real AluPro outlines** from production DXF files. Four profiles: `repeat` (series with holes), `hard`, `balanced`, `alupro` (a real mix). Spacing 0 / 3 / 6 mm, strips 1000–1500 mm. Split 32 tuning / 16 control instances; both halves behave the same (−2.39 vs −1.58), so nothing is overfitted.

3. How the solver nests

The engine has two storeys and each is measured on its own.

Construction (10–40 s). First the job is analysed: which parts have holes able to take other parts (*part-in-part* — they are assembled into rigid bodies, quantities preserved), which classes come in series (for those we compute the densest **lattice** — a periodic pattern in which the part repeats every constant vector; on squares and circles this reaches densities a general search never finds), which pairs of parts sit well together (*clusters*). From this alphabet a bottom-left construction builds the layout on a conservative raster (the grid is coarsened so that "no overlap on the raster" means "no overlap in geometry"), tries several orders and picks the variant (lattice, cluster, plain) by measurement, not by a rule. Position search is a scan on a bit grid in C with pruning (for every x only the y below the best so far is tested); the same scan runs in parallel on the graphics card as an **OpenCL 1.2** kernel (no vendor lock-in): construction of a 100-part job takes 20–50 s on the CPU, 9–22 s on the card, identical positions.

Compaction (the rest of the budget). Construction never revisits a decision, so its layout is a fixed point for every move that forbids overlap — we measured this literally: zero moves. Compaction works differently: it narrows the strip by a fraction of a percent, lets parts overlap *for a while* and pushes them apart with a search in which stubbornly colliding pairs accumulate penalties and separated pairs shed them. Three threads try different move orders and the best one survives; after a run of failures the layout is disrupted (two large bodies swap) and tried again. The collision engine underneath is our own bit grid (64-bit words, AND and popcount), written in C — 4–10 thousand part moves per second on typical jobs. Parts nested in holes travel with their host as one rigid body, so part-in-part survives compaction intact. At the end: an audit on exact geometry.

Time. The solver derives the budget from the job size and stops early when the strip stops narrowing; the user only picks *fast / normal / thorough* (calibration: jobs of 30–35 parts saturate after ~60 s, 80–100 parts keep gaining up to ~240 s). The code also carries a small neural network trained on our own runs that predicts whether the full search over construction variants will pay off; measured on the 48 jobs it changed neither the result (−0.05 pp) nor the time, so it is switched off in the product — we say so because numbers matter more than labels.

4. Results

4.1. 48 instances, one judge

profile	n	construction	**full engine**	sparrow	gap	within −3 pp	time
repeat	12	62.62	66.70	68.75	−2.05	8/12	92 s
balanced	12	57.52	61.67	63.96	−2.29	8/12	95 s
alupro	12	68.12	69.92	71.87	−1.95	9/12	118 s
hard	12	56.43	62.69	64.87	−2.18	9/12	102 s
all	**48**	**61.17**	**65.25**	**67.36**	**−2.12**	**34/48**	**101 s**

Sparrow had two runs of 60 s on three threads (about 120–130 s). Job size does not change the gap (≥ 100 parts: −1.67; < 50: −2.17). Wins: `shop_repeat_005` +6.8, `shop_alupro_006` +0.7, `shop_alupro_007` +0.6, `shop_balanced_005` +0.6, `shop_hard_005` +0.5 — all with series of identical parts.

4.2. Where structure is visible

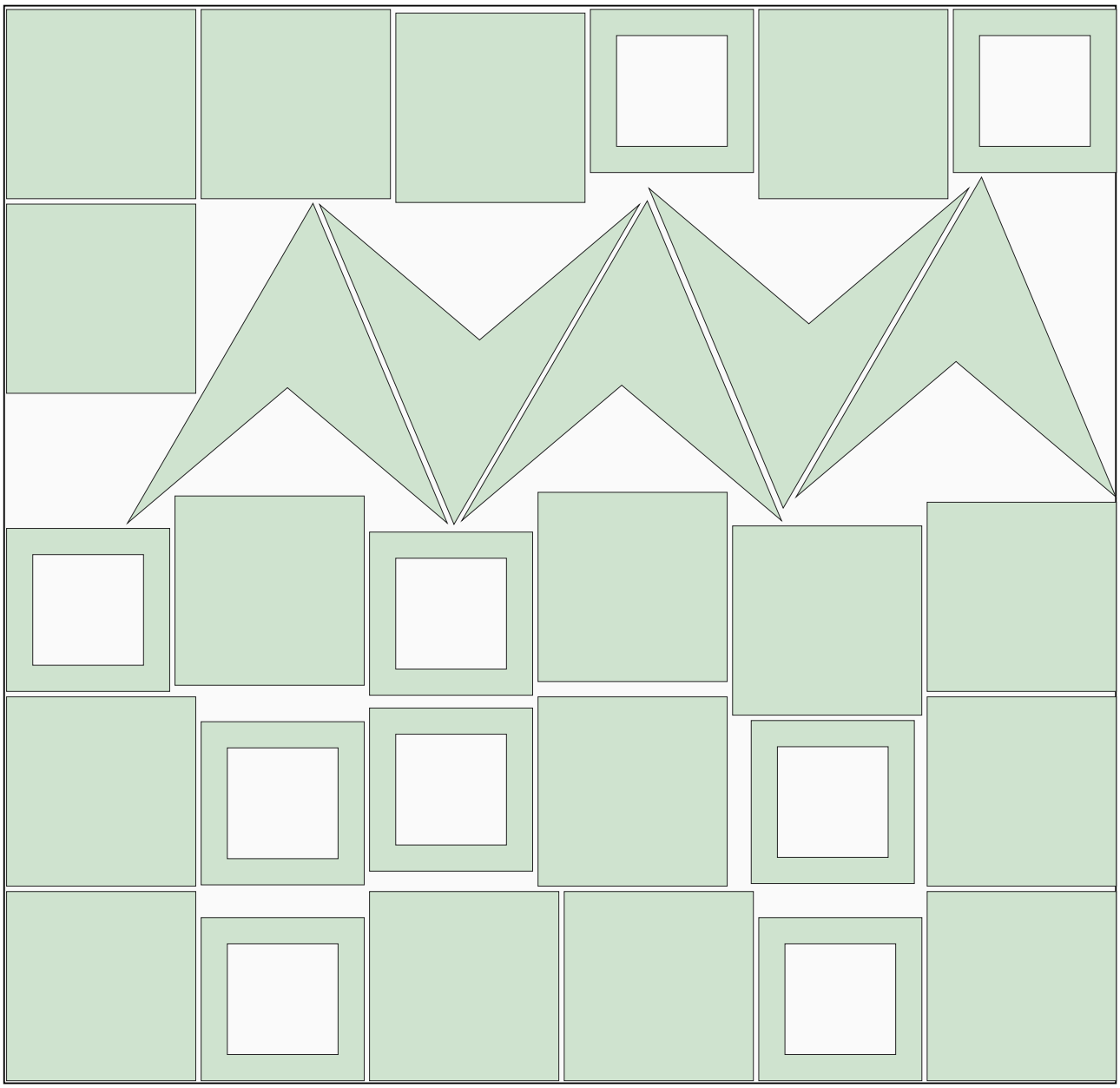
set	AlgoNest	sparrow	note
squares (one class)	**92.41**	85.49	lattice; sparrow tries 12 angles and misses the period
part-in-part	**69.98**	53.46	sparrow does not see holes in parts
circles	65.91	81.15	we lose: hexagonal lattice vs free packing

4.3. Deepnest, four instances, same judge

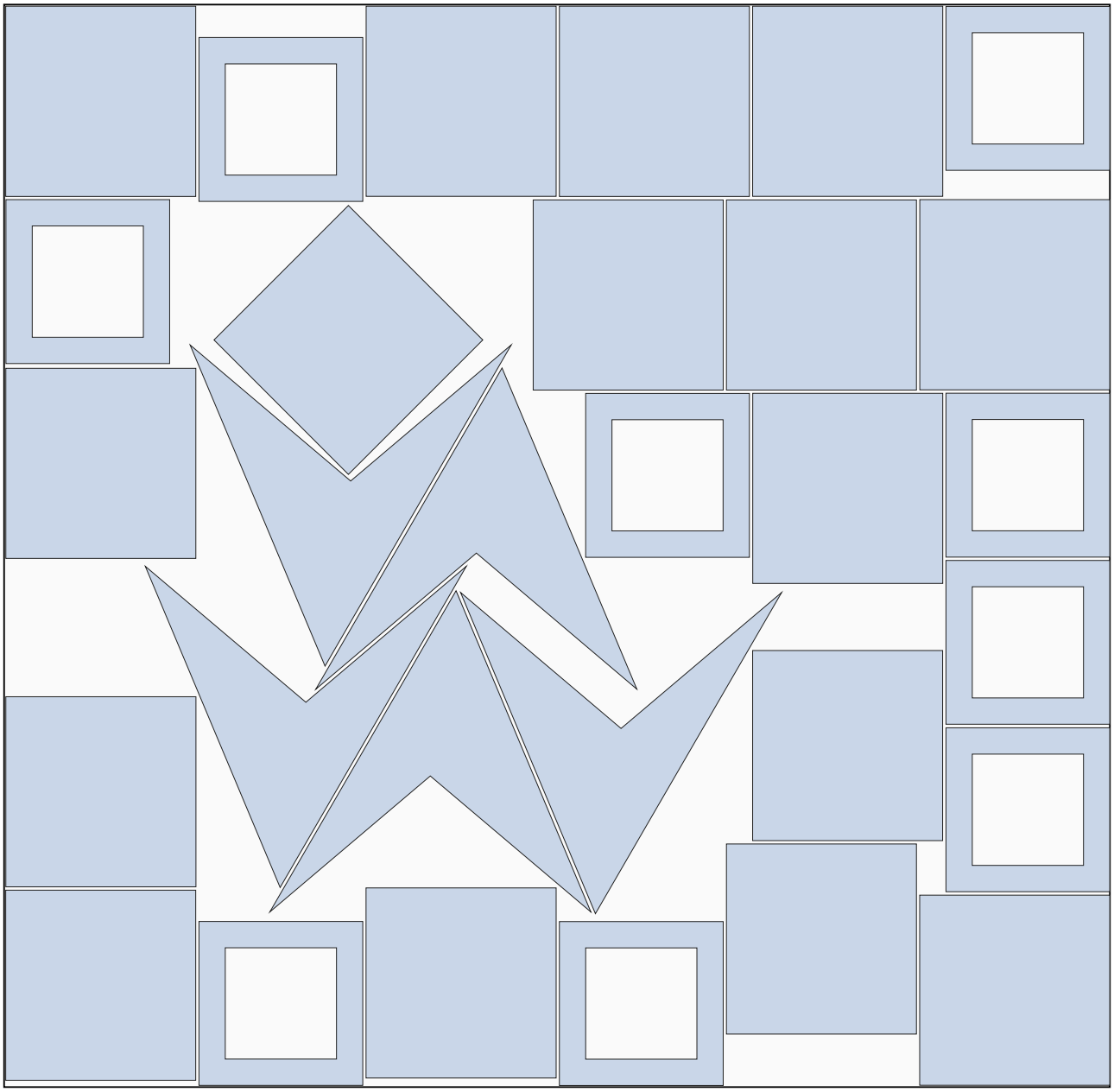
instance	Deepnest (180–240 s)	AlgoNest (120 s)	sparrow
hard_010	58.65	70.86	71.59
balanced_007	52.44	67.77	70.97
repeat_007	63.23	76.49	77.02
alupro_001	48.60	51.73	52.90

Deepnest violates the requested spacing in every instance (6–15 pairs too close), so its numbers are generous to it.

AlgoNest: 70.86% (1032 mm)



sparrow: 71.59% (1022 mm)



Deepnest: 58.65% (1247 mm)

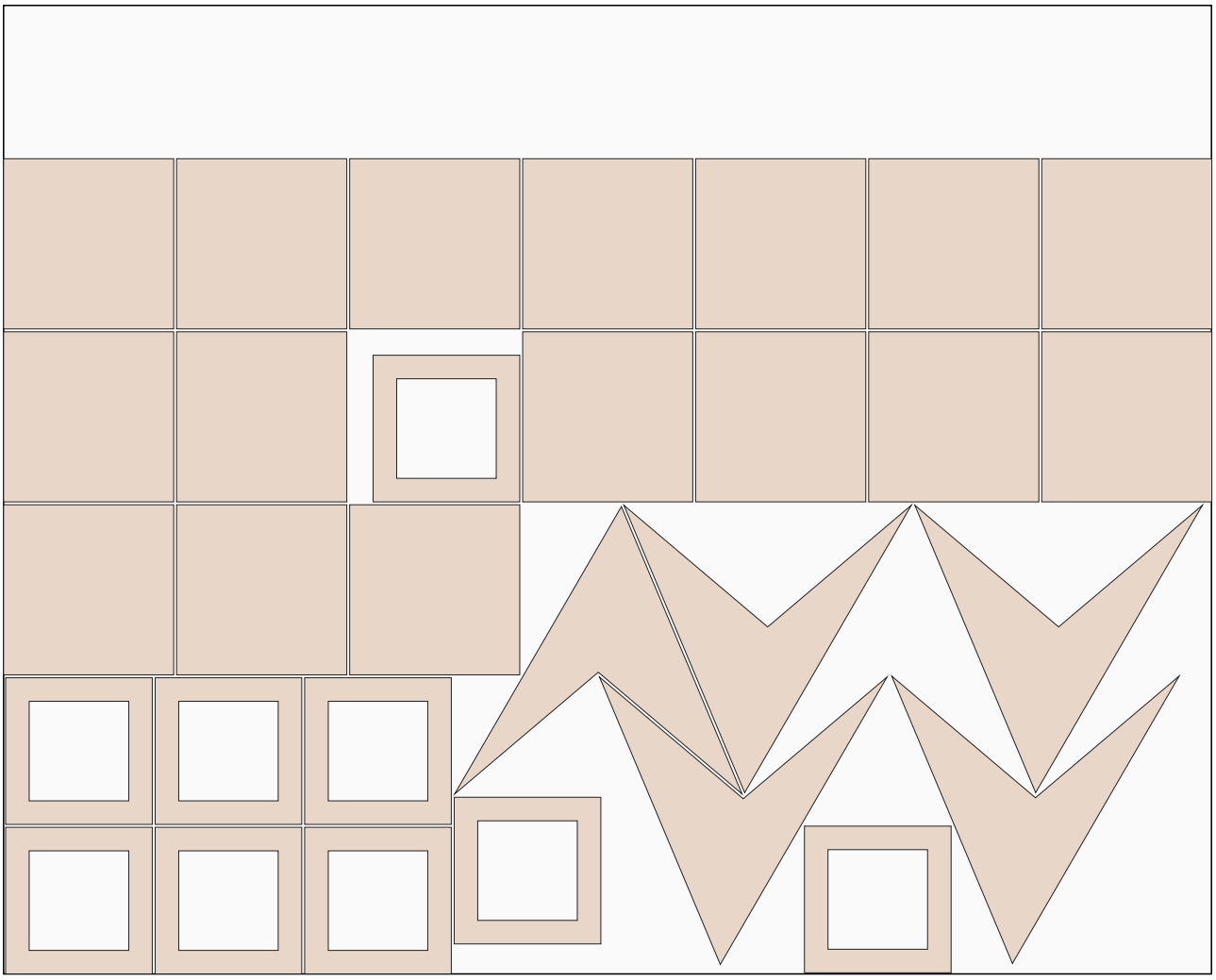


Fig. 1. The same instance (`shop_hard_010`*, 30 parts, 3 mm spacing) nested by the three engines; drawn from the true outlines, as the judge counts them.*

5. Lead-ins

For every contour (outer, each hole) the solver reports **where to enter and how long a lead fits there**: it checks the whole lead-in/lead-out geometry against neighbours grown by the minimum pierce clearance, against the plate edge and — for holes — against the hole interior (a hole lead never enters the parent part). It shortens the lead when it must; when not even 1 mm fits, it reports a pierce on the contour and leaves the decision to the operator. Leads of different contours never cross. **"On the edge" mode**: in the cut order chosen in the program, every next part starts inside the kerf of a neighbour already cut (or just outside the plate edge) if the flight across the scrap is shorter than the set limit and crosses nothing — saving a pierce and nozzle life on thick plate. The lead geometry is the same the program draws and the operator can edit by hand.

6. What the solver does not do, and what we know about its limits

- On mixes of irregular shapes it stays 2–3 points behind sparrow. We measured

every idea meant to close that — beam search, decision trees, three neural networks, exhaustive collision maps on the GPU — and none gave more than 2–4 points where 8 were needed. Only compaction with overlap closed it.

- On large outlines (over a metre) with wide spacing the engine makes fewer

moves per second and 120 s can be too little; that is the nearest thing to improve.

- Circles: a hexagonal lattice loses to free packing.

7. Technology

Python (numpy, shapely) in the control layer; the collision engine and the compaction in C99 with POSIX threads, no dependencies; construction on the CPU (a C scan), OpenCL 1.2 (pyopencl) optional, with an identical result; a small neural network (pure numpy) as the "how much to compute" gate — present, switched off. The judge and the 48 instances live in the repository and every number in this document can be reproduced with one command.