ALGONEST · RESEARCH REPORT

Attacking sparrow

How we tried to beat the best open nesting algorithm, and what we learned by failing

AlgoNest · 12 September 2026

Abstract

We set out to beat `sparrow` — the strongest publicly available algorithm for irregular-shape nesting, published in September 2025 by a team at KU Leuven. We built a nesting engine from scratch around rasterisation and GPU acceleration, stated six falsifiable hypotheses, and tested every one of them by measurement on thirteen classic benchmark instances.

We failed. Our engine lost to `sparrow` on thirteen instances out of thirteen, by an average of 8.78 percentage points. It did, however, beat our own previous generation by 7.13 points on twelve instances out of thirteen.

When the frontal attack failed, we tried to improve `sparrow` **from the outside** with three ideas of our own — gravity settling, vibration compaction, and clustering parts before nesting. All three measured at zero or below. We could not improve it externally either.

Two of the six hypotheses were confirmed, three refuted, one we failed to implement effectively. Along the way we caught **two measurement errors of our own** that briefly showed a success that was not there. We describe them, because they say more about this field than any of our positive results.

The most valuable finding turned out to be that the same structural technique (clustering) **improves a weak solver by 2.12% and degrades a strong one by 0.36 points** — which says something general about when propping up an algorithm with a heuristic helps, and when it merely gets in the way.

---

1. Why attack someone else's algorithm

In sheet-metal fabrication, nesting is money. One percentage point of sheet utilisation across a thousand plates a year is a real sum, and the spread between tools runs into double digits. Anyone building software in this trade needs to know where they actually stand — not where they imagine they stand.

We had been benchmarking our previous engine against **Deepnest**, a popular open-source tool from 2016. We were barely keeping up with it, and we treated that as a result. That was a methodological mistake: **a bar set too low produces a false sense of progress.** Successive iterations stopped adding anything, because we were optimising against a reference point that was not itself ambitious.

So we started over, with an explicit goal: **measure ourselves against the best that exists in public**, and find out how far short we fall.

---

2. The opponent

sparrow 0.2.0 — a heuristic for two-dimensional irregular strip packing, published as *"An open-source heuristic to reboot 2D nesting research"* (arXiv:2509.13329, September 2025) by Jeroen Gardeyn of KU Leuven. It is built on the **jagua-rs** collision engine, described separately in *INFORMS Journal on Computing*. MIT licence. Roughly 4,600 lines of Rust.

We chose it because it satisfies three conditions at once: it is **the strongest published** result, it is **fully available** (so the comparison is verifiable rather than resting on a vendor's claims), and it is **current** — it represents today's state of knowledge, not that of a decade ago.

The difference in algorithmic family is worth stating up front, because it turns out to be the heart of the matter:

	constructive	**separation and compaction**
examples	Deepnest, our raster engine, most commercial systems	`sparrow`, ILSQN, Umetani's work
principle	place part after part, **never allow overlap**	place everything, **allow overlap**, then pay it down
move	choosing order and position	pushing parts around in a tight layout

3. The test material

Two data sets. The first is a real production job from our own shop — 120 parts of seven distinct shapes, cut from 10 and 15 mm plate, on a 1500×3000 mm sheet with 6 mm spacing between parts.

Figure 1. The production part set. All seven shapes drawn at a common scale, so the size relationships are true. Quantities are for the full job of 120 parts.

The second set is thirteen classic ESICUP benchmark instances — `albano`, `blaz1`, `dagli`, `fu`, `jakobs1`, `jakobs2`, `mao`, `marques`, `shapes0`, `shapes1`, `shirts`, `swim`, `trousers`. Between 4 and 25 distinct shapes, between 12 and 99 pieces. These have been used in the literature for decades and,

importantly, **we did not choose them ourselves** — a self-selected test set can always be unconsciously fitted to one's own algorithm.

---

4. Our hypothesis

The starting observation was that the dominant geometric approach in nesting — the **no-fit polygon**, essentially the Minkowski sum of two polygons — is expensive to prepare. For n shapes and r orientations you must compute $(n \cdot r)^2$ pairs. For a hundred distinct parts at four angles that is 160,000 geometric operations, each requiring convex decomposition and degenerate-case handling.

Our hypothesis: **if you rasterise the shapes, the same collision test reduces to a bitwise AND on machine words**, preparation cost falls from quadratic to linear, and the test becomes simple enough to parallelise on a graphics card and search far more layouts.

One point deserves honest statement up front: **rasterisation is not an escape from the Minkowski sum — it is its numerical equivalent.** The map of admissible positions for a part *is* the discrete Minkowski difference. The gain is therefore in cost, not in geometry.

Correctness: conservative dilation

For the raster not to lie, every part is inflated before rasterisation by `spacing/2 + s·√2/2`, where `s` is the pixel size. The proof is one line: any point of the part lies no further than `s·√2/2` from the nearest pixel centre, so that pixel is certainly set. If two parts genuinely overlap they share a point, and therefore share a set pixel — **the collision is always detected**.

The implication runs one way only: the algorithm may reject a good layout, but it **can never accept a bad one**. That is the right direction of error for production. Verified empirically: zero overlaps, minimum real clearance 6.83 mm against 6.0 mm required.

---

5. Hypotheses

We framed six statements so that each could be **refuted by measurement**.

#	hypothesis	outcome
H1	Raster gives linear preparation cost instead of quadratic NFP	**confirmed**
H2	More layouts evaluated on the GPU translates into a better result	**refuted**
H3	A better metaheuristic (ruin & recreate) beats a genetic algorithm	**refuted**
H4	Changing the decoder's **input** beats changing the search	**confirmed, conditionally** — only for a weak solver
H5	Settling with vibration extracts more from a finished layout	**refuted**
H6	Overlap minimisation on the raster escapes the constructive ceiling	**not achieved**

6. Method

Honest measurement is harder in this field than the algorithm itself, so we describe it separately.

Hardware. One machine: NVIDIA RTX 2060 (TU106, 6 GB GDDR6, 1920 CUDA cores, 336 GB/s), six physical CPU cores (12 logical), Windows 11.

Budget. 60 seconds per instance for each tool. `sparrow` was given six parallel runs across six physical cores — the whole CPU; our engine was given the whole GPU. Run **sequentially**, so they never competed for the machine.

Metric. One metric for everyone: part area divided by the area of the bounding rectangle of the layout. `sparrow`'s results were read from its own output files and **recomputed by our code**, so that the same function produced both columns.

Geometric verification. Every layout checked on real polygons (Shapely), not on the raster: number of overlapping pairs, minimum real clearance, containment within the sheet. **This step turned out to be the most important part of the method** — see section 8.

Resolution. Measured, not guessed. On `jakobs1` a coarser raster beat a finer one (600 px: 79.88%, 4000 px: 75.20%), because throughput outweighs precision. The optimum sat around 600 pixels across the strip height, and we used that value throughout.

7. Results

7.1 Head to head

Thirteen instances, 60 s each, same machine. The "previous engine" column is our own earlier NFP-based generation, measured on the same set.

instance	shapes	pieces	previous engine	AlgoNest raster	`sparrow`
albano	8	24	81.83%	83.02%	**88.91%**
blaz1	7	28	56.47%	73.33%	**84.12%**
dagli	10	30	73.55%	81.84%	**88.12%**
fu	12	12	76.57%	84.73%	**91.92%**
jakobs1	25	25	65.77%	79.88%	**89.08%**
jakobs2	25	25	62.44%	71.29%	**83.90%**
mao	9	20	81.12%	76.90%	**85.99%**
marques	8	24	82.11%	82.80%	**90.91%**
shapes0	4	43	48.75%	59.78%	**67.61%**
shapes1	4	43	52.63%	66.12%	**75.19%**
shirts	8	99	67.31%	80.20%	**88.72%**
swim	10	48	64.88%	65.64%	**76.38%**
trousers	17	64	81.88%	82.46%	**91.32%**
average			**68.87%**	**76.00%**	**84.78%**

Our raster engine beat the previous generation on 12 instances out of 13 (+7.13 pp average). **`sparrow` beat our raster engine on 13 out of 13** (+8.78 pp average). Without a single exception.

The gap is visible to the naked eye. Both layouts below solve the same instance under the same time budget:

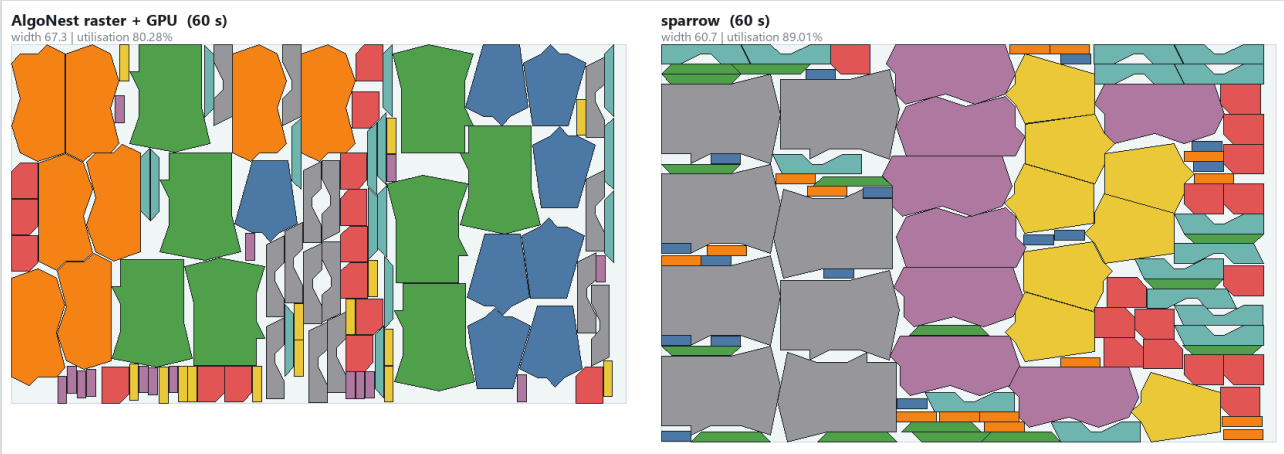


Figure 2. shirts — 99 pieces, 60 s, same machine. Left: our raster engine, 80.28%. Right: sparrow, 89.01%. Note how sparrow interlocks neighbouring shapes and leaves far less white space; our constructive decoder places each part once and never revisits it.

7.2 Throughput

On the production job (120 parts, 1500×3000 mm sheet):

	layouts per second
CPU, FFT correlation	0.04
GPU, CUDA	108
GPU, OpenCL	305

Speed-up over the CPU: **2,700×** for CUDA and **7,600×** for OpenCL. Preparing 28 oriented raster stencils took **97 ms**; the corresponding set of NFP pairs is 406 geometric operations, each requiring convex decomposition. **H1 confirmed.**

A surprising side result: NVIDIA's OpenCL compiler produced code **2.9× faster** than NVRTC for the same algorithm, with bit-identical output. We did not establish why, and we are not going to guess.

7.3 What compute does not buy

The single most important measurement of the whole exercise:

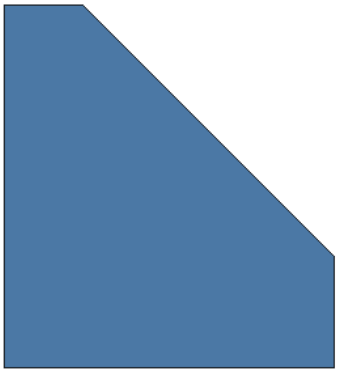
run	layouts evaluated	result
CUDA, 600 s	64,768	1944 px
OpenCL, 600 s	**184,320**	**1944 px**

2.85 times more search produced a result identical to the pixel. The record was set at generation 260, after 66,816 evaluations; the following **117,504 layouts improved nothing**.

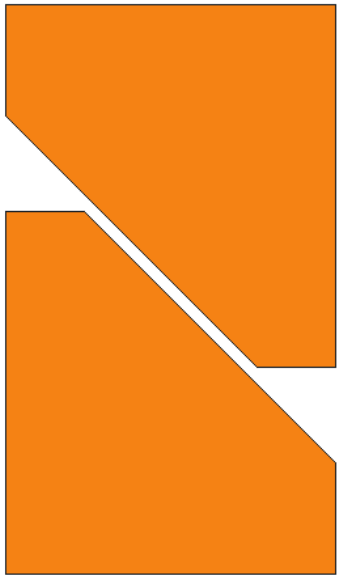
Independently: ruin & recreate tied with the genetic algorithm — 1959.3 against 1959.0, at a standard deviation of 10–13. **H2 and H3 refuted.**

7.4 What changing the input does buy

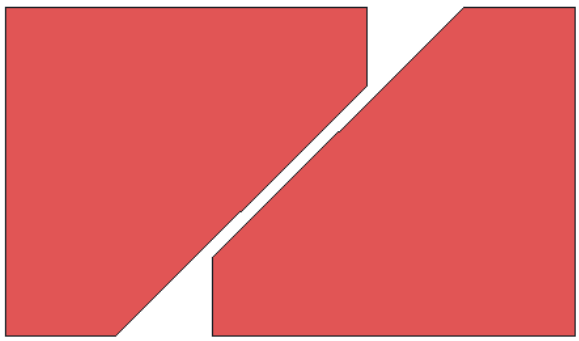
Pairwise clustering — exhaustively searching every relative placement of two shapes and fusing the best pairs into a single object before nesting:



single: 74.6% of its box



paired: 96.1% (+28.7%)



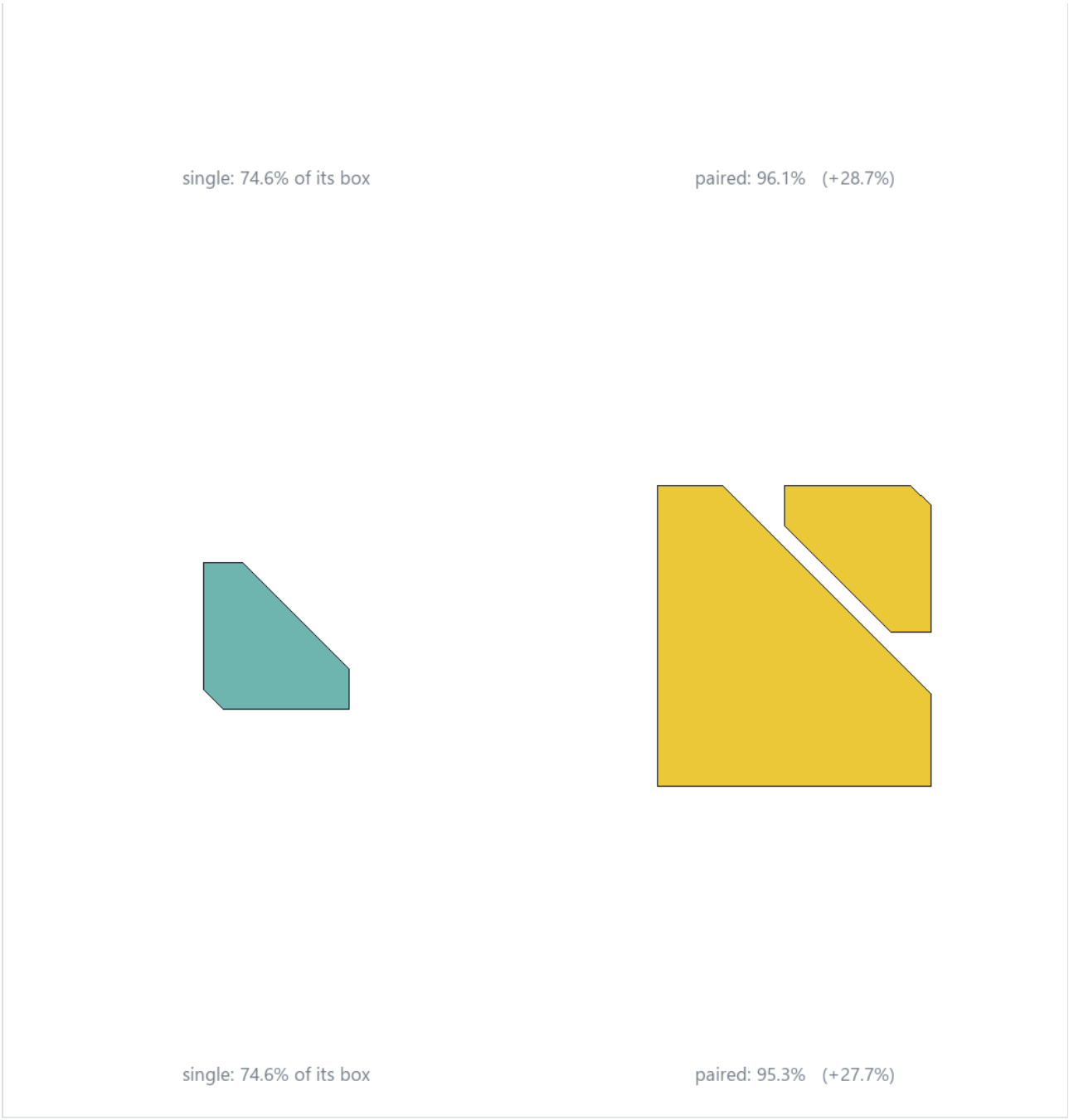


Figure 3. Clustering finds which shapes have slack. Left column: the part alone, filling its own bounding box. Right column: the best pair found. Triangular ribs gain 25–29%; a part that already fills 97% of its box is correctly left alone.

variant	mean of three seeds	utilisation
without clustering	1964.7 px	78.54%
with clustering	**1923.0 px**	**80.24%**

+2.12%, and every seed improved; the worst clustered result beat the best unclustered one. The metric correctly picks what is worth pairing: triangular ribs (74.6% dense on their own) combine into pairs at 96.1% density, while the trapezium, which already fills 97.2% of its own rectangle, is skipped — there is nothing there to improve. **H4 confirmed** (with the qualification in section 8.4).

This was the only thing in the entire exercise that moved the number upward.

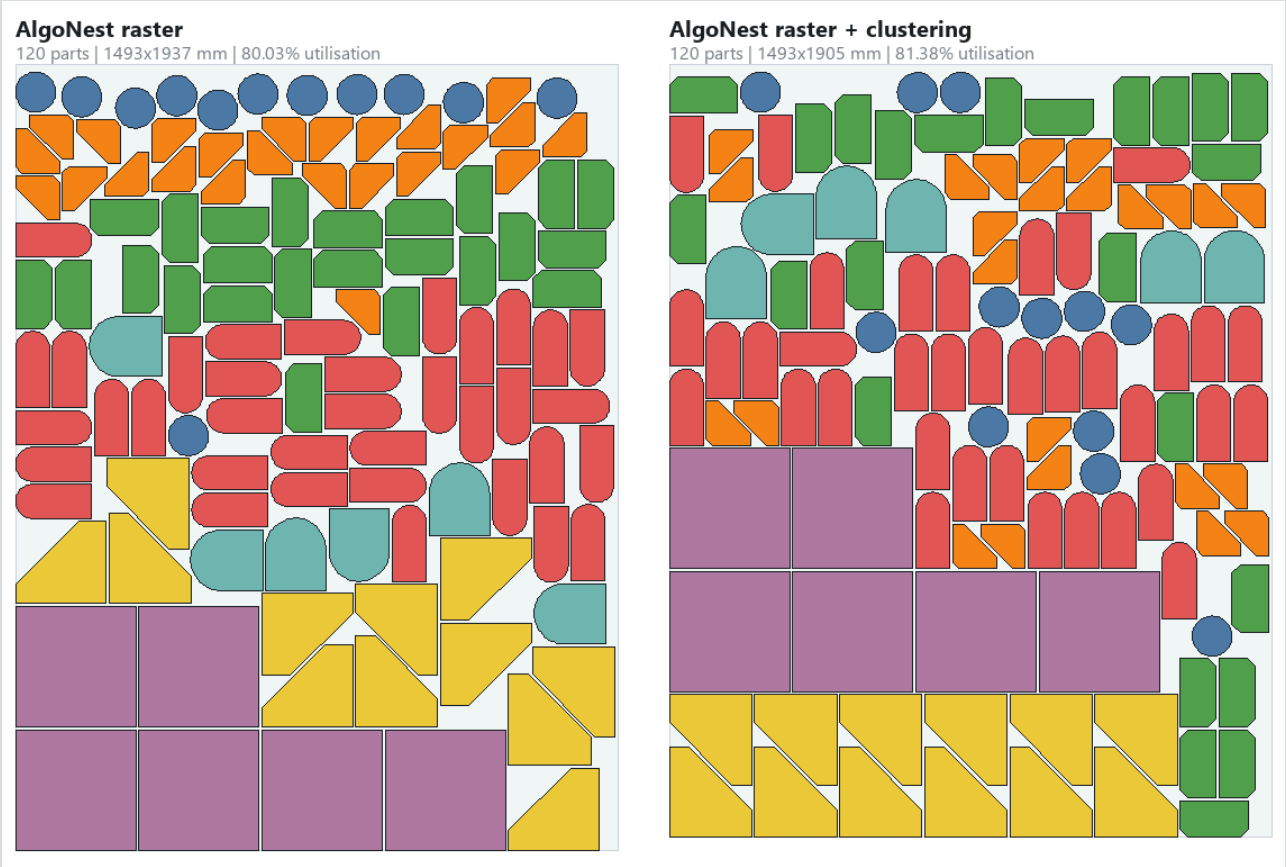


Figure 4. The production job, 120 parts on a 1500×3000 mm sheet. Left: raster engine alone, 80.03%. Right: the same engine with clustering, 81.38% — note the triangular ribs fused into near-perfect rectangles.

---

8. Three attempts to wrap `sparrow` — and two errors of our own

With the frontal attack lost, we tried to improve `sparrow` **from the outside**, without touching its code: two treatments on the finished layout, one on the input.

The first part of this section is in the document deliberately, though it does us no credit. It describes two moments when our code reported a success that did not exist.

8.1 Vibration that drove parts through each other

The idea — borrowed from the physics of granular materials — was: at the end, rotate the small parts by two or three degrees at random and pull everything down under gravity, as when you tap a jar to settle its contents. Repeat thousands of times.

The first measurement on a finished `sparrow` layout showed **+9.38 percentage points** on instance `blaz1`. The result looked sensational.

Geometric verification found **38 pairs of overlapping parts**. The cause: rotation about the centroid immediately drove a part into its neighbour, and the settling routine could only slide left and **never repaired it**. That was not a denser packing; it was parts driven into one another.

8.2 Gravity that passed through neighbours

After the fix we measured settling alone, without rotation. On `blaz1` it gave **+2.98 pp**. Again it looked good.

Verification: **17 overlapping pairs**. The cause was subtler — in one branch of the code, neighbours were searched within a radius five times smaller than the permitted movement. Parts were passing through objects that were never tested.

8.3 How it actually is

After fixing both faults and adding a guard that rejects **any** layout containing an overlap:

instance	`sparrow`	+ gravity	+ vibration
albano	88.91%	88.91%	88.91%
blaz1	84.12%	84.21%	84.21%
fu	91.91%	91.93%	91.93%
shirts	89.53%	89.55%	—
swim	77.42%	77.43%	—

Settling gives between 0.01 and 0.09 points. Vibration — across 463 taps in total — gave **zero improvements**. **H5 refuted.**

8.4 Clustering fed to `sparrow`

Since clustering was the one thing that improved **our** engine (+2.12%), the natural next step was to test whether it also improves `sparrow`, which does not do it and starts from its own simple constructive placement on the raw part list.

Pairs were fused exactly, on continuous geometry rather than on the raster; instances where the outlines did not merge into a single figure without a hole were rejected. The net-area check passed in all five cases.

instance	shapes	pieces	`sparrow`	`sparrow` + clusters	difference
shirts	8 → 10	99 → 77	89.01%	88.59%	−0.42 pp
swim	10 → 10	48 → 48	76.95%	75.91%	−1.04 pp
dagli	10 → 14	30 → 25	87.34%	87.70%	+0.36 pp
trousers	17 → 16	64 → 54	90.98%	90.27%	−0.72 pp
jakobs1	25 → 23	25 → 23	89.07%	89.07%	±0.00 pp
average			**86.67%**	**86.31%**	**−0.36 pp**

Clustering made `sparrow` worse — it helped on one instance out of five and hurt on average. In our view this is the single most interesting result of the study.

The same technique yields **+2.12% for our engine and −0.36 pp for `sparrow`**. The explanation: clustering **commits in advance** to pairing two parts and **takes freedom away from the solver**. A constructive decoder could not find those tight pairs by itself, so the crutch helps. A separation solver finds them on its own, through continuous pushing — so the same crutch becomes a shackle.

> ***The value of a structural heuristic is inversely proportional to the strength of the solver receiving it.***

Together with 8.1–8.3, this completes the picture: **all three of our ideas for wrapping `sparrow` measured at zero or below.** We could not improve it from the outside in any way.

8.5 Why vibration fails, when tapping a jar works

This is the most interesting technical conclusion of the exercise.

Tapping a jar densifies its contents because the grains move **all at once** and can **momentarily squeeze through** the loosened structure. Our correct implementation enforces feasibility at every instant — so nothing can ever pass anything else, the structure is frozen, and every rotation is pulled back by gravity to where it started.

The broken version "worked" **solely because it let parts interpenetrate permanently**. The correct mechanism requires allowing overlap **temporarily** and paying it down afterwards.

And that is precisely what `sparrow` does in its exploration phase. The physical intuition was sound — it turned out that the best existing algorithm already implements it, correctly.

---

9. Does the starting layout matter?

A natural follow-up: if we cannot beat `sparrow`, can we at least give it a better starting point — from our raster engine, or from a trained model?

We measured its run-to-run spread. `sparrow` launches several independent runs from different random starting layouts; if they all land in the same place, the starting point cannot matter.

instance	six runs, 60 s each	standard deviation	spread
swim	74.885% … 76.069%	0.393 pp	1.18 pp
jakobs1	89.070% … 89.074%	**0.002 pp**	**0.004 pp**

On `jakobs1`, six completely different random starts converge to the same answer to three decimal places. The exploration phase washes the starting layout out almost entirely. Additionally, the compression phase contributes only +0.03 to +0.22 pp — essentially all the work happens in exploration.

There is nothing there for a better starting layout to buy. This closes, on measurement rather than opinion, the most obvious remaining avenue — including the idea of training a neural network to produce that start.

9.1 A note on the greedy placement rule

A related question: our decoder places each part where it minimises a chosen criterion. Would a different criterion help? We implemented the classic "Tetris" rule — place the part so that the total occupied bounding area is smallest — alongside our bottom-left rule, and gave each the same budget and seeds.

instance	bottom-left	minimise area	minimise length
jakobs1	79.88%	**80.31%**	79.88%
shirts	**80.44%**	79.80%	80.44%
dagli	**81.84%**	81.05%	81.84%
swim	**65.18%**	64.90%	65.18%
average	**76.83%**	76.52%	76.83%

A wash — 0.31 points worse on average, better on one instance out of four. (Minimising the resulting strip length turns out to be exactly equivalent to bottom-left, which is why those two columns are identical.)

The ceiling is set by greediness itself, not by which greedy rule you choose. This matters for the neural-network question: a network trained to pick placements can at best reproduce the exact answer our decoder already computes by exhaustive search over every position, and inherits the same ceiling.

10. What we failed to build

H6 assumed we could escape the constructive ceiling through overlap minimisation on the raster. We wrote a kernel based on thermometer coding across bit planes, which reduces all overlap arithmetic to word operations and `popcount`. The kernel passed its control test — for a feasible layout it returns zero.

The algorithm, however, did not work. After compressing the strip, the overlap stood at 247 pixels and **remained 247 after both 6 and 120 sweeps**. Greedy descent freezes immediately: in a dense layout every one of the 121 candidate positions examined lies on another part, so no move is an improvement and the whole board stands still.

The conclusion matches the literature we reached afterwards: in this family of methods **the guiding mechanism is the algorithm**. Without penalty weights (guided local search) or full penetration maps that let a stuck part jump to the globally best position, overlap minimisation on its own does not move.

---

11. Conclusions

1. Throughput was not the bottleneck — representation was. Four independent measurements say the same thing: more compute does not buy quality. As long as the decoder has a ceiling, you can feed it any number of evaluated layouts and nothing changes. This is a warning against the commonest error in this field: adding cores where the problem is structural.

2. Demanding feasibility at every instant is a muzzle. Every denser packing is separated from the current one by states that are temporarily worse — overlapping or looser. The constructive approach forbids the first, greedy descent forbids the second. We forbade ourselves precisely the states one has to pass through. `sparrow` wins because it does not forbid them.

3. Changing the input beats changing the search — but only for a weak solver. Clustering was the only thing that moved **our** number. Two different metaheuristics and three times the compute budget gave nothing. The same technique fed to `sparrow` **made it worse by 0.36 points**. Generalising: the value of a structural heuristic is inversely proportional to the strength of the solver receiving it. That is a practical caution against carrying "proven tricks" between engines of different strength.

4. A correctness guard is part of the measurement, not an extra. Twice we measured a success that did not exist, and both times only the geometric test on real polygons caught it. In nesting, **a result without overlap verification is not a result** — it is a number.

5. A bar set too low stops development. Measured against Deepnest, we felt we were near the front. Measured against `sparrow`, we saw almost nine percentage points missing. Only the second number is information you can work with.

6. Raster has its place in nesting — just not at the core. It lost the contest for the packing algorithm. But linear preparation cost, natural handling of parts nested inside the holes of other parts, and a structural guarantee of non-overlap make it a very good tool **before** nesting (clustering) and **after** it (lead-in

placement, pierce points, collision checking).

---

12. What we build from here

An exercise meant to pick a winner instead produced a division of labour — and it is not the one we assumed at the start.

stage	technology	status	rationale
packing core	`sparrow` (MIT)	**measured**	+8.78 pp over our best, 13/13
lead-ins, pierce points, collision checks	our raster	design	conservative dilation guarantees non-collision by construction
instant preview	learned model	hypothesis, **gated**	see section 9 — the gate has not been passed
~~clustering before `sparrow` ~~	—	**rejected**	measured at −0.36 pp

We mark the "measured / design / hypothesis" distinction deliberately. The only thing confirmed by measurement is `sparrow`'s superiority as a core. The rest is a plan, and a plan should not be presented as a result.

`sparrow` is MIT-licensed, which permits use in a closed commercial product provided the copyright notice is preserved. There is no reason to rewrite a year of someone else's work from scratch just to arrive at the point where improvement can begin. This is how research is done — you take the best published result and push it further.

It is worth noting separately where **our** work retains lasting value despite the loss: the raster engine beat our previous generation by 7.13 points and does so three orders of magnitude faster than its CPU equivalent. As a fast path and as a finishing layer it remains useful — simply not as the core.

---

13. Limitations of this study

Stated, because without them the numbers above would be overclaimed.

- **One machine, one budget.** 60 seconds per instance. Proportions may shift at longer runtimes; `sparrow`'s own published experiments use 1200 seconds.
- **Thirteen instances.** A classic but incomplete set; seven entries from our earlier measurements have no counterpart here.

- **One random seed per instance** in the head-to-head. Variance was measured separately (standard deviation 10–13 px on the production job), but not for every benchmark instance.
- **OpenCL portability is by design, not demonstrated.** The code is written in OpenCL 1.2 core without extensions and verified bit-identical against CUDA, but tested only on NVIDIA hardware.
- **We did not attempt to tune `sparrow`.** It was run with default settings. Its results could be better still.

References

1. *An open-source heuristic to reboot 2D nesting research* (sparrow), arXiv:2509.13329 (2025) — J. Gardeyn, KU Leuven. Code: https://github.com/JeroenGar/sparrow
2. *Decoupling Geometry from Optimization: an Open-Source Collision Detection Engine for 2D Irregular Cutting and Packing* (jagua-rs), INFORMS Journal on Computing. Code: https://github.com/JeroenGar/jagua-rs
3. *An iterated local search algorithm based on nonlinear programming for the irregular strip packing problem* (ILSQN), Discrete Optimization (2009).
4. *Solving the irregular strip packing problem via guided local search for overlap minimization*, International Transactions in Operational Research (2009).
5. *The Dotted-Board Model: A new MIP model for nesting irregular shapes*, International Journal of Production Economics (2013).
6. *Innovative raster penetration map applied to the irregular packing problem*, European Journal of Operational Research (2019).
7. *Extreme Point-Based Heuristics for Three-Dimensional Bin Packing*, INFORMS Journal on Computing (2008).
8. *Two-dimensional polygon classification and pairwise clustering for pairing in ship parts nesting*, Journal of Intelligent Manufacturing (2023).

For entries 3–8 we give titles and venues without full author lists — citations should be completed before any academic publication. Here they serve to indicate the literature against which we positioned our own results.