Proof that $(20, 21, 29)$ admits no four-piece rookwise-connected discrete Pythagorean dissection

Shisheng Li
daizisheng@gmail.com

April 11, 2026 at 22:53

1. Introduction.

Theorem. *The Pythagorean triple* $(20, 21, 29)$ *does not admit a four-piece rookwise-connected discrete dissection.*

This theorem is proved by exhaustive computational verification: we show that no valid 4-tuple of transforms exists for any of the four split cases. A brute-force search over all possible transforms would be infeasible (the raw search space exceeds 10^{12} candidates), so the bulk of this document develops the theoretical tools—symmetry reductions, private-region analysis, target-coverage filters, and a bipartite matching formulation—that reduce the search to about 13 million candidates, each resolved in microseconds by degree-1 propagation. The result is a self-contained C program (produced by `ctangle`) that completes the verification in under four minutes.

This answers Exercise 89 [M46] of Knuth's *The Art of Computer Programming*, Volume 4, Pre-Fascicle 9B (§7.2.2.8) in the negative. The exercise asks: "Are [four-piece rookwise connected discrete Pythagorean] dissections possible for *all* (u, v, w) with $u^2 + v^2 = w^2$?"

We begin by defining all necessary concepts from scratch.

2. Discrete dissections. Think of the Euclidean plane as an infinite grid of unit squares, called *pixels*. A *discrete dissection* of two shapes A and B, each consisting of N pixels, is a partition of their pixels into *pieces* (groups of pixels), such that each piece in A is congruent to the corresponding piece in B. Each piece is moved from A to B by a rigid transformation.

The allowed transformations come from the dihedral group D_8—the eight symmetries of a square: four rotations (0°, 90°, 180°, 270°) and four reflections—followed by a translation $\sigma_{a,b}\colon (x,y) \mapsto (x{+}a, y{+}b)$. Following Knuth, we write $\alpha\sigma_{a,b}$ for the composite transformation that *first* applies $\alpha \in D_8$, *then* translates by (a,b):

$$(\alpha\sigma_{a,b})(x,y) = \alpha(x,y) + (a,b).$$

This is left-to-right application order. In standard function composition, where $(f \circ g)(x) = f(g(x))$ applies g first, we have $\alpha\sigma_{a,b} = \sigma_{a,b} \circ \alpha$.

The dihedral group D_8. The *dihedral group* D_8 consists of the eight symmetries of a square. It has two generators: a 90° clockwise rotation ρ and a reflection τ across the main diagonal. On a grid of $(m{+}1) \times (m{+}1)$ pixels with coordinates $0 \le x, y \le m$ (so an $n{\times}n$ grid has $m = n{-}1$; in particular the $29{\times}29$ source grid has $m = 28$), the eight elements and their formulas are:

Rotations (powers of ρ):

$$\begin{aligned} e&\colon (x,y) \mapsto (x,\ y) && \text{(identity)},\\ \rho&\colon (x,y) \mapsto (y,\ m{-}x) && \text{(90° clockwise)},\\ \rho^2&\colon (x,y) \mapsto (m{-}x,\ m{-}y) && \text{(180°)},\\ \rho^3&\colon (x,y) \mapsto (m{-}y,\ x) && \text{(270° clockwise)}. \end{aligned}$$

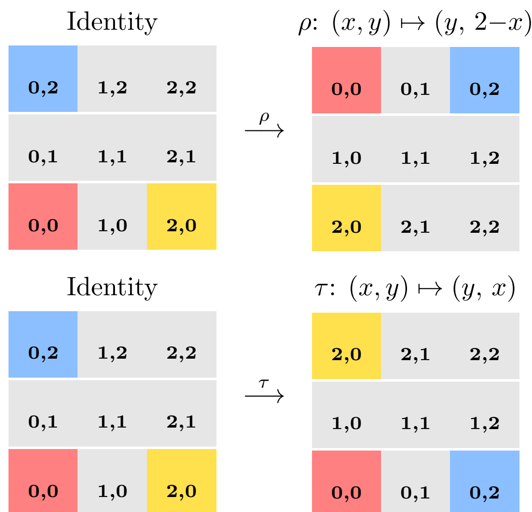
Reflections (a rotation followed by τ):

$$\begin{aligned} \tau&\colon (x,y) \mapsto (y,\ x) && \text{(diagonal } y{=}x),\\ \rho\tau&\colon (x,y) \mapsto (m{-}x,\ y) && \text{(vertical axis } x{=}m/2),\\ \rho^2\tau&\colon (x,y) \mapsto (m{-}y,\ m{-}x) && \text{(anti-diagonal)},\\ \rho^3\tau&\colon (x,y) \mapsto (x,\ m{-}y) && \text{(horizontal axis } y{=}m/2). \end{aligned}$$

These satisfy $\rho^4 = \tau^2 = e$ and $\tau\rho\tau = \rho^3$.

Composition convention. In standard mathematics, fg means "apply g first, then f" (right-to-left). Following Knuth, this document uses *left-to-right* composition throughout: $\rho\tau$ means "apply ρ first, then τ." For example, $\rho\tau$ maps $(x,y) \xmapsto{\rho} (y,\ m{-}x) \xmapsto{\tau} (m{-}x,\ y)$, confirming the vertical-axis reflection.

Visualizing ρ and τ. To make these transformations concrete, consider a $3{\times}3$ grid with $m = 2$. Each cell shows coordinates (x,y), with three pixels colored for tracking:



Each cell in the result is labeled with the *original* coordinates of the pixel now occupying that position. Under ρ (90° clockwise): **0,0** moves to the top-left, **0,2** moves to the top-right, and **2,0** moves to the bottom-left. Under τ (diagonal reflection): **0,0** stays at the bottom-left, **0,2** moves to the bottom-right, and **2,0** moves to the top-left.

After a rotation/reflection, a *shift* $\sigma_{a,b}$ translates the result into a target grid. For example, $\rho\sigma_{1,0}$ first rotates 90° clockwise, then shifts right by 1:

The three source pixels land at target positions shifted right by 1. Only the part that falls within the target grid counts as "covered."

Lemma (rigid motions of the grid; see Knuth §7.2.2.8). *Every isometry of the plane that maps the integer lattice* $\mathbf{Z}^2$ *to itself has the form* $\alpha\sigma_{a,b}$ *for some* $\alpha \in D_8$ *and* $(a, b) \in \mathbf{Z}^2$.

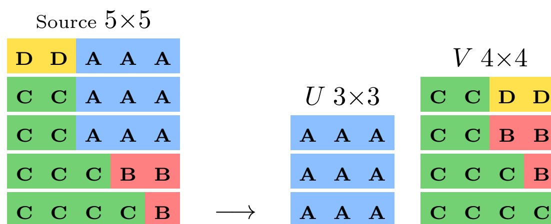
This justifies Knuth's decomposition $\phi_k = \alpha_k\sigma_{a_k,b_k}$: we need only enumerate 8 rotations times all valid shifts, which is a finite search.

Connectivity. A piece is *rookwise connected* if any two of its pixels can be reached from each other by a sequence of horizontal and vertical steps through pixels of the same piece (like a rook in chess—diagonal steps are not allowed).

3. Pythagorean dissections. A *Pythagorean triple* is a triple (u, v, w) of positive integers with $u^2+v^2 = w^2$. For such a triple, a *discrete Pythagorean dissection* partitions the $w \times w$ square into pieces that tile both the $u \times u$ and $v \times v$ squares.

More precisely: let A be the $w \times w$ grid (the "source"), and let the "target" consist of two disjoint grids—a $u \times u$ grid (called U) and a $v \times v$ grid (called V). Since $u^2 + v^2 = w^2$, both sides have exactly $N = w^2$ pixels. A dissection assigns each source pixel to a piece, and each piece is moved to either U or V by a D_8 transformation plus translation, such that each target pixel is covered exactly once.

Example: $(3, 4, 5)$**.** The simplest Pythagorean dissection cuts a 5×5 source into four pieces that tile a 3×3 and a 4×4 target. Here is one solution (1+3 split), with each piece shown in a distinct color:



Both target grids use coordinates starting at $(0, 0)$: U has $0 \le x, y \le 2$ and V has $0 \le x, y \le 3$. Piece A (blue, 9 pixels) fills all of U via the identity rotation and shift $\sigma_{-2,-2}$ (source pixel $(2, 2)$ maps to U-pixel $(0, 0)$). Pieces B (3 pixels), C (11 pixels), D (2 pixels) map to V via shifts $\sigma_{-1,1}$, $\sigma_{0,0}$, $\sigma_{2,-1}$ respectively. Each piece is rookwise connected and maps entirely to one target ("no jumping").

No jumping. Each piece maps entirely to one target square—it cannot jump between U and V (Exercise 87). To see why: a single transform $\phi = \alpha\sigma_{a,b}$ maps every pixel of its piece by the *same* rigid motion. Since U and V are separate grids in distinct coordinate systems—each with its own origin at $(0, 0)$—a rigid motion that maps one pixel into U maps nearby pixels into the same coordinate space; it cannot simultaneously place other pixels into V's coordinate space. Thus each piece's pixels all land in one target. A piece assigned to U uses only one transformation $\phi = \alpha\sigma_{a,b}$ (where $\alpha \in D_8$ and $\sigma_{a,b}$ shifts by (a, b), following Knuth's notation) to map all its pixels to U-target positions.

The question. Frederickson (Exercise 88 of the same fascicle) proved that four-piece rookwise-connected dissections exist for two infinite families of Pythagorean triples. Exercise 89 [M46] asks whether they exist for *all* triples. The smallest unsolved cases are $(20, 21, 29)$, $(28, 45, 53)$, $(33, 56, 65)$, and $(48, 55, 73)$.

4. Structural properties of transforms. Before describing the computational framework, we establish some elementary geometric properties that hold for *any* Pythagorean triple (u, v, w) with $u < v < w$ and $u^2 + v^2 = w^2$. We always have $u \neq v$: if $u = v$ then $w^2 = 2u^2$, so $w = u\sqrt{2}$, contradicting integrality. (In particular, $(20, 21, 29)$ is primitive—$\gcd(20, 21) = 1$—and satisfies $20 < 21 < 29$.)

Transforms and rectangles. Each transform $\phi = \alpha\sigma_{a,b}$ consists of a D_8 element α followed by a translation $\sigma_{a,b}$. The element α maps the $w\times w$ source square to itself (rotations and reflections preserve the square), so $\alpha(W)$ is still a $w\times w$ axis-aligned square occupying the same position. The translation then shifts this square by (a, b).

The image $\phi(W)$ is thus a $w\times w$ square, and its intersection with a $u\times u$ (or $v\times v$) target grid is a rectangle (possibly a square, possibly empty). This rectangle is $\mathrm{tgt}(\phi)$—the target footprint of ϕ. Taking the preimage under ϕ, the source footprint $\mathrm{src}(\phi) = \phi^{-1}(\mathrm{tgt}(\phi))$ is also a rectangle in the $w\times w$ grid. (Since ϕ is a D_8 element plus translation, it preserves the rectangular shape.)

Source rectangles are small. The dimensions of $\mathrm{tgt}(\phi)$ are at most $u\times u$ (or $v\times v$), since it is the intersection of a translated square with the target grid. Since ϕ is an isometry, $\mathrm{src}(\phi)$ has the same dimensions. In particular, each side of $\mathrm{src}(\phi)$ has length at most $\max(u, v) = v < w$.

Each corner needs its own transform. Since each piece uses a single transform (Section 3, "No jumping") and $v < w$, no source rectangle $\mathrm{src}(\phi)$ can simultaneously cover two corners of the $w\times w$ grid. (To cover two adjacent corners, the rectangle would need to span the full width w in one direction, but its dimensions are bounded by $v < w$. Two opposite corners are even farther apart.)

In any valid four-piece dissection, all w^2 source pixels must be covered—in particular, the four corner pixels of W. Since no single source rectangle can reach two corners, each corner must be covered by a different transform. This gives us a fundamental constraint:

Corner assignment property. *In any four-piece discrete Pythagorean dissection, the four corner pixels of the $w\times w$ source grid are covered by four distinct transforms, one corner per piece.*

Corollary: the lone piece occupies a corner block. In the 1+3 split (one piece to U, three to V), the single U-piece must cover all of U, so $\mathrm{src}(\phi)$ has area u^2 with dimensions $\leq u$, giving an exact $u\times u$ square. By the corner assignment property it covers one corner of W, so it occupies a $u\times u$ corner block. Symmetrically, in the 3+1 split the single V-piece occupies a $v\times v$ corner block.

Moreover, by the rigidity of isometries—an isometry of $\mathbf{Z}^2$ is determined by the images of two adjacent points, since these fix both the rotation/reflection and the translation—the transform for the lone piece is completely determined (up to the choice of which corner it occupies) by the D8 element α, and the shift (a, b) is forced so that the corner block maps exactly onto the target grid.

5. Symmetry groups. The search space of quadruples $(\phi_1, \phi_2, \phi_3, \phi_4)$ has a natural symmetry group G under which feasibility is invariant. Understanding G lets us reduce the search by a factor of up to $|G|$.

We define a *feasibility function* F that maps a quadruple (together with its U/V partition) to 1 if a valid coloring exists, and 0 otherwise. The *automorphism group* is $\mathrm{Aut}(F) = \{g \mid F(g \cdot \Phi) = F(\Phi) \text{ for all } \Phi\}$.

2+2 split group.

Theorem. *For the* 2+2 *split with* $u \neq v$ (e.g. $(u, v, w) = (20, 21, 29)$), *the full automorphism group is*

$$G = D_8^{\mathrm{src}} \times D_8^{U\text{-tgt}} \times D_8^{V\text{-tgt}} \times \mathbf{Z}_2 \times \mathbf{Z}_2,$$

of order $8 \times 8 \times 8 \times 2 \times 2 = 2048$.

Proof. G has five independent factors. We verify that each preserves F and that they act on disjoint domains (hence commute).

(1) **Source-grid symmetry D_8^{src} (order 8).** The $W \times W$ source grid is a square with dihedral symmetry group D_8 (four rotations and four reflections). If $g \in D_8$ and $\phi = \alpha\sigma_{a,b}$ is a transform, define the conjugate transform $g \cdot \phi$ by

$$(g \cdot \phi)(p) = \phi(g^{-1}(p))$$

for each source pixel p. In other words, if ϕ maps source pixel q to target pixel t, then $g \cdot \phi$ maps $g(q)$ to the same target pixel t (substitute $p = g(q)$). This is well-defined because g is a bijection on the $W \times W$ source grid.

Concretely, if $\phi = \alpha\sigma_{a,b}$ then $(g \cdot \phi)(p) = \alpha(g^{-1}(p)) + (a, b) = (g^{-1}\alpha)(p) + (a, b)$, so the effect is to *prepend* the same g^{-1} to the D_8 index of each ϕ_i, leaving translations unchanged. (The constants of g^{-1} and α are absorbed into the constant of the composite element $g^{-1}\alpha$, since both act on the same grid; only the D_8 index changes, not the translation (a, b).) Choosing $g = \alpha_0$ makes the first transform's index become $\alpha_0^{-1}\alpha_0 = e$ (identity); this is one of the symmetry reductions used in the program.

To see that F is preserved: suppose $(\phi_1, \dots, \phi_4)$ has a valid matching $\mu\colon s \mapsto t$, where μ assigns each source pixel s to a target pixel t such that every source and target pixel is matched exactly once. Define $\mu'(g(s)) = \mu(s)$ for all s. Then μ' is a valid matching for $(g \cdot \phi_1, \dots, g \cdot \phi_4)$: every source pixel $s' = g(s)$ is matched to $\mu'(s') = \mu(g^{-1}(s'))$, and every target pixel is still covered exactly once (since g is a bijection on source pixels but acts as the identity on target pixels). Conversely, any matching for the conjugate induces one for the original via g^{-1}. Thus $F(\phi_1, \dots, \phi_4) = F(g \cdot \phi_1, \dots, g \cdot \phi_4)$.

(2) **U-target symmetry $D_8^{U\text{-tgt}}$ (order 8).** The U-target grid is also a square ($u \times u$). For $h \in D_8$ acting on U-target coordinates, define $h \star \phi$ for a U-transform by $(h \star \phi)(p) = h(\phi(p))$ for each source pixel p. This appends h *after* the transform: in left-to-right notation, $\phi = \alpha\sigma_{a,b}$ becomes $\alpha\sigma_{a,b}h$ (apply α, translate, then apply h).

To convert back to standard form $\beta\sigma_{a',b'}$, we decompose each D_8 element into a *linear part* L (a 2×2 matrix, independent of grid size) and a *constant* $c(m)$ depending on the grid size m:

$$\begin{aligned} e(x,y) &= (x,y), & \rho^2(x,y) &= (m-x, m-y),\\ \rho(x,y) &= (y, m-x), & \rho^3(x,y) &= (m-y, x),\\[6pt] \tau(x,y) &= (y,x), & \rho^2\tau(x,y) &= (m-y, m-x),\\ \rho\tau(x,y) &= (m-x, y), & \rho^3\tau(x,y) &= (x, m-y). \end{aligned}$$

Each formula has the form $\alpha(p) = L_\alpha \cdot p + c_\alpha(m)$ where L_α is a 2×2 matrix of ± 1's and 0's (the *linear part*, independent of m) and $c_\alpha(m)$ is a constant involving m (e.g., ρ has $L_\rho = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$ and $c_\rho(m) = (0, m)$). Since h acts on the $u \times u$ grid ($m = u-1$) while α acts on the $w \times w$ grid ($m = w-1$), their constants use different m. Expanding:

$$h(\alpha(p) + (a,b)) = L_h\bigl(L_\alpha \cdot p + c_\alpha(w-1) + (a,b)\bigr) + c_h(u-1) = (L_h L_\alpha) \cdot p + L_h\bigl(c_\alpha(w-1) + (a,b)\bigr) + c_h(u-1).$$

The new D_8 index β has $L_\beta = L_h L_\alpha$ (the same composition regardless of m), and the new translation is

$$(a', b') = L_h\big(c_\alpha(w-1) + (a, b)\big) + c_h(u-1) - c_\beta(w-1).$$

Thus h changes *both* the D_8 index and the translation—unlike factor (1), where only the index changed. In the program, choosing $h = \alpha_1^{-1}$ (for the first U-transform) makes its index become identity; this is the second symmetry reduction.

This operation preserves:

(a) *Source coverage.* $\mathrm{src}(h \star \phi) = \mathrm{src}(\phi)$, since h acts only on the target side.

(b) *Target coverage.* Since h is a bijection on U-target pixels, the pair $(h \star \phi_1, h \star \phi_2)$ covers all of U if and only if (ϕ_1, ϕ_2) does.

(c) *Matching feasibility.* The bipartite graph is transformed by h on the target side. Since h is a bijection, any perfect matching for the original induces one for the image, and vice versa.

Crucially, h acts only on U-transforms and does not affect the V-side transforms. So $D_8^{U\text{-tgt}}$ acts independently of all other factors.

(3) **V-target symmetry $D_8^{V\text{-tgt}}$ (order 8).** By the same argument with the 21×21 V-target grid.

(4) **U-pair exchange $\mathbf{Z}_2$ (order 2).** The pair $\{\phi_1, \phi_2\}$ is unordered: swapping $\phi_1 \leftrightarrow \phi_2$ gives the same dissection (just relabeling colors 1 and 2).

(5) **V-pair exchange $\mathbf{Z}_2$ (order 2).** Similarly, $\phi_3 \leftrightarrow \phi_4$ gives the same dissection.

(6) **No U–V exchange.** Since $u = 20 \neq 21 = v$, we cannot swap the U-pair with the V-pair: they map to targets of different sizes.

Commutativity. Each factor induces permutations on a distinct set of bipartite-graph nodes: D_8^{src} permutes source nodes (W-pixels); $D_8^{U\text{-tgt}}$ permutes U-target nodes; $D_8^{V\text{-tgt}}$ permutes V-target nodes; the two $\mathbf{Z}_2$ factors relabel transforms within each pair (recoloring pieces without moving pixels). Although multiple factors may affect the same transform's parameters (e.g., both D_8^{src} and $D_8^{U\text{-tgt}}$ change a U-transform's D_8 index), they commute because their induced actions on the three disjoint node sets (W, U, V) are independent. Therefore $G = D_8^{\mathrm{src}} \times D_8^{U\text{-tgt}} \times D_8^{V\text{-tgt}} \times \mathbf{Z}_2 \times \mathbf{Z}_2$ of order $8 \times 8 \times 8 \times 2 \times 2 = 2048$. ■

6. 1+3 split group.

Theorem. *For the* 1+3 *split with $u \neq v$, the full automorphism group is*

$$G_{1+3} = D_8^{\mathrm{src}} \times D_8^{U\text{-tgt}} \times D_8^{V\text{-tgt}} \times S_3,$$

of order $8 \times 8 \times 8 \times 6 = 3072$.

The proof follows the same pattern as the 2+2 case. The four independent factors are: D_8^{src} (conjugating transforms on the source grid), $D_8^{U\text{-tgt}}$ (rotating the single U-block's target), $D_8^{V\text{-tgt}}$ (rotating the three V-transforms' shared target), and S_3 (permuting the three V-transforms among themselves). Since $u \neq v$ and the split is asymmetric (1 U-piece vs. 3 V-pieces), no U–V exchange is possible.

7. 3+1 split group.

Theorem. *For the* 3+1 *split,*

$$G_{3+1} = D_8^{\mathrm{src}} \times D_8^{U\text{-tgt}} \times D_8^{V\text{-tgt}} \times S_3,$$

of order $8 \times 8 \times 8 \times 6 = 3072$.

By the same argument as 1+3, with $U \leftrightarrow V$.

8. The framework. The proof of our main theorem rests on a general computational framework that we call the *G-invariant filter pipeline*. This framework applies to any discrete Pythagorean dissection problem and can be tuned for different triples. We describe its components here before applying them to $(20, 21, 29)$.

Setting. Let X be a finite *search space*—for a given split type, the set of all transform quadruples $\Phi = (\phi_1, \phi_2, \phi_3, \phi_4)$ with each ϕ_k of the form $\alpha_k \sigma_{a_k, b_k}$. Let $F \colon X \to \{0, 1\}$ be a *feasibility function*: $F(\Phi) = 1$ iff Φ admits a valid dissection (a perfect matching in the bipartite graph between source and target pixels). The goal is to determine whether $F(\Phi) = 1$ for any $\Phi \in X$.

Tool 1: Symmetry and orbit reduction. A group G acts on X and preserves F: $F(g \cdot \Phi) = F(\Phi)$ for all $g \in G$, $\Phi \in X$. Therefore F is constant on G-orbits, and we need test only one *orbit representative* per orbit. This reduces the search from $|X|$ to $|X/G|$, a factor of up to $|G|$.

Tool 2: Space decomposition. If the search space factors as $X = X_1 \times X_2$ (e.g., U-pairs $\times$ V-pairs for the 2+2 split), we can apply *independent filters* to each factor before forming the Cartesian product. Filters that depend on both factors (e.g., source coverage, force intersection) are applied only to the surviving pairs. The group may also decompose: if $G = G_1 \times G_2$ with G_i acting on X_i, then orbit reduction on each factor is independent.

Tool 3: G-invariant filters. A predicate $P \colon X \to \{0, 1\}$ is *G-invariant* if $P(\Phi) = P(g \cdot \Phi)$ for all $g \in G$, $\Phi \in X$. Equivalently, $\{\Phi : P(\Phi) = 1\}$ is a *G-stable subset* (a union of complete orbits).

Why G-invariance matters. If a filter P is G-invariant, then orbit reduction and filtering *commute*: applying P to orbit representatives yields the same surviving orbits as applying P to all elements and then reducing. In symbols,

$$\{[x] \in X/G : P(x) = 1\} = \{[x] \in X/G : P(\mathrm{rep}(x)) = 1\}$$

where $\mathrm{rep}(x)$ is any orbit representative. A filter that is *not* G-invariant could eliminate an orbit representative while leaving a feasible element in the same orbit undetected—a soundness violation.

9. The $(20, 21, 29)$ case — verification program. Consider $(u, v, w) = (20, 21, 29)$ with $20^2 + 21^2 = 400 + 441 = 841 = 29^2$. The source is a 29×29 square W (841 pixels), with targets U ($20 \times 20 = 400$ pixels) and V ($21 \times 21 = 441$ pixels).

Theorem. $(20, 21, 29)$ *does not admit a four-piece rookwise-connected discrete Pythagorean dissection.*

10. Split cases. A four-piece dissection assigns each piece to one of U or V. Since there are four pieces, the only possible split types are: 2+2 (two pieces to each target), 1+3 (one to U, three to V), and 3+1 (three to U, one to V). The 0+4 and 4+0 cases are impossible because each target is smaller than the source: $U = 400 < 841 = W$ and $V = 441 < 841 = W$, so neither target alone can accommodate all 841 source pixels. We prove each remaining split type infeasible.

11. Corner assignment. Each piece must be rookwise-connected and occupy a contiguous region of W. Since W has four corners and each piece covers at most one corner (by the corner assignment property of Section 4: each source rectangle has side length at most $v = 21 < w = 29$, so it cannot span the full width of W to reach two corners), the four pieces biject onto the four corners. In the 2+2 case there are two sub-cases: the two same-target pieces sit at *diagonal* corners (e.g. $(0, 0)$ and $(28, 28)$) or *adjacent* corners (e.g. $(0, 0)$ and $(0, 28)$).

12. Private regions. For each corner, the *private region* is the set of W-pixels reachable only by the piece at that corner—no other piece's transform can map them into any target square. These pixels are forced into that piece.

Derivation (diagonal 2+2, corner $(0, 0) \to V$). A source pixel (x, y) can potentially be claimed by a piece at another corner only if that piece's source rectangle contains (x, y). We check each of the three other pieces:

- *V-piece at* $(28, 28)$ (diagonal opponent): source rectangle has at most $v = 21$ pixels per side and contains $(28, 28)$, so $x \geq 28 - (v-1) = 8$ and $y \geq 8$. A pixel with $x \leq 7$ or $y \leq 7$ is unreachable by this piece.
- *U-piece at* $(0, 28)$: source rectangle has at most $u = 20$ pixels per side and contains $(0, 28)$, so $x \leq 19$ and $y \geq 28 - (u-1) = 9$. A pixel with $y \leq 8$ is unreachable by this piece.
- *U-piece at* $(28, 0)$: by symmetry with the above, $x \geq 9$ and $y \leq 19$. A pixel with $x \leq 8$ is unreachable by this piece.

Combining: a pixel with $x \leq 7$ and $y \leq 7$ is unreachable by all three, hence private. On the boundary $x = 8$: the U-pieces still cannot reach it ($x = 8 \leq 8$ for $(28, 0)$; $y \leq 8$ for $(0, 28)$ when $y \leq 7$), but the V-piece at $(28, 28)$ requires $y \geq 8$, so only $(8, 8)$ is reachable (by that V-piece alone). By the same argument with $x \leftrightarrow y$, on the boundary $y = 8$ only $(8, 8)$ is reachable. Hence the private region is $[0, 8]^2 \setminus \{(8, 8)\}$, comprising $9 \times 9 - 1 = 80$ cells.

The private regions for all four split cases are computed analogously and are listed as *priv_def* constants in the code. The sizes vary: in the diagonal 2+2 case, V-pieces (at diagonal corners) have 80-cell private regions while U-pieces have 64. This asymmetry arises because $u \neq v$: the thresholds $28 - (v-1) = 8$ and $28 - (u-1) = 9$ differ by 1, producing a $9\times 9-1 = 80$ region for V-corners versus an $8\times 8 = 64$ region for U-corners.

13. Bipartite matching formulation. Given a candidate 4-tuple of transforms $(\phi_1, \phi_2, \phi_3, \phi_4)$, we construct a bipartite graph on $W + U + V = 1682$ nodes. Each transform ϕ_i contributes edges: for every W-pixel p in ϕ_i's source, connect p to $\phi_i(p)$ in the appropriate target square. A valid dissection corresponds to a *perfect matching* in this graph (every node matched exactly once).

We test feasibility by *degree-1 propagation*: any node with exactly one neighbor must be matched to it (in any perfect matching, a degree-1 node's unique neighbor is its only possible partner); we force this match, remove both nodes, update neighbors' degrees, and repeat. If any node reaches degree 0, the graph has no perfect matching and the 4-tuple is infeasible.

Soundness proof. We show that if a perfect matching M exists in the original graph, then propagation never produces a degree-0 node. The argument is by induction on propagation steps. At each step a degree-1 node v is found with unique neighbor u. Since M is a perfect matching, v must be matched in M; its only neighbor is u, so M matches v to u. Removing v and u and their edges yields a smaller graph in which $M \setminus \{(v, u)\}$ is a perfect matching of the remaining nodes. In particular, every remaining node still has at least one neighbor in this restricted matching, so no node reaches degree 0. By induction, propagation cannot encounter a contradiction if a perfect matching exists; equivalently, a contradiction certifies that no perfect matching exists.

Note on rookwise connectivity. The bipartite matching formulation does not enforce the rookwise-connectivity constraint on pieces. This is deliberate: we are proving *impossibility*. If no valid pixel assignment exists even without requiring connectivity, then certainly none exists with the additional connectivity constraint. Connectivity would only need to be checked if survivors remained (to distinguish genuine dissections from spurious ones), but since all candidates are eliminated, the point is moot.

14. Pre-filters. Degree-1 propagation takes $O(W^2)$ time per 4-tuple. To reduce the number of calls, we apply three fast filters:

1. *Private-source containment.* Each transform's source must contain the private region of its corner.
2. *Private-target disjointness.* Two transforms mapping to the same target must have disjoint private-target images (otherwise the same target pixel is forced into two pieces). For 2+2 this is checked pairwise on each target's pair; for 1+3/3+1 it is precomputed as a compatibility matrix and checked via table lookup.
3. *Target corner coverage.* Each transform's source covers certain corners of the target square (precomputed as a 4-bit mask). Two transforms sharing a target can tile it only if their corner masks OR to `#F`; similarly for three transforms. This is a necessary but not sufficient condition for full target coverage, yet it runs in $O(1)$ versus $O(w^2)$ for the exact check. Since infeasible candidates are ultimately rejected by degree-1 propagation, this filter need only be a necessary condition; its role is to reduce the workload, not to be exact.

Together these three pre-filters reduce the number of candidate 4-tuples from over 10^{12} (the raw search space) to about 13 million—a level at which degree-1 propagation (at $\sim 15\,\mu$s per call) completes in a few minutes.

15. Three independent D_8 symmetries. The dihedral group D_8 acts independently on three squares: W (source), U (target), and V (target). Applying $\alpha_w \in D_8(W)$ to the source permutes pixels but preserves the dissection structure; similarly $\alpha_u \in D_8(U)$ rotates/reflects the U target, and $\alpha_v \in D_8(V)$ rotates/reflects V. These three actions commute and are independent.

Each symmetry allows us to fix one transform's D8 index to 0 (the identity element) without loss of generality, giving an 8× reduction in the search space. For 2+2 we use $D_8(W)$ on the V-piece and $D_8(U)$ on the U-piece, for 64× total. For 1+3/3+1 the single piece is already fixed as the identity (using $D_8(W)$), and D_8 of the triple's target gives 8×.

Symmetries actually used. For the 2+2 split, the program exploits four of the five factors of the automorphism group (Section 5): $D_8(W)$ fixes slot 0's index to 0 (8×); $D_8(U)$ fixes slot 1's index to 0 (8×); and the two $\mathbf{Z}_2$ pair exchanges are used implicitly by fixing the corner-to-slot assignment within each pair—e.g., slot 0 is always at corner $(0, 0)$ and slot 2 at $(28, 28)$, rather than also considering the reverse $(2 \times 2 = 4\times)$. Concretely, the private-region table `PRIV_2V2_DIAG` hard-codes the corner assignments (slot 0 at $(0, 0)$, slot 2 at $(28, 28)$ for the V-pair; slot 1 at $(0, 28)$, slot 3 at $(28, 0)$ for the U-pair). Swapping slot 0↔slot 2 produces a $D_8(W)$-equivalent 4-tuple (the 180° rotation ρ^2 exchanges both diagonal pairs of corners), which is already covered by the D_8^{src} factor. Thus the $\mathbf{Z}_2$ exchanges do not enlarge the search—they are absorbed into the existing symmetry reductions. This gives a total reduction of $8 \times 8 \times 4 = 256\times$.

For the 1+3/3+1 splits, $D_8(W)$ fixes the lone piece to the identity transform at corner $(0, 0)$ (8×); S_3 is used implicitly by fixing the assignment of the three remaining slots to the three remaining corners (6×); and D_8 of the triple's shared target fixes one slot's index to 0 (8×), for a total of $8 \times 6 \times 8 = 384\times$.

The remaining unused factor in each case (D_8 of one target) could in principle further reduce the search space, but since the current reductions already yield a tractable search (∼13 million candidates verified in minutes), this was not pursued. (Under-exploiting symmetry affects only efficiency, not soundness: it means we test redundant orbit representatives, but never skip any orbit. The exhaustiveness of the search is therefore unaffected.) For larger triples with bigger search spaces, exploiting these additional symmetries may become worthwhile.

16. Results.

The column "Tested" shows the number of candidate 4-tuples that pass all three pre-filters (private-source containment, private-target disjointness, and target corner coverage) and are submitted to degree-1 propagation. "Survivors" is the number of 4-tuples for which propagation fails to find a contradiction—i.e., potentially valid dissections. A survivor count of 0 means every candidate is provably infeasible.

Case	Tested	Survivors
2+2 diagonal	4,794,036	**0**
2+2 adjacent	4,735,265	**0**
1+3	2,526,032	**0**
3+1	655,850	**0**
Total	12,711,183	**0**

Total running time: 3 minutes 19 seconds on a single core of an Intel Xeon E5-2686 v4 at 2.30 GHz, compiled with `gcc -O3`. The program is fully deterministic (no randomness, no floating-point arithmetic, no platform-dependent behavior): the tested counts and zero-survivor result are reproducible on any conforming C compiler.

All 12,711,183 candidate 4-tuples are eliminated by degree-1 propagation: every one leads to a node with degree 0 (a pixel with no remaining candidate), proving infeasibility. Since the pre-filters are necessary conditions (they never reject a valid dissection) and propagation is sound (it only declares infeasibility when no perfect matching exists), the zero-survivor result is a complete proof.

Note that degree-1 propagation is sound but not complete: it can certify infeasibility but may fail to detect it in some cases (returning "survivor" for a 4-tuple that actually has no perfect matching). For $(20, 21, 29)$, propagation happens to suffice for every candidate. If survivors remained, a full matching algorithm (e.g., Hopcroft–Karp) would be needed as a final check.

17. Cross-validation on smaller triples. As a sanity check, a parameterized version of the same algorithm was run on all four primitive Pythagorean triples with $w < 29$. These are exactly the triples covered by Frederickson's two infinite families (Exercise 88), for which four-piece rookwise-connected dissections are known to exist.

(u, v, w)	Family	Survivors	
$(3, 4, 5)$	(a), p=1	291	
$(5, 12, 13)$	(a), p=2	181	
$(8, 15, 17)$	(b), p=2	40	
$(7, 24, 25)$	(a), p=3	1430	
$(20, 21, 29)$	—	**0**	← first non-Frederickson triple

In every case where a dissection is known to exist, the program correctly produces survivors (degree-1 propagation does not reject valid dissections). The sharp transition to zero survivors at $(20, 21, 29)$—the smallest primitive triple not in either Frederickson family—provides empirical confidence in the program's correctness.

$(20, 21, 29)$ does not admit a four-piece rookwise-connected discrete Pythagorean dissection. Since Exercise 89 [M46] of Knuth's *The Art of Computer Programming*, Volume 4, Pre-Fascicle 9B asks whether such dissections exist for *all* Pythagorean triples, a single counterexample suffices: the answer is **no**.

Remark on other triples and generalizability. Although a single counterexample suffices for Exercise 89, the computational framework developed here—the G-invariant filter pipeline of Section 8—is designed to be general. All components (symmetry reduction, private-region analysis, corner-coverage filters, bipartite matching) are parameterized by (u, v, w) and apply to any Pythagorean triple; indeed, the cross-validation above confirms this for four smaller triples.

The remaining small open cases—$(28, 45, 53)$, $(33, 56, 65)$, $(48, 55, 73)$—are not addressed here. The raw search space grows roughly as w^8 (four transforms, each with $O(w^2)$ shifts), so extending to $w = 53$ would require stronger filters—such as exploiting the currently unused D_8 target-side factor, tighter source-overlap constraints, or region-based pruning beyond the corner private regions—to keep the candidate count tractable. Whether these triples also lack four-piece dissections, or whether $(20, 21, 29)$ is an isolated exception, remains an interesting open question that may shed light on the structural boundary between Frederickson's families and the general case.

18. The program. A *transform_t* stores the D8 index, target flag, and translation. The D8 indices 0–7 correspond to the named elements of Section 2 as follows, using $m = w-1 = 28$ for pixel coordinates $0 \le x, y \le 28$:

idx	element	formula
0	e	$(x,\ y)$
1	ρ	$(y,\ m-x)$
2	ρ^2	$(m-x,\ m-y)$
3	ρ^3	$(m-y,\ x)$
4	τ	$(y,\ x)$
5	$\rho\tau$	$(m-x,\ y)$
6	$\rho^2\tau$	$(m-y,\ m-x)$
7	$\rho^3\tau$	$(x,\ m-y)$

In the code, m appears as $ww - 1 = 28$.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  ⟨ Transform type 19 ⟩
  ⟨ Bipartite graph 20 ⟩
  ⟨ Private regions 21 ⟩
  ⟨ Transform enumeration 23 ⟩
  ⟨ Filters 24 ⟩
  ⟨ 2+2 split 27 ⟩
  ⟨ 1+3 and 3+1 split 28 ⟩

  int main(void)
  {
    long long total = 0;

    setvbuf(stdout, Λ, _IONBF, 0);
    printf("Verifying␣(20,21,29).\n");
    total += run_2v2(1);
    total += run_2v2(0);
    total += run_1v3_or_3v1(1);
    total += run_1v3_or_3v1(0);
    if (total ≡ 0) printf("VERIFIED:␣no␣dissection␣exists.\n");
    else printf("FAILED:␣%lld␣survivors.\n", total);
    return (total ≠ 0);
  }
```

19.

```
#define uu 20
#define vv 21
#define ww 29
#define nodes (ww * ww + uu * uu + vv * vv)
```

⟨ Transform type 19 ⟩ ≡

```
typedef struct {
    int idx;      /* D8 index, 0–7 */
    int is_u;     /* 1 if target is U, 0 if V */
    int dx, dy;   /* translation */
} transform_t;

transform_t make_t(int idx, int is_u, int dx, int dy)
{
    transform_t t;
    t.idx = idx; t.is_u = is_u; t.dx = dx; t.dy = dy;
    return t;
}

void apply_t(transform_t t, int x, int y, int *ox, int *oy)
{
    switch (t.idx) {
    case 0: *ox = x;
        *oy = y; break;
    case 1: *ox = y;
        *oy = ww - 1 - x; break;
    case 2: *ox = ww - 1 - x;
        *oy = ww - 1 - y; break;
    case 3: *ox = ww - 1 - y;
        *oy = x; break;
    case 4: *ox = y;
        *oy = x; break;
    case 5: *ox = ww - 1 - x;
        *oy = y; break;
    case 6: *ox = ww - 1 - y;
        *oy = ww - 1 - x; break;
    case 7: *ox = x;
        *oy = ww - 1 - y; break;
    }
    *ox += t.dx; *oy += t.dy;
}
```

This code is used in section 18.

20. The bipartite graph has 1682 nodes with max degree 4. Each source node has degree at most 4 because it appears in at most 4 transforms' source footprints (one per corner piece). Each target node also has degree at most 4: a target pixel can be the image of at most one source pixel per transform (since each ϕ_i is injective), and there are exactly 4 transforms. Degree-1 propagation forces matching and detects contradictions.

⟨ Bipartite graph 20 ⟩ ≡

```
int deg[nodes], adj[nodes][4], bpq[nodes];
void bp_add(transform_t t)
{
  int x, y, ox, oy, s, off;
  s = t.is_u ? uu : vv;
  off = t.is_u ? ww * ww : ww * ww + uu * uu;
  for (x = 0; x < ww; x++)
    for (y = 0; y < ww; y++) {
      apply_t(t, x, y, &ox, &oy);
      if (ox ≥ 0 ∧ ox < s ∧ oy ≥ 0 ∧ oy < s) {
        int sn = x * ww + y, tn = ox * s + oy + off;    /* source and target node indices */
        if (deg[sn] ≥ 4 ∨ deg[tn] ≥ 4) {
          printf("degree␣overflow:␣sn=%d␣tn=%d␣deg=%d,%d\n", sn, tn, deg[sn], deg[tn]);
          exit(1);
        }
        adj[sn][deg[sn]++] = tn; adj[tn][deg[tn]++] = sn;
      }
    }
}
int bp_propagate(void)
{
  int i, head = 0, tail = 0, node, mate, other, j;
  for (i = 0; i < nodes; i++) {
    if (¬deg[i]) return 0;
    if (deg[i] ≡ 1) bpq[tail++] = i;
  }
  while (head < tail) {
    node = bpq[head++];
    if (deg[node] ≠ 1) continue;    /* A node enqueued at degree 1 may since have been matched
        away (degree 0) or gained nothing (still 1). If deg[node] ≡ 0, its sole neighbor was matched by
        another propagation step; but that step already set deg[node] = 0 and would have returned 0
        if a contradiction arose, so skipping is safe. The ≠ 1 guard handles both cases. */
    mate = adj[node][0];    /* forced match */
    for (i = 0; i < deg[mate]; i++) {
      other = adj[mate][i]; if (other ≡ node) continue;
      for (j = 0; j < deg[other]; j++)
        if (adj[other][j] ≡ mate) {
          adj[other][j] = adj[other][−−deg[other]]; break;
        }
      if (¬deg[other]) return 0;
      if (deg[other] ≡ 1) bpq[tail++] = other;
    }
    deg[node] = 0; deg[mate] = 0;
  }
  return 1;
```

```
}
int bp_test(transform_t a, transform_t b, transform_t c, transform_t d)
{
  memset(deg, 0, sizeof (deg));
  bp_add(a); bp_add(b); bp_add(c); bp_add(d);
  return bp_propagate();
}
```

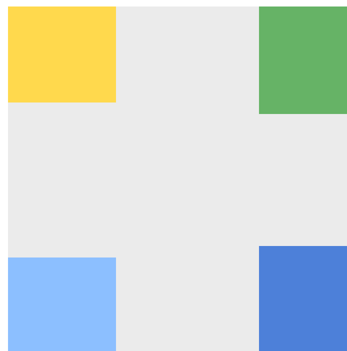
This code is used in section 18.

21. Private regions. Each private region is a rectangle (possibly minus one pixel for L-shaped corners). The diagrams below show each case's private regions on the 29×29 source grid W. Colored cells are private (forced to that piece); gray cells are free (reachable by multiple pieces).

2+2 diagonal ($\phi_1, \phi_3 \to V$; $\phi_2, \phi_4 \to U$):

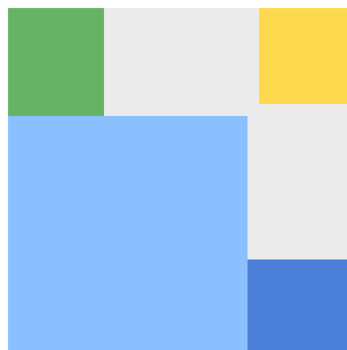
ϕ_1: $[0,8]^2\backslash\{(8,8)\}$ (80), ϕ_2: $[0,7]\times[21,28]$ (64), ϕ_3: $[20,28]^2\backslash\{(20,20)\}$ (80), ϕ_4: $[21,28]\times[0,7]$ (64)

2+2 adjacent ($\phi_1, \phi_3 \to V$; $\phi_2, \phi_4 \to U$):



ϕ_1: $[0,8]\times[0,7]$ (72), ϕ_2: $[21,28]\times[20,28]$ (72), ϕ_3: $[0,8]\times[21,28]$ (72), ϕ_4: $[21,28]\times[0,8]$ (72)

1+3 ($\phi_1 \to U$, identity; $\phi_2, \phi_3, \phi_4 \to V$):



ϕ_1: $[0,19]^2$ (400, full U), ϕ_2: $[0,7]\times[20,28]$ (72), ϕ_3: $[21,28]^2$ (64), ϕ_4: $[20,28]\times[0,7]$ (72)

3+1 ($\phi_1 \to V$, identity; $\phi_2, \phi_3, \phi_4 \to U$):

ϕ_1: $[0,20]^2$ (441, full V), ϕ_2: $[0,8]\times[21,28]$ (72), ϕ_3: $[20,28]^2\setminus\{(20,20)\}$ (80), ϕ_4: $[21,28]\times[0,8]$ (72)

```
⟨ Private regions 21 ⟩ ≡
  typedef struct {
    int x0, x1, y0, y1, ex, ey, expect;
  } priv_def;
  static const priv_def PRIV_2V2_DIAG[4] = {{0, 8, 0, 8, 8, 8, 80}, {0, 7, 21, 28, −1, −1, 64}, {20, 28, 20, 28,
      20, 20, 80}, {21, 28, 0, 7, −1, −1, 64}, };
  static const priv_def PRIV_2V2_ADJ[4] = {{0, 8, 0, 7, −1, −1, 72}, {21, 28, 20, 28, −1, −1, 72}, {0, 8, 21,
      28, −1, −1, 72}, {21, 28, 0, 8, −1, −1, 72}, };
  static const priv_def PRIV_1V3[3] = {{0, 7, 20, 28, −1, −1, 72}, {21, 28, 21, 28, −1, −1, 64}, {20, 28, 0, 7,
      −1, −1, 72}, };
  static const priv_def PRIV_3V1[3] = {{0, 8, 21, 28, −1, −1, 72}, {20, 28, 20, 28, 20, 20, 80}, {21, 28, 0, 8,
      −1, −1, 72}, };
```

See also section 22.

This code is used in section 18.

22. **#define** *max_priv* 100

```
⟨ Private regions 21 ⟩ +≡
  int pv_x[4][max_priv], pv_y[4][max_priv], pv_n[4];
  void set_priv(int k, const priv_def *p)
  {
    int x, y; pv_n[k] = 0;
    for (x = p→x0; x ≤ p→x1; x++)
      for (y = p→y0; y ≤ p→y1; y++)
        if (x ≠ p→ex ∨ y ≠ p→ey) {
          if (pv_n[k] ≥ max_priv) {
            printf("max_priv␣overflow!\n");
            exit(1);
          }
          pv_x[k][pv_n[k]] = x; pv_y[k][pv_n[k]] = y; pv_n[k]++;
        }
    if (pv_n[k] ≠ p→expect) {
      printf("priv_def␣sanity␣check␣failed:␣slot␣%d␣got␣%d,␣expected␣%d\n", k, pv_n[k], p→expect);
      exit(1);
    }
  }
```

23. Transform enumeration with corner-mask precomputation. The corner mask records which of the target's four corners are reachable; two transforms can jointly tile the target only if their masks OR to #F.

```
#define max_t 1700
    /* Without private-region filtering, each slot could have up to 8 × (w + S − 1)^2 transforms
       (8 D8 indices times all valid shifts), or 8 × 49^2 ≈ 19,000 for S = v = 21. The private-region
       constraint drastically reduces this: the largest observed count is ∼1,500 (slot 0 in the adjacent
       2+2 case), so 1700 suffices with margin. An overflow is caught by a runtime check below. */

⟨ Transform enumeration 23 ⟩ ≡
  transform_t tl[4][max_t];
  unsigned char tc[4][max_t];    /* corner mask */
  int tn[4];
  unsigned char corner_mask(transform_t t)
  {   /* Which target corners are covered by this transform's source? We scan source pixels and check if
         any maps to a target corner. An O(1) inverse-transform approach is possible but unnecessary: this
         runs only during initialization, not in the inner loop. */
    int S = t.is_u ? uu : vv;
    int cx[4] = {0, 0, S − 1, S − 1}, cy[4] = {0, S − 1, S − 1, 0};
    unsigned char mask = 0;
    int x, y, ox, oy, c;
    for (x = 0; x < ww ∧ mask ≠ #F; x++)
      for (y = 0; y < ww ∧ mask ≠ #F; y++) {
        apply_t(t, x, y, &ox, &oy);
        for (c = 0; c < 4; c++)
          if (ox ≡ cx[c] ∧ oy ≡ cy[c]) mask |= (1 ≪ c);
      }
    return mask;
  }
  void enumerate(int k, int is_u, int fix)
  {   /* fix ≥ 0: only enumerate D8 index fix (symmetry reduction); fix < 0: enumerate all 8 D8 indices.
         Translation bounds: a source pixel has D8-image in [0, w−1], and after shifting by dx it must land
         in [0, S−1], so dx ranges from −(w−1) to S−1; similarly for dy. */
    int idx, dx, dy, i, ox, oy, S = is_u ? uu : vv;
    int lo = fix ≥ 0 ? fix : 0, hi = fix ≥ 0 ? fix : 7;
    tn[k] = 0;
    for (idx = lo; idx ≤ hi; idx++)
      for (dx = −(ww − 1); dx ≤ S − 1; dx++)
        for (dy = −(ww − 1); dy ≤ S − 1; dy++) {
          transform_t t = make_t(idx, is_u, dx, dy);
          int ok = 1;
          for (i = 0; i < pv_n[k] ∧ ok; i++) {
            apply_t(t, pv_x[k][i], pv_y[k][i], &ox, &oy);
            if (ox < 0 ∨ ox ≥ S ∨ oy < 0 ∨ oy ≥ S) ok = 0;
          }
          if (ok) {
            if (tn[k] ≥ max_t) {
              printf("max_t␣overflow!\n");
              exit(1);
            }
            tl[k][tn[k]] = t; tc[k][tn[k]] = corner_mask(t); tn[k]++;
          }
```

```
    }
}
```

This code is used in section 18.

24. Filters: private-target disjointness check and precomputed pairwise compatibility matrix.

The array *ptbuf* marks which target pixels are "touched" by mapping the private source cells of one transform; if a second transform's private cells touch any already-marked pixel, the two transforms conflict (their forced target regions overlap).

⟨ Filters 24 ⟩ ≡

```
char ptbuf[vv][vv];     /* sized for the larger target (v > u); safe for both */
int ptd(transform_t a, int ka, transform_t b, int kb)
{
  int S = a.is_u ? uu : vv;
  int i, ox, oy;
  memset(ptbuf, 0, sizeof (ptbuf));
  for (i = 0; i < pv_n[ka]; i++) {
    apply_t(a, pv_x[ka][i], pv_y[ka][i], &ox, &oy);
    if (ox ≥ 0 ∧ ox < S ∧ oy ≥ 0 ∧ oy < S) ptbuf[ox][oy] = 1;
  }
  for (i = 0; i < pv_n[kb]; i++) {
    apply_t(b, pv_x[kb][i], pv_y[kb][i], &ox, &oy);
    if (ox ≥ 0 ∧ ox < S ∧ oy ≥ 0 ∧ oy < S)
      if (ptbuf[ox][oy]) return 0;
  }
  return 1;
}
```

See also sections 25 and 26.

This code is used in section 18.

25. Pairwise compatibility matrices for the 1+3/3+1 splits. *compat_ab*[*i*][*j*] is 1 iff slots *a* and *b* (with transforms *tl*[*a*][*i*] and *tl*[*b*][*j*]) have disjoint private-target images. The subscripts are *slot numbers*: *compat12* stores slot 1 vs. slot 2, *compat31* stores slot 3 vs. slot 1, and *compat32* stores slot 3 vs. slot 2. The digit order matches the index order: *compat31*[*i3*][*i1*] means the first index is a slot-3 transform and the second is a slot-1 transform.

⟨ Filters 24 ⟩ +≡

```
char compat12[max_t][max_t], compat31[max_t][max_t], compat32[max_t][max_t];
void precompute_compat(int a, int b, char dst[][max_t])
{
  int i, j;
  for (i = 0; i < tn[a]; i++)
    for (j = 0; j < tn[b]; j++) dst[i][j] = ptd(tl[a][i], a, tl[b][j], b);
}
```

26. Pair storage for 2+2 (slots 0,2 share V; slots 1,3 share U). Unlike the 1+3/3+1 case, we compute *ptd* on the fly rather than precomputing a full matrix, since *build_pairs* also applies the corner-coverage filter and stores only surviving pairs.

```
#define max_pair 100000   /* After private-target disjointness and corner-coverage filtering, the largest
        observed pair count is ~70,000 (the U-pair in the adjacent 2+2 case). The bound 100,000
        provides comfortable headroom; an overflow is caught at runtime. */

⟨ Filters 24 ⟩ +≡
  int pa[max_pair], pb[max_pair], pn;
  int qa[max_pair], qb[max_pair], qn;
  void build_pairs(int a, int b, int *da, int *db, int *dn)
  {
    int i, j; *dn = 0;
    for (i = 0; i < tn[a]; i++)
      for (j = 0; j < tn[b]; j++) {
        if (¬ptd(tl[a][i], a, tl[b][j], b)) continue;
        if ((tc[a][i] | tc[b][j]) ≠ #F) continue;
        if (*dn ≥ max_pair) {
          printf("max_pair␣overflow!\n");
          exit(1);
        }
        da[*dn] = i; db[*dn] = j; (*dn)++;
      }
  }
```

27. 2+2 split. By the $D_8(W)$ symmetry (Section 5), any 4-tuple has an equivalent representative where slot 0 has $idx = 0$; we enumerate only this representative, reducing the search by 8×. Similarly, by $D_8(U)$ symmetry we fix slot 1 to $idx = 0$, for another 8×. Together these give a 64× reduction.

⟨ 2+2 split 27 ⟩ ≡

```
long long run_2v2(int diag)
{
  long long surv = 0, tested = 0;
  int i, j;
  const priv_def *P = diag ? PRIV_2V2_DIAG : PRIV_2V2_ADJ;
  printf("␣␣2+2␣%s:␣", diag ? "diag" : "adj");
  for (i = 0; i < 4; i++) set_priv(i, &P[i]);
  enumerate(0, 0, 0);    /* slot 0: V-piece, fix idx = 0 by D8(W) */
  enumerate(1, 1, 0);    /* slot 1: U-piece, fix idx = 0 by D8(U) */
  enumerate(2, 0, −1);    /* slot 2: V-piece, all 8 indices */
  enumerate(3, 1, −1);    /* slot 3: U-piece, all 8 indices */
  printf("transforms␣%d,%d,%d,%d;␣", tn[0], tn[1], tn[2], tn[3]);
  build_pairs(0, 2, pa, pb, &pn);
  build_pairs(1, 3, qa, qb, &qn);
  printf("pairs␣%d,%d;␣", pn, qn);
  for (i = 0; i < pn; i++)
    for (j = 0; j < qn; j++) {
      tested++;
      if (bp_test(tl[0][pa[i]], tl[1][qa[j]], tl[2][pb[i]], tl[3][qb[j]])) surv++;
    }
  printf("tested␣%lld,␣survivors␣%lld\n", tested, surv);
  return surv;
}
```

This code is used in section 18.

28. 1+3 and 3+1 splits. The single piece must cover its entire target. By $D_8(W)$ symmetry we may assume it is the identity transform at corner $(0,0)$. By D_8 of the triple's target we fix slot 1 to $idx = 0$, giving an 8× reduction.

⟨ 1+3 and 3+1 split 28 ⟩ ≡

```
long long run_1v3_or_3v1(int is_u_single)
{
  long long surv = 0, tested = 0;
  int i1, i2, i3, iu = ¬is_u_single;
  const priv_def *P = is_u_single ? PRIV_1V3 : PRIV_3V1;
  printf("␣␣%s:␣", is_u_single ? "1+3" : "3+1");   /* Slot 0 (the lone piece) is fixed to the identity
      transform at corner (0,0) by D8(W); no enumeration or private region is needed—it is constructed
      directly via make_t(0, ... ,0,0). */
  set_priv(1, &P[0]); set_priv(2, &P[1]); set_priv(3, &P[2]);
  enumerate(1, iu, 0);   /* fix idx = 0 by D8 of the triple's target */
  enumerate(2, iu, −1);   /* all 8 indices */
  enumerate(3, iu, −1);   /* all 8 indices */
  printf("transforms␣%d,%d,%d;␣", tn[1], tn[2], tn[3]);
  precompute_compat(1, 2, compat12);
  precompute_compat(3, 1, compat31);
  precompute_compat(3, 2, compat32);
  for (i1 = 0; i1 < tn[1]; i1++)
    for (i2 = 0; i2 < tn[2]; i2++) {
      if (¬compat12[i1][i2]) continue;
      for (i3 = 0; i3 < tn[3]; i3++) {
        if (¬compat31[i3][i1]) continue;
        if (¬compat32[i3][i2]) continue;
        if ((tc[1][i1] | tc[2][i2] | tc[3][i3]) ≠ #F) continue;
        tested++;
        if (bp_test(make_t(0, is_u_single, 0, 0), tl[1][i1], tl[2][i2], tl[3][i3])) surv++;
      }
    }
  printf("tested␣%lld,␣survivors␣%lld\n", tested, surv);
  return surv;
}
```

This code is used in section 18.

29. Acknowledgment. Claude (Anthropic) served as a research assistant throughout this project: writing and iterating on the C verification code and this CWEB document, producing exploratory scripts to test algorithmic variants, and collecting experimental data across parameter choices. All mathematical reasoning, algorithm design decisions, and final correctness verification were performed by the author by hand.

30. Index.

_IONBF: 18.
a: 20, 24, 25, 26.
adj: 20.
apply_t: 19, 20, 23, 24.
b: 20, 24, 25, 26.
bp_add: 20.
bp_propagate: 20.
bp_test: 20, 27, 28.
bpq: 20.
build_pairs: 26, 27.
c: 20, 23.
compat_ab: 25.
compat12: 25, 28.
compat31: 25, 28.
compat32: 25, 28.
corner_mask: 23.
cx: 23.
cy: 23.
d: 20.
da: 26.
db: 26.
deg: 20.
diag: 27.
dn: 26.
dst: 25.
dx: 19, 23.
dy: 19, 23.
enumerate: 23, 27, 28.
ex: 21, 22.
exit: 20, 22, 23, 26.
expect: 21, 22.
ey: 21, 22.
fix: 23.
head: 20.
hi: 23.
i: 20, 23, 24, 25, 26, 27.
idx: 18, 19, 23, 27, 28.
is_u: 19, 20, 23, 24.
is_u_single: 28.
iu: 28.
i1: 25, 28.
i2: 28.
i3: 25, 28.
j: 20, 25, 26, 27.
k: 22, 23.
ka: 24.
kb: 24.
lo: 23.
main: 18.
make_t: 19, 23, 28.
mask: 23.
mate: 20.
max_pair: 26.
max_priv: 22.
max_t: 23, 25.
memset: 20, 24.
node: 20.
nodes: 19, 20.
off: 20.
ok: 23.
other: 20.
ox: 19, 20, 23, 24.
oy: 19, 20, 23, 24.
P: 27, 28.
p: 22.
pa: 26, 27.
pb: 26, 27.
pn: 26, 27.
precompute_compat: 25, 28.
printf: 18, 20, 22, 23, 26, 27, 28.
priv_def: 12, 21, 22, 27, 28.
PRIV_1V3: 21, 28.
PRIV_2V2_ADJ: 21, 27.
PRIV_2V2_DIAG: 15, 21, 27.
PRIV_3V1: 21, 28.
ptbuf: 24.
ptd: 24, 25, 26.
pv_n: 22, 23, 24.
pv_x: 22, 23, 24.
pv_y: 22, 23, 24.
qa: 26, 27.
qb: 26, 27.
qn: 26, 27.
run_1v3_or_3v1: 18, 28.
run_2v2: 18, 27.
S: 23, 24.
s: 20.
set_priv: 22, 27, 28.
setvbuf: 18.
sn: 20.
stdout: 18.
surv: 27, 28.
t: 19, 20, 23.
tail: 20.
tc: 23, 26, 28.
tested: 27, 28.
tl: 23, 25, 26, 27, 28.
tn: 20, 23, 25, 26, 27, 28.
total: 18.
transform_t: 18, 19, 20, 23, 24.
uu: 19, 20, 23, 24.
vv: 19, 20, 23, 24.

ww: 18, 19, 20, 23.
x: 19, 20, 22, 23.
x0: 21, 22.
x1: 21, 22.
y: 19, 20, 22, 23.
y0: 21, 22.
y1: 21, 22.

⟨ 1+3 and 3+1 split 28 ⟩ Used in section 18.
⟨ 2+2 split 27 ⟩ Used in section 18.
⟨ Bipartite graph 20 ⟩ Used in section 18.
⟨ Filters 24, 25, 26 ⟩ Used in section 18.
⟨ Private regions 21, 22 ⟩ Used in section 18.
⟨ Transform enumeration 23 ⟩ Used in section 18.
⟨ Transform type 19 ⟩ Used in section 18.

PROOF202129

	Section	Page
Introduction	1	1
Discrete dissections	2	2
Pythagorean dissections	3	4
Structural properties of transforms	4	5
Symmetry groups	5	6
The framework	8	8
The $(20, 21, 29)$ case — verification program	9	9
Index	30	24