Optimizing the 2D Shortest Superstring: Complexity, Combinatorial Reduction, and Metaheuristics

Dat Thanh Tran[a], Khai Quang Tran[a] and Van Khu Vu[a,*]

[a] *VinUniversity, Hanoi, Vietnam*

ARTICLE INFO

Keywords:
Shortest superstring problem
Two-dimensional strings
Genetic algorithm
Integer linear programming
Combinatorial optimization
Metaheuristics

ABSTRACT

The Shortest Superstring Problem (SSP)—finding the shortest string containing a given set of strings as substrings—is a cornerstone of combinatorial optimization with deep connections to genome assembly, data compression, and scheduling. We introduce and systematically investigate its two-dimensional generalization (2D-SSP): given a collection of rectangular symbol arrays, arrange them on the integer lattice with symbol-consistent overlaps so as to minimize the bounding-box area. This generalization is far from straightforward: while 1D-SSP is a pure sequencing problem, 2D-SSP becomes a *simultaneous sequencing-and-packing* problem where strings overlap from four directions and may fill internal holes, fundamentally altering the optimization landscape.

We establish the computational complexity of 2D-SSP, proving NP-hardness for both the area and balanced-area objectives, and APX-hardness for the area objective via an L-reduction from 1D-SSP that preserves approximation ratios exactly. More significantly, we develop a *Discrete Combinatorial Reduction* that transforms this ostensibly geometric problem—optimization over an infinite Cartesian coordinate space—into a finite combinatorial problem over spanning trees of a placement graph. The key insight is the *Connectivity Principle*: every optimal placement can be made 4-connected without increasing cost. This guarantees that at least one optimal solution admits a spanning-tree representation, so searching over trees loses no optima. The correspondence is not a bijection—a single placement may be represented by multiple trees (via different root choices or spanning-tree selections from a cyclic contact graph), and not every tree decodes to a feasible placement—but the tree space provably *covers* all connected placements. This reduction collapses the infinite coordinate space into a finite (though exponential) search space amenable to both exact and metaheuristic methods.

Exploiting this tree-based representation, we design a Tree-Based Genetic Algorithm (T-GA) featuring a *locality-preserving crossover* operator grounded in the building-block hypothesis: by recombining subtrees rather than raw coordinates, the crossover preserves functional clusters of tightly overlapping strings across generations. Computational experiments demonstrate that T-GA matches ILP-optimal solutions on small instances and consistently outperforms greedy heuristics by 6–12% on larger instances, bridging the gap between fast but suboptimal heuristics and optimal but non-scalable exact methods. Notably, our theoretical framework relies only on translation and adjacency, making it directly applicable to d-dimensional generalizations for any $d \geq 1$.

1. Introduction

The Shortest Superstring Problem (SSP) is a classical NP-hard problem (Gallant, Maier and Storer, 1980): given a collection of strings, the goal is to construct the shortest string that contains each input string as a contiguous substring. SSP has been extensively studied, with known approximation algorithms (Blum, Jiang, Li, Tromp and Yannakakis, 1994; Mucha, 2013) and rich connections to combinatorial optimization and data compression.

In many modern applications, however, the objects of interest are not one-dimensional strings but two-dimensional patterns: small images, symbol arrays, or 2D strings. Examples include patterned fabrication, 2D barcodes, structured-light patterns, and 2D covering sequences. Recent work has begun to explore 2D analogues of covering and de Bruijn-type sequences, introducing the notion of covering 2D-sequences whose windows cover all patterns of a given size up to small Hamming radius (Yehezkeally and Schwartz, 2025).

In this work we study a two-dimensional generalization of the Shortest Superstring Problem (2D-SSP), where the basic objects are rectangular 2D strings $T_1, \dots, T_n$ (finite 2D arrays over a finite alphabet), collected into a set $\mathcal{T}$. The

*Corresponding author
dat.tt3@vinuni.edu.vn (D.T. Tran); khai.tq@vinuni.edu.vn (K.Q. Tran); khu.vv@vinuni.edu.vn (V.K. Vu)
ORCID(s):

goal is to place them in the plane with overlaps so that all 2D strings are embedded consistently while minimizing a bounding-box cost derived from the minimal axis-aligned bounding rectangle enclosing the occupied region. We consider two cost variants:

- the *area objective*, which minimizes the rectangle area $H \cdot W$, the natural 2D generalization of string length in the classical SSP;
- a *balanced-area objective*, which minimizes the side length $\max\{H, W\}$ of the smallest enclosing square, useful in applications requiring bounded aspect ratio (e.g., display panels, chip layouts).

The transition from 1D to 2D introduces a fundamental *complexity leap*. In 1D-SSP, strings can only overlap from two directions (left/right), making it a pure sequencing problem. In 2D-SSP, strings can overlap from four directions, and—crucially—a new string can fill a "hole" created by the arrangement of other strings. This transforms the problem into a *simultaneous sequencing and packing* problem, where the optimal placement of one string depends not just on its neighbors but on the global geometric configuration. Our approach is the first to bridge these two domains by using a graph-based structure to handle the sequencing aspect and a grid-based canvas to handle the packing constraints.

Our central modelling innovation is a *topological encoding* that shifts the search space from absolute Cartesian coordinates to relative placement trees. Rather than representing a solution as an explicit 2D array or a vector of (x, y) coordinates, we encode it as a *tree of relative offsets between 2D strings*. This representation achieves *symmetry-breaking* by collapsing the infinitely many translationally equivalent coordinate vectors into a single canonical form, and enforces structural connectivity by construction. Unlike coordinate-based representations, which suffer from high redundancy due to translational invariance—a solution at $(0, 0)$ is equivalent to one at $(1, 1)$, $(1, 2)$, etc.—our topological encoding ensures that each individual represents a *unique relative arrangement*, significantly increasing metaheuristic search efficiency by eliminating redundant exploration of the coordinate space.

From a *theoretical* perspective, we develop a *Discrete Combinatorial Reduction* for 2D-SSP. We establish a *Connectivity Principle* (Lemma 3): the search for optimal solutions can be restricted to 4-connected placements without loss of optimality. While "sliding" arguments are foundational in VLSI floorplanning to achieve compacted placements (e.g., Sequence Pairs (Murata, Fujiyoshi, Nakatake and Kajitani, 1996a)), we provide a formal proof that this property extends to the regime of *symbol-consistent overlaps*—a constraint absent in traditional floorplanning. Building on this, we prove an *Optimality-Preserving Equivalence* (Theorem 6): every connected placement can be represented by a spanning tree of a naturally defined placement graph (though the correspondence is many-to-one: multiple trees may encode the same placement, and not every tree yields a feasible placement when decoded). Crucially, at least one optimal solution always admits such a tree representation, so the tree search space is *complete* for optimization. Together, these results establish that *the search space can be reduced from an infinite Cartesian plane to a finite discrete combinatorial space*—the space of spanning trees with bounded edge labels. While this space remains exponential in the worst case, it transforms a geometric optimization problem into a structured combinatorial one amenable to both exact and metaheuristic methods.

From an *algorithmic* perspective, the topological encoding enables a *locality-preserving crossover* operator that directly addresses the *building block hypothesis* (Goldberg, 1989), the principle that effective genetic operators should recombine functional substructures rather than destroying them. Consider a good partial solution where strings A, B, and C form a tightly packed cluster. In our tree representation, this cluster corresponds to a subtree encoding exactly how these strings fit together. A crossover operator can transplant this entire subtree from one parent to another, preserving the beneficial local arrangement and enabling *schema preservation* across generations. In contrast, a coordinate-based representation stores only absolute positions $(x_A, y_A), (x_B, y_B), (x_C, y_C)$; crossover that mixes coordinates from different parents destroys the cluster's internal structure, since the absolute positions are meaningful only relative to a specific global arrangement.

This observation confirms that the topological encoding enables effective schema preservation: tree-based crossover produces higher-quality offspring than coordinate-based approaches because it maintains the functional building blocks that make solutions effective. Our Tree-Based GA (T-GA) consistently outperforms greedy baselines and matches ILP-optimal solutions on small instances while scaling to much larger problems.

2. Background and Related Work

2.1. Shortest Superstring Problem

In the classical SSP, the input is a set of strings

$$S = \{s_1, \dots, s_n\}$$

over an alphabet Σ. A superstring is a string S in which each s_i appears as a substring. The objective is to minimize $|S|$. SSP is NP-hard, and there is a substantial literature on constant-factor approximations (e.g., greedy maximum-overlap merging, cycle-cover-based algorithms) and heuristic implementations used in practice.

2.2. 2D Covers and 2D Covering Sequences

Two-dimensional generalizations of string concepts appear in several areas:

- *2D covers and 2D strings.* Work on covers of 2D arrays considers how a small pattern can cover a larger 2D string with overlaps, generalizing the notion of a cover in 1D (Charalampopoulos, Pissis, Radoszewski, Waleń and Zuba, 2021).
- *Covering sequences and 2D covering sequences.* Recent research introduces covering sequences and covering 2D-sequences, where all $m \times n$ windows of a large 2D array form a covering code for patterns of that size up to a given radius. These provide natural sources of structured test instances (Yehezkeally and Schwartz, 2025).

These works focus on covering combinatorial spaces, whereas we focus on overlapping a *given finite set* of 2D strings with exact symbol consistency.

2.3. 2D Bin Packing and Cutting Stock

Classical two-dimensional bin packing problems ask how to place a collection of rectangles into one or more rectangular bins so as to minimize, for example, the number of bins used or the height of a single strip, under strict *non-overlap* constraints. The items are unlabeled shapes. A closely related problem is the *two-dimensional cutting stock problem* (Gilmore and Gomory, 1965; Lodi, Martello and Monaci, 2002), where the goal is to cut rectangular pieces from large stock sheets while minimizing waste. Both problems have been extensively studied using exact methods (branch-and-bound, column generation) and metaheuristics (genetic algorithms, simulated annealing) (Bennell and Oliveira, 2009).

Our setting is similar in that we also optimize a global bounding box for a family of rectangular pieces, but differs in two key ways. First, each string is a *discrete symbol array* rather than an unlabeled rectangle; second, *overlaps are allowed* as long as they are *symbol-consistent*. Thus a solution is not simply a packing of shapes, but a combinatorial "gluing" of patterns in which overlaps can reduce the effective occupied area, a phenomenon absent from standard 2D bin packing and cutting stock formulations. In the language of cutting stock, 2D-SSP allows "pieces" to share material when their patterns match—a constraint that transforms the problem from pure geometry to a hybrid of sequencing and packing.

2.4. Related Geometric and Assembly Problems

While 2D-SSP is distinct in its requirement for exact symbol consistency, it shares significant structural similarities with problems in VLSI design, computer graphics, and molecular computing.

In Very Large Scale Integration (VLSI) physical design, the floorplanning stage seeks to arrange rectangular modules without overlap to minimize total area and wirelength (Wong and Liu, 1986). This problem is a close cousin to 2D bin packing, but the methodological overlap with our work lies in the *representation* of solutions. Topological representations such as Sequence Pairs (Murata, Fujiyoshi, Nakatake and Kajitani, 1996b) and B*-trees (Chang, Chang, Wu and Wu, 2000) encode relative positions of rectangles to explore the search space efficiently. Our use of placement trees (Section 3) is conceptually related to these nonslicing floorplan representations, adapted to the regime where adjacency is defined by content overlap rather than physical interconnects.

In computer graphics, patch-based texture synthesis aims to generate large textures by stitching together small sample patches (Efros and Freeman, 2001; Kwatra, Schödl, Essa, Turk and Bobick, 2003). Algorithms in this domain typically place patches greedily or via graph cuts to minimize the visual error in the overlapping regions. 2D-SSP can be viewed as the discrete, lossless limit of these problems: instead of minimizing a pixel-difference norm (soft constraint), we require zero Hamming distance in the overlap (hard constraint) and optimize for maximum compression.

The constraints of 2D-SSP strongly resemble the algorithmic self-assembly of DNA tiles. In the Tile Assembly Model (TAM) introduced by Winfree, Liu, Wenzler and Seeman (1998), square tiles (Wang tiles) attach to a growing crystal lattice only if their edges match the specific "sticky ends" (symbols) of neighbors. While DNA self-assembly focuses on the forward kinetic process of crystallization or the computational power of the resulting tiling, 2D-SSP effectively asks for the most compact configuration (or "seed") that could theoretically be assembled from a given multiset of patterned tiles.

2D-SSP can also be viewed through the lens of computational jigsaw puzzle assembly, where the goal is to reconstruct an image from a set of pieces based on visual compatibility at boundaries (Sholomon, David and Netanyahu, 2013). In our setting, the "pieces" are 2D strings and the compatibility constraint is exact symbol matching rather than visual similarity. A key difference is that in classical jigsaw puzzles each piece has a unique location, whereas in 2D-SSP multiple valid placements may exist and the objective is to minimize the bounding box rather than reconstruct a known target. The problem also relates to polyomino packing (Golomb, 1994), which asks whether a given set of polyomino shapes can tile a region. However, polyomino packing typically assumes non-overlapping pieces with geometric interlocking, while 2D-SSP permits symbol-consistent overlaps that effectively "merge" the content of multiple strings, a feature that enables compression beyond what pure geometric packing allows.

3. Preliminaries

We now formalize the Two-Dimensional Shortest Superstring Problem (2D-SSP), introduce the two objective variants, and develop the *Discrete Combinatorial Reduction* that enables structured algorithmic treatment. The central contribution of this section is proving that the search space can be *losslessly reduced* from an infinite coordinate space to a finite (though exponential) space of spanning trees with bounded edge labels.

Let Σ be a finite alphabet over which the 2D strings are defined. A *2D-string* over Σ is a finite $m\times n$ array $T \in \Sigma^{m\times n}$, for some $m, n \in \mathbb{N}$. For indices $1 \le i \le j \le m$ and $1 \le i' \le j' \le n$, we write $T[i..j,\ i'..j']$ for the corresponding subarray and call this a *2D-substring* of T.

We identify a 2D string T with a function on a finite index set $C_T \subset \mathbb{Z}^2$, its set of *local cell coordinates*. We write cells as pairs $(u, v) \in \mathbb{Z}^2$ and use the same coordinate system for both local and global positions: a translation by an offset $p(i) = (x_i, y_i)$ sends a local cell (u, v) of T_i to the global cell $p(i) + (u, v) = (x_i + u,\ y_i + v)$. The choice of which axis is drawn horizontally or vertically is irrelevant for our arguments; we only rely on coordinate-wise addition in $\mathbb{Z}^2$.

Let P be an $m' \times n'$ 2D-string. We denote the set of its occurrences in T by

$$\mathrm{Occ}(P, T) = \{(i, j) : T[i..i + m' - 1,\ j..j + n' - 1] = P\}.$$

We will derive cost functions from the dimensions of the minimal axis-aligned bounding rectangle of a placement, and consider two variants: one that minimizes the area and one that minimizes the maximum side length (balanced-area objective).

Definition 1. Let $\mathcal{T} = \{T_1, \dots, T_n\}$ be a finite multiset of 2D strings over Σ, which we call *2D strings*. An $m \times n$ 2D-string S is a *2D-superstring* of $\mathcal{T}$ if each string T_i occurs as a 2D-substring of S, i.e.,

$$\mathrm{Occ}(T_i, S) \neq \emptyset \quad \text{for all } i \in \{1, \dots, n\}.$$

We denote by

$$|S|_{\text{area}} := m \cdot n \quad \text{and} \quad |S|_{\text{bal}} := \max\{m, n\}$$

the *area* and the *balanced side length* of S, respectively. The area measure is the natural 2D analogue of string length in 1D-SSP, while the balanced measure constrains the aspect ratio. Both are derived from the minimal axis-aligned rectangle containing S and penalize any empty cells inside that rectangle.

Thus $|S|_{\text{area}}$ is the primary objective, the natural 2D analogue of superstring length (total cells in the bounding box), while $|S|_{\text{bal}}$ is a constrained variant that bounds the aspect ratio, ensuring neither dimension dominates.

Remark 1 (Wildcard characters and 2D holes)**.** The wildcard symbol “$*$” in Figure 1 represents a fundamental distinction between 1D-SSP and 2D-SSP. In 1D-SSP, every optimal superstring is *fully covered*: each position belongs

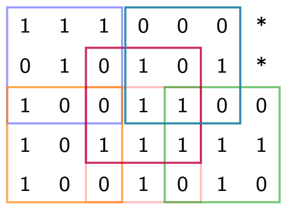

Figure 1: A 2D-superstring S (5 rows × 7 columns) containing six 2D strings $T_1, \dots, T_6$ as 2D-substrings (highlighted rectangles). The cells marked with "$*$" are *wildcard* (or *don't-care*) positions—cells within the bounding box that are not covered by any input string. The area is $|S|_{\text{area}} = 35$, and the balanced side length is $|S|_{\text{bal}} = 7$.

to at least one input string. If any position were uncovered, we could delete that character to obtain a shorter superstring—contradicting optimality.

In 2D-SSP, this property fails. The two-dimensional geometry permits *holes*: cells within the bounding box that no input string covers. These holes cannot simply be "deleted" as in 1D, because removing a row or column would disrupt the geometric arrangement of strings in other parts of the superstring. The wildcard cells contribute to the bounding-box cost but carry no information—they can be filled with any symbol (or left undefined) without affecting which strings are embedded.

Formally, given a placement p, let $R(p) = \bigcup_{i=1}^{n} \text{footprint}(T_i, p)$ denote the set of occupied cells, and let $B(p)$ denote the bounding box. The *hole set* is $B(p) \setminus R(p)$—cells inside the bounding box but not covered by any string. In Figure 1, this set contains two cells (the "$*$" positions). These holes represent "wasted" area that inflates the objective value, and minimizing them is part of the optimization challenge unique to 2D-SSP.

Definition 2. Given a finite multiset $\mathcal{T}$ of 2D strings over Σ, we define two variants of the Two-Dimensional Shortest Superstring Problem:

- *Area-based 2D-SSP* (2D-SSP$_{\text{area}}$): find a 2D-superstring S of $\mathcal{T}$ minimizing $|S|_{\text{area}}$. This is the primary variant, directly generalizing the 1D-SSP objective.
- *Balanced-area 2D-SSP* (2D-SSP$_{\text{bal}}$): find a 2D-superstring S of $\mathcal{T}$ minimizing $|S|_{\text{bal}}$. This variant constrains the aspect ratio and is useful in applications requiring near-square layouts.

Both objectives depend only on the minimal axis-aligned bounding rectangle of S and penalize all empty cells inside it.

3.1. Computational complexity

We establish that 2D-SSP inherits the computational hardness of its 1D counterpart.

Theorem 1 (NP-hardness of 2D-SSP)**.** *Both 2D-SSP$_{\text{area}}$ and 2D-SSP$_{\text{bal}}$ are NP-hard.*

Proof. We reduce from 1D-SSP, which is known to be NP-hard (Gallant et al., 1980). Given an instance $\mathcal{S} = \{s_1, \dots, s_n\}$ of 1D-SSP over alphabet Σ, construct a 2D-SSP instance $\mathcal{T} = \{T_1, \dots, T_n\}$ where each T_i is a $1 \times |s_i|$ array (a single row) containing the characters of s_i.

We claim that the optimal 1D superstring length equals the optimal 2D-SSP$_{\text{area}}$ cost for this instance.

(1D → 2D): Let S^* be an optimal 1D superstring of $\mathcal{S}$ with length $L^* = |S^*|$. Interpret S^* as a $1 \times L^*$ 2D array. Since each s_i occurs as a substring of S^*, the corresponding T_i occurs as a 2D-substring at the same position. Thus S^* is a 2D-superstring of $\mathcal{T}$ with $|S^*|_{\text{area}} = 1 \cdot L^* = L^*$.

(2D → 1D): Let p^* be an optimal placement for the 2D instance with bounding box $H \times W$. We show that a single-row arrangement ($H = 1$) is always optimal for the area objective. Suppose we partition strings into $k \geq 2$ rows, with each row i having optimal 1D superstring width W_i. The area is $k \cdot \max_i W_i$. Now, consider merging all strings into a single row: the optimal width is at most $\sum_i W_i$ (placing rows end-to-end), but typically much smaller due to inter-row overlaps. Crucially, $\max_i W_i \geq L^*/k$ where L^* is the single-row optimal, because no partition can compress better than the merged optimum. Thus $k \cdot \max_i W_i \geq k \cdot (L^*/k) = L^*$, with equality only when rows are perfectly balanced—which rarely occurs and never beats the single-row optimum. Therefore, $H = 1$ is optimal, and reading off the single row gives a valid 1D superstring of length $W = L^*$.

Thus the optimal 2D-SSP$_{\text{area}}$ cost equals the optimal 1D-SSP length, and any polynomial-time algorithm for 2D-SSP$_{\text{area}}$ would solve 1D-SSP.

Remark on 2D-SSP$_{\text{bal}}$*:* The above argument does *not* extend to the balanced-area objective. For 2D-SSP$_{\text{bal}}$, the cost is $\max\{H, W\}$, and using multiple rows can reduce W faster than it increases H. For example, if $L^* = 100$ and strings can be partitioned into 10 rows each of width 10, the balanced cost becomes $\max\{10, 10\} = 10 < 100$. Thus, the optimal 2D-SSP$_{\text{bal}}$ solution may use multiple rows even for $1 \times L$ strings, breaking the reduction from 1D-SSP. NP-hardness of 2D-SSP$_{\text{bal}}$ still follows (any 2D-SSP$_{\text{area}}$ instance is also a 2D-SSP$_{\text{bal}}$ instance), but APX-hardness requires a separate argument—see Remark 2. □

Remark 2 (APX-hardness via L-reduction)**.** The 1D-SSP is APX-hard: no polynomial-time approximation scheme (PTAS) exists unless P = NP (Blum et al., 1994). We establish that our reduction is, in fact, an *L-reduction* (linear reduction) (Papadimitriou and Yannakakis, 1991), which provides a stronger transfer of inapproximability.

Recall that an L-reduction from problem A to problem B requires two constants $\alpha, \beta > 0$ such that for every instance I_A of A with optimal value OPT_A, the corresponding instance I_B satisfies:

1. $\text{OPT}_B \le \alpha \cdot \text{OPT}_A$, and
2. for any solution to I_B with cost c_B, one can construct a solution to I_A with cost c_A satisfying $|c_A - \text{OPT}_A| \le \beta \cdot |c_B - \text{OPT}_B|$.

Our reduction achieves the strongest possible parameters: $\alpha = \beta = 1$. For $1 \times L$ strings (height 1, width L), the 1D-SSP cost (superstring length) maps *exactly* to the 2D-SSP$_{\text{area}}$ cost (bounding box area $1 \times W = W$). Specifically:

- $\text{OPT}_{\text{2D}} = \text{OPT}_{\text{1D}}$ (the optimal values are identical), so condition (1) holds with $\alpha = 1$.
- Any 2D placement with cost W directly yields a 1D superstring of length W (by reading the single row), with *zero* loss in the approximation gap: $|c_{\text{1D}} - \text{OPT}_{\text{1D}}| = |c_{\text{2D}} - \text{OPT}_{\text{2D}}|$. Thus condition (2) holds with $\beta = 1$.

This $(\alpha, \beta) = (1, 1)$ L-reduction implies that approximation ratios transfer perfectly: a ρ-approximation for 2D-SSP$_{\text{area}}$ would yield a ρ-approximation for 1D-SSP. Since 1D-SSP admits no PTAS, neither does 2D-SSP$_{\text{area}}$.

Status of 2D-SSP$_{\text{bal}}$*:* The L-reduction does not apply to 2D-SSP$_{\text{bal}}$ because, as noted above, the optimal 2D solution may use multiple rows even for $1 \times L$ strings, breaking the cost equivalence. Nevertheless, 2D-SSP$_{\text{bal}}$ is NP-hard via a different argument: when the alphabet is large enough that each string contains a unique symbol, no two strings can overlap, and 2D-SSP$_{\text{bal}}$ reduces to *rectangle packing with square-minimization objective*—packing axis-aligned rectangles into the smallest enclosing square. This problem is NP-hard (Leung, Tam, Wong, Young and Chin, 1990). Thus, NP-hardness of 2D-SSP$_{\text{bal}}$ follows from the packing special case, while APX-hardness remains an open question requiring a separate reduction.

This L-reduction provides theoretical justification for pursuing constant-factor approximations or metaheuristic approaches for 2D-SSP$_{\text{area}}$, rather than arbitrarily good polynomial-time approximation schemes.

Remark 3 (Alphabet size and the spectrum of hardness)**.** The role of alphabet size $|\Sigma|$ merits clarification. Our reduction from 1D-SSP preserves the alphabet, so hardness holds for any $|\Sigma| \ge 2$ (since 1D-SSP is NP-hard even for binary alphabets (Gallant et al., 1980)).

The case $|\Sigma| = 1$ is fundamentally different: with a unary alphabet, symbol-consistency is trivially satisfied (all symbols match), and all strings can be stacked at the origin. The optimal bounding box is simply the smallest rectangle containing any single string—the problem becomes trivial.

The opposite extreme is equally instructive: when $|\Sigma| \ge n$ and each string T_i contains at least one cell with a unique symbol σ_i not appearing in any other string, then *no two strings can overlap*. Symbol-consistency forces all strings to be placed disjointly, and 2D-SSP reduces to a pure *rectangle packing problem*—minimizing the bounding box of non-overlapping axis-aligned rectangles. This relates to classical VLSI floorplanning problems such as *Area Minimization*, which are NP-hard (**?**).

Thus, 2D-SSP hardness arises from the *interplay* between overlap opportunities and overlap constraints:

- *Packing regime* (high entropy, $|\Sigma|$ large): When strings are sufficiently distinct that few pairs can overlap, the problem approaches NP-hard rectangle packing. The geometric arrangement dominates.
- *Sequencing regime* (low entropy, $|\Sigma|$ small): When many pairs admit overlaps, finding the optimal overlap structure inherits the combinatorial hardness of 1D-SSP. Overlaps become a resource (enabling compression) constrained by symbol compatibility.

The intermediate cases, where some but not all pairs can overlap, combine both sources of difficulty. Our algorithms must navigate this spectrum.

3.1.1. Complexity sensitivity and problem positioning

Having established the basic hardness results, we now discuss how the problem's complexity varies with instance characteristics, positioning 2D-SSP within the landscape of related combinatorial problems.

SSP vs. Packing: Opposing objectives. It is important to distinguish 2D-SSP from classical *2D bin packing* or *pallet loading* problems (Morabito and Morales, 1998; Korf, 2003). In packing problems, the goal is to fit rectangles into a container *without overlap*; overlap is forbidden. In 2D-SSP, overlaps are not merely permitted but *encouraged*—they reduce the bounding box by allowing strings to share cells. The objectives are thus diametrically opposed:

- *Packing*: Maximize density subject to *no overlap*.
- *SSP*: Maximize overlap (compression) subject to *symbol-consistency*.

For $|\Sigma| = 1$, where symbol-consistency is trivially satisfied, 2D-SSP permits *maximal overlap*—strings can stack arbitrarily—making the problem easier (in fact, trivial: stack all strings at the origin, yielding bounding box equal to the largest string). In contrast, packing with $|\Sigma| = 1$ (i.e., unlabeled rectangles) remains NP-hard because the no-overlap constraint persists. This reversal highlights that 2D-SSP and packing, despite superficial similarity, have fundamentally different structure.

Information entropy and the sequencing–packing spectrum. The effective difficulty of a 2D-SSP instance depends on the *information entropy* of its strings. Consider two extremes:

- *High entropy (random strings, large $|\Sigma|$):* When strings are drawn uniformly at random over a large alphabet, the probability that two strings have a symbol-consistent overlap decreases exponentially with overlap size. Most pairs of strings can only be placed adjacently (no overlap), and 2D-SSP degenerates toward a *packing* problem—the geometric arrangement dominates, and the symbol structure provides little compression opportunity.
- *Low entropy (repetitive strings, small $|\Sigma|$):* When strings are highly repetitive or drawn from a small alphabet, many pairs admit large overlaps. The problem becomes a *sequencing* problem—finding the optimal order and alignment to exploit overlaps, akin to classical 1D-SSP. The combinatorial explosion of valid overlap configurations dominates the difficulty.

Thus, $|\Sigma|$ and string structure jointly determine where an instance lies on the *sequencing–packing spectrum*. Low-entropy instances stress the sequencing core (inherited from 1D-SSP), while high-entropy instances stress the packing core (inherited from rectangle packing).

Experimental scope. Our experimental evaluation (Section 5) focuses primarily on the *sequencing core* of 2D-SSP: we use binary alphabets ($|\Sigma| = 2$) and generate instances adapted from 1D-SSP benchmarks, where strings exhibit sufficient repetition to enable meaningful overlaps. This design choice allows direct comparison with 1D-SSP literature and isolates the novel 2D aspects (vertical overlaps, aspect ratio constraints) from the confounding effects of high-entropy packing regimes. Exploring the full sequencing–packing spectrum, including high-entropy instances where packing dominates, remains an interesting direction for future work.

Whenever the distinction between the two variants is not important, we simply refer to either as *2D-SSP*. Hereafter, we use *string* to mean *2D string* unless otherwise specified.

We assume throughout that strings are axis-aligned rectangles and cannot be rotated or reflected; repeated strings in $\mathcal{T}$ are treated as distinct objects that must each be embedded at least once.

Rather than working directly with the superstring S, we use placements on the integer grid.

Definition 3. A *placement* of $\mathcal{T}$ is a function

$$p : \{1, \dots, n\} \to \mathbb{Z}^2, \qquad p(i) = (x_i, y_i),$$

assigning an integer offset to each string T_i. A cell of T_i with local coordinates (u, v) (row and column indices) is mapped to global coordinates $(x_i + u,\ y_i + v) \in \mathbb{Z}^2$.

A placement p is symbol-consistent if for every global coordinate $(x, y) \in \mathbb{Z}^2$, all strings covering (x, y) under p write the same symbol. We denote by

$$R(p) := \{(x, y) \in \mathbb{Z}^2 : (x, y) \text{ is covered by some } T_i \text{ under } p\}$$

the union of occupied global cells, and let $B(p)$ be the minimal axis-aligned rectangle containing $R(p)$. Let $W(p)$ and $H(p)$ be the width and height of $B(p)$, and define

$$\text{cost}_{\text{area}}(p) := W(p) \cdot H(p), \qquad \text{cost}_{\text{bal}}(p) := \max\{W(p), H(p)\}.$$

Restricting the symbol map to $B(p)$ yields an $m \times n$ array T_p; by construction T_p is a 2D-superstring of $\mathcal{T}$, and its area and balanced side satisfy

$$|T_p|_{\text{area}} = \text{cost}_{\text{area}}(p), \qquad |T_p|_{\text{bal}} = \text{cost}_{\text{bal}}(p).$$

In particular, empty cells of $B(p)$ that are not covered by any strings still contribute to both objectives. Thus every symbol-consistent placement defines a feasible solution to both 2D-SSP$_{\text{area}}$ and 2D-SSP$_{\text{bal}}$, with cost equal to the chosen bounding-box functional.

Remark 4 (Bounding box vs. union area)**.** One might consider an alternative objective $|R(p)|$—the number of occupied cells—rather than the bounding box area $W(p) \cdot H(p)$. We adopt the bounding box for three reasons:

(i) Theoretical coherence. In 1D-SSP, the objective is superstring *length*, which equals the bounding interval of the placement. There is no distinction between “bounding box” and “union” in one dimension. The bounding box area $H \cdot W$ is thus the natural 2D generalization of 1D length.

(ii) Application requirements. In the motivating application of 2D barcode fabrication (e.g., DataMatrix or QR codes), the output is printed on a rectangular substrate. Empty cells inside the bounding box consume substrate area regardless of whether they encode data; they represent “wasted” space that increases manufacturing cost. Minimizing $|R(p)|$ instead would permit “Swiss cheese” solutions with scattered voids that still require the same physical substrate.

(iii) Structural constraints, not structural compactness. The Connectivity Principle (Lemma 3) guarantees that optimal placements can be made 4-connected, eliminating *disconnected* components floating in separate regions. However, 4-connectivity does not imply convexity or the absence of internal voids: a U-shaped region, or a square with a large central hole connected to the exterior by a thin channel, is 4-connected yet has significant empty space within its bounding box. Thus, the bounding box area $W \cdot H$ can exceed the union area $|R(p)|$ even for connected placements.

We adopt the bounding box objective not because it approximates $|R(p)|$, but because it is the *relevant* metric for our target applications: substrate-based fabrication where the physical medium is rectangular. Minimizing $|R(p)|$ would be appropriate for applications where only occupied cells incur cost (e.g., ink consumption), but would permit fragmented layouts that waste substrate area. In practice, our experimental instances with small, regular strings tend to produce compact placements where $W \cdot H$ and $|R(p)|$ are close, but this is an empirical observation about typical instances, not a theoretical guarantee for arbitrary inputs.

Remark 5 (Why 2D optimality requires more than row/column coverage)**.** In 1D-SSP, a classical observation simplifies the analysis: every position in an optimal superstring S^* must belong to at least one occurrence of some input string $s_i \in \mathcal{S}$. If any position were “uncovered,” we could remove that character to obtain a shorter superstring—a contradiction. This ensures the optimal superstring has no wasted characters.

One might hope for an analogous property in 2D: “every row and every column of the bounding box contains at least one cell from some input string.” While this *necessary* condition is easy to verify, it is *not sufficient* to guarantee optimality. Figure 2 illustrates a placement where every row and column intersects some string, yet the strings can be translated to reduce the bounding box dimensions.

The fundamental difference is that 2D placement admits *diagonal slack*: strings may be spread apart in a way that covers all rows and columns but leaves room for compaction along diagonal directions. The Connectivity Principle (Lemma 3) addresses this by showing that optimal placements can always be made 4-connected, which eliminates such diagonal gaps. This structural result is the 2D analogue of the 1D “no uncovered position” property, but requires a substantially more sophisticated proof.

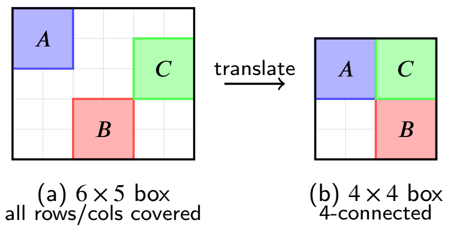

Figure 2: Row/column coverage is necessary but not sufficient for 2D optimality. (a) Both strings cover all rows and columns of the 6×5 bounding box, yet they are disconnected. (b) Translating B diagonally yields a 4-connected placement with a smaller 4 × 4 bounding box.

Conversely, let S be a 2D-superstring of $\mathcal{T}$ and fix an arbitrary occurrence $(i, j) \in \mathrm{Occ}(T_k, S)$ for each string T_k. Placing T_k with offset (i, j) then yields a symbol-consistent placement whose induced array is S up to a global translation. Hence optimizing over 2D-superstrings is equivalent to optimizing over symbol-consistent placements, modulo a global shift of all coordinates. We therefore work with placements from now on.

A subset $R \subseteq \mathbb{Z}^2$ is *4-connected* if its adjacency graph under the 4-neighbourhood ($\|x - y\|_1 = 1$) is connected. Two cells $x, y \in \mathbb{Z}^2$ overlap if $x = y$.

Definition 4. Given the string set $\mathcal{T}$, the *placement graph* G^{pl} has vertex set $\{1, \dots, n\}$. Its edges are triples

$$e = (i, j, \delta) \quad \text{with} \quad i \neq j,\ \delta \in \mathbb{Z}^2,$$

where δ is a relative offset such that placing T_j at position $p(j) = p(i) + \delta$ yields symbol-consistent contact between T_i and T_j, meaning they either overlap with matching symbols, or are 4-adjacent (share a boundary edge). Multiple edges may exist for the same unordered pair $\{i, j\}$, corresponding to different valid offsets.

Intuitively, each edge (i, j, δ) in G^{pl} specifies a way of gluing T_j next to T_i. The placement graph is determined entirely by $\mathcal{T}$ and can be precomputed before searching for solutions.

Lemma 2. *Let T_i and T_j have bounding boxes of dimensions $w_i \times h_i$ and $w_j \times h_j$. If (i, j, δ) is an edge in G^{pl}, then $\delta = (\Delta x, \Delta y)$ satisfies:*

$$|\Delta x| \le w_i + w_j - 1, \qquad |\Delta y| \le h_i + h_j - 1.$$

Proof. For T_i and T_j to be in contact (overlap or 4-adjacent), their bounding boxes must intersect or share an edge. The x-projections $[0, w_i - 1]$ and $[\Delta x, \Delta x + w_j - 1]$ intersect or are adjacent if $|\Delta x| \le w_i + w_j - 1$. The bound for Δy follows symmetrically. □

Remark 6 (Graph size vs. tree space)**.** Lemma 2 ensures that the *placement graph* G^{pl} has polynomial size: for each pair (T_i, T_j), there are $O((w_i + w_j)(h_i + h_j))$ candidate offsets to check. For uniform $w \times h$ strings, the placement graph has $O(n^2 wh)$ edges.

However, the *search space of spanning trees* remains exponential. By Cayley's formula, a complete graph on n vertices has n^{n-2} spanning trees; while our placement graph is typically sparser, the number of feasible placement trees can still grow exponentially with n. The significance of our reduction is not that it yields a polynomial search space, but that it transforms an *infinite continuous space* (absolute coordinates on $\mathbb{Z}^2$) into a *finite discrete combinatorial space* (spanning trees with bounded edge labels). This discretization enables exact enumeration for small instances and structured metaheuristic search for larger ones.

Remark 7 (Edge density and string entropy)**.** The bound $O(n^2 wh)$ on $|E(G^{\mathrm{pl}})|$ is a *worst-case geometric bound*—it counts all candidate offsets where bounding boxes are in contact. The actual number of *symbol-consistent* edges depends critically on the *information entropy* of the strings.

Let $C_{ij} \subseteq \mathbb{Z}^2$ denote the set of offsets δ such that $(i, j, \delta) \in E(G^{\mathrm{pl}})$. This set decomposes as:

$$C_{ij} = C_{ij}^{\mathrm{adj}} \cup C_{ij}^{\mathrm{ovl}},$$

where C_{ij}^{adj} contains offsets for 4-adjacency (bounding boxes share an edge but do not overlap), and C_{ij}^{ovl} contains offsets for symbol-consistent overlap.

- *Adjacency edges* $|C_{ij}^{\mathrm{adj}}|$: These are always present and contribute $O(w_i + w_j + h_i + h_j)$ edges per pair—the perimeter of the contact region. This component is independent of alphabet or string content.
- *Overlap edges* $|C_{ij}^{\mathrm{ovl}}|$: This depends on symbol compatibility. For an overlap of size k cells, symbol-consistency requires all k overlapping positions to match. If symbols are drawn uniformly from alphabet Σ:

  $$\Pr[k\text{-cell overlap is consistent}] = |\Sigma|^{-k}.$$

  Thus, for random strings over a large alphabet, $|C_{ij}^{\mathrm{ovl}}| \approx 0$ (almost no overlaps are valid), and G^{pl} is sparse—dominated by adjacency edges.
- *Periodic/low-entropy strings*: If strings are highly repetitive (e.g., all symbols identical, or periodic patterns), many overlaps become symbol-consistent. In the extreme case $|\Sigma| = 1$, *every* offset in the contact region is valid, yielding $|C_{ij}| = O(w_i w_j + h_i h_j)$ and a dense G^{pl}.

Summarizing, the effective edge count satisfies:

$$|E(G^{\mathrm{pl}})| = \Theta\big(n^2 \cdot (\text{perimeter} + \text{entropy-dependent overlap count})\big).$$

Our experimental instances use binary alphabets with moderate repetition, placing them in the intermediate regime where both adjacency and overlap edges contribute. This sensitivity to string structure is another manifestation of the *sequencing–packing spectrum* discussed in Section **??**.

Definition 5. Let p be a symbol-consistent placement of $\mathcal{T}$. The *contact graph* $G^{\mathrm{ct}}(p)$ is the subgraph of G^{pl} induced by p: it has vertex set $\{1, \dots, n\}$, and edge (i, j, δ) is present if $\delta = p(j) - p(i)$ and this edge exists in G^{pl}.

Equivalently, strings i and j are adjacent in $G^{\mathrm{ct}}(p)$ if they are in contact under p (overlapping or 4-adjacent).

The contact graph $G^{\mathrm{ct}}(p)$ records which edges of the placement graph are "realized" by a given placement. Different placements of the same instance may realize different subsets of edges.

Remark 8. Our cost functionals depend only on the bounding rectangle $B(p)$, not directly on the cardinality of $R(p)$. In particular, two placements with the same bounding box have the same cost for both 2D-SSP$_{\mathrm{area}}$ and 2D-SSP$_{\mathrm{bal}}$. The area-based objective directly generalizes the 1D shortest superstring objective (string length), while the balanced-area variant additionally constrains the aspect ratio. Both differ from geometric covering formulations that minimize the area of the union $R(p)$.

Assumption 1. In the structural discussion below we work with placements whose occupied region $R(p)$ is 4-connected. The following lemma shows this is without loss of optimality.

The following lemma establishes the *Connectivity Principle*—a structural reduction showing that the search for optimal solutions can be restricted to connected placements without loss of optimality. This result is analogous to "sliding" or "compaction" arguments in VLSI floorplanning (Murata et al., 1996a), where disconnected components can be moved together without increasing area. However, our setting introduces the additional constraint of *symbol-consistent overlaps*, which requires careful verification that the sliding procedure does not introduce symbol conflicts. In VLSI floorplanning, components are non-overlapping rectangles and any translation through empty space is trivially valid. In 2D-SSP, components may have complex shapes (the union of multiple overlapping string footprints), and we must ensure that sliding one component toward another does not cause newly occupied cells to conflict with existing symbols. The key insight enabling our proof is that *disjoint components slide through empty space*: since components are initially non-overlapping and non-adjacent, the translation path consists entirely of unoccupied cells, and contact occurs only at the moment of 4-adjacency—a boundary touch, not an overlap.

Lemma 3. *Let p be an optimal symbol-consistent placement for 2D-SSP (under either* $\mathrm{cost}_{\mathrm{area}}$ *or* $\mathrm{cost}_{\mathrm{bal}}$*). If $R(p)$ is not 4-connected, then there exists an optimal symbol-consistent placement p' such that $R(p')$ is 4-connected and* $\mathrm{cost}(p') \leq \mathrm{cost}(p)$.

Proof. Suppose $R(p)$ is not 4-connected. Then the occupied region decomposes into $k \geq 2$ maximal 4-connected components $R_1, R_2, \dots, R_k$. Each component R_ℓ corresponds to a subset $\mathcal{T}_\ell \subseteq \mathcal{T}$ of strings whose footprints are entirely contained in R_ℓ. Let B_ℓ denote the axis-aligned bounding box of R_ℓ, with width W_ℓ and height H_ℓ.

We prove by *strong induction on k* that all components can be merged into a single 4-connected region without increasing the bounding box dimensions.

Base case ($k = 2$): We have exactly two disconnected components R_1 and R_2. Since they are not 4-adjacent, there exists a *separation gap*—at least one empty cell lies on any 4-connected path between them. We translate R_2 toward R_1 until they become 4-adjacent.

Consider the axis-aligned directions. Since R_1 and R_2 are disjoint and not 4-adjacent, at least one of the following holds: (i) R_2 can be translated left, right, up, or down toward R_1 through empty space. Choose the translation direction that brings R_2 closest to R_1 without crossing any occupied cell. Translate R_2 by the minimum amount needed to make R_1 and R_2 share a 4-adjacent boundary.

Key observation: This translation cannot increase the global bounding box $B(p)$. If R_2 moves toward R_1 (inward), the combined footprint can only shrink or stay the same. The only way the bounding box could grow is if R_2 moves *away* from R_1, but we explicitly translate toward R_1, closing the gap.

After translation, R_1 and R_2 are 4-adjacent, forming a single connected component. Symbol-consistency is preserved since the translation path crosses only empty cells (by construction).

Inductive step ($k \geq 3$): Suppose the lemma holds for any configuration with fewer than k components. We distinguish two geometric cases:

Case A: Internal hole (nested bounding boxes). Suppose some component R_j has its bounding box B_j strictly contained in the interior of another component's bounding box B_i, i.e., $B_j \subset \operatorname{int}(B_i)$. Then R_j occupies a "hole" in the region surrounded by R_i.

We now describe a *symbol-consistency-preserving sliding procedure* adapted from VLSI compaction techniques but tailored to the 2D-SSP constraint regime. The key insight is that since R_j and R_i are initially *disjoint* (they occupy non-overlapping cells and are not 4-adjacent), a translation path exists that maintains symbol consistency throughout the motion.

Sliding procedure: Choose an axis-aligned direction $d \in \{\text{left}, \text{right}, \text{up}, \text{down}\}$ such that translating R_j in direction d moves it toward either (i) the boundary of B_i where R_i cells are located, or (ii) another hole component R_ℓ if one lies in that direction. Let Δ be the minimum translation distance in direction d such that R_j becomes 4-adjacent to either R_i or R_ℓ.

Claim: The translation preserves symbol-consistency. Consider any intermediate position $R_j^{(t)}$ during the translation, where $t \in [0, \Delta]$. At $t = 0$, the components R_j and R_i (and any other components) occupy disjoint cells. As t increases, $R_j^{(t)}$ sweeps through cells that were previously unoccupied. Since Δ is chosen as the *minimum* distance to achieve 4-adjacency:

- For all $t \in [0, \Delta)$, the cells newly occupied by $R_j^{(t)}$ are empty (not part of R_i or any other component), so no symbol conflicts can occur.
- At $t = \Delta$, the component $R_j^{(\Delta)}$ becomes 4-adjacent to another component. The 4-adjacency is a *contact* (shared boundary), not an overlap—the components touch but do not occupy the same cells.

Therefore, symbol-consistency is preserved: there are no overlapping cells between R_j and any other component, so no symbol matching is required. The translation is simply a rigid shift through empty space until contact.

Component reduction: After translation, R_j is 4-adjacent to either R_i or another hole component R_ℓ. In either case:

(a) If R_j becomes 4-adjacent to R_i, they merge into one component.
(b) If R_j becomes 4-adjacent to another hole component R_ℓ, they merge, reducing the total hole count.

Either way, the number of components decreases to $k - 1$, and by the induction hypothesis, the remaining components can be merged. All translations occur within B_i, so the global bounding box is unchanged.

Case B: External separation (disjoint bounding boxes). If no component is nested inside another, all component bounding boxes are "side by side" (possibly with diagonal separation, as in Figure 2).

Pick any two components R_i and R_j. Since they are not 4-adjacent, translate R_j toward R_i until they become 4-adjacent (using the same procedure as the base case). This reduces the component count to $k-1$. By the induction hypothesis, the remaining $k-1$ components can be merged into a single connected region.

Bounding box analysis: Each translation moves a component *inward* (toward another component), which cannot increase the global bounding box. In the worst case, the bounding box dimensions are preserved; typically, they decrease as gaps are closed.

By induction, all k components can be merged into a single 4-connected component with $\mathrm{cost}(p') \le \mathrm{cost}(p)$.

Verification of key properties:

1. *Symbol-consistency is preserved.* Before any translation, distinct components R_i and R_j occupy disjoint cells. Each translation moves a component through empty space toward another component (we always translate toward the nearest occupied region). Since the translation path contains no occupied cells, no new symbol conflicts are introduced. Symbol-consistency within each component is trivially preserved since internal offsets are unchanged.
2. *The bounding box dimensions do not increase.* Each translation moves a component *inward* (toward another component), closing a gap. This cannot increase the global bounding box—in the worst case, the dimensions are preserved; typically, they decrease as diagonal slack is eliminated (see Figure 2 for an example where diagonal slack allows bounding box reduction).
   In both cases (internal hole and external separation), $W' \le W$ and $H' \le H$, so:

$$\mathrm{cost}_{\mathrm{area}}(p') = W' \cdot H' \le W \cdot H = \mathrm{cost}_{\mathrm{area}}(p), \tag{1}$$

$$\mathrm{cost}_{\mathrm{bal}}(p') = \max(W', H') \le \max(W, H) = \mathrm{cost}_{\mathrm{bal}}(p). \tag{2}$$

Since p was optimal and $\mathrm{cost}(p') \le \mathrm{cost}(p)$, we have $\mathrm{cost}(p') = \mathrm{cost}(p)$, so p' is also optimal. □

Corollary 4. *For any instance of 2D-SSP, there exists an optimal placement whose occupied region is 4-connected. Consequently, Assumption 1 is justified for all instances.*

This *Connectivity Principle* establishes that restricting attention to connected placements is a *lossless reduction*: we lose no optimal solutions by imposing this structural requirement. The assumption matches our experimental focus and enables the tree-based structural perspective developed in this section. It is not required for the correctness of the algorithms in Section 4, which operate on arbitrary symbol-consistent placements.

Lemma 5. *Let p be a symbol-consistent placement such that $R(p)$ is 4-connected. Then the contact graph $G^{\mathrm{ct}}(p)$ is connected.*

Proof. If $G^{\mathrm{ct}}(p)$ were disconnected, we could partition the strings into two nonempty sets A and B with no edge between them. For each string T_i, let R_i be the set of global cells covered by T_i under p. Define

$$R_A = \bigcup_{i \in A} R_i, \qquad R_B = \bigcup_{j \in B} R_j,$$

so that $R(p) = R_A \cup R_B$ and $R_A \cap R_B = \emptyset$. By construction there is no pair of 4-adjacent cells across R_A and R_B, which contradicts 4-connectivity of $R(p)$. □

By the Connectivity Principle (Lemma 3) and Lemma 5, every optimal connected placement has a connected contact graph that admits a spanning tree. We now formalize this tree representation.

3.2. Placement trees

A placement tree encodes a solution as a spanning tree with labeled edges specifying relative offsets between strings. The key insight is that edge labels are drawn from a *bounded* set determined by string dimensions, which—combined with the finite number of spanning tree topologies—yields a finite search space.

Definition 6. A *placement tree* for $\mathcal{T}$ is a rooted tree $F = (V, E)$ with vertex set $V = \{1, \dots, n\}$ together with, for each edge $\{i, j\} \in E$, a label $\delta_{ij} \in \mathbb{Z}^2$ interpreted as the relative offset from i to j. For each oriented edge (i, j) we store δ_{ij} and require $\delta_{ji} = -\delta_{ij}$.

By Lemma 2, valid edge labels are bounded: if T_i and T_j have dimensions $w_i \times h_i$ and $w_j \times h_j$, then any symbol-consistent contact requires $|\Delta x| \le w_i + w_j - 1$ and $|\Delta y| \le h_i + h_j - 1$. This bounds the number of candidate labels per edge to $O(w_i w_j h_i h_j)$, ensuring the search space of placement trees is finite.

The *realization* of F with root r and root position $p(r) \in \mathbb{Z}^2$ is the placement p_F defined by

$$p_F(i) \;=\; p(r) + \sum_{t=0}^{\ell-1} \delta_{v_t v_{t+1}},$$

where $r = v_0, v_1, \dots, v_\ell = i$ is the unique simple path from r to i in F.

We call F *feasible* if the realization p_F is symbol-consistent for some (equivalently, every) choice of root position.

Remark 9 (Placements vs. Trees)**.** Not every symbol-consistent placement corresponds to a connected tree. If the occupied region $R(p)$ is disconnected, the contact graph $G^{\mathrm{ct}}(p)$ is a forest of multiple components, not a single spanning tree. However, by the Connectivity Principle (Lemma 3), *at least one optimal placement* has a connected occupied region, and thus corresponds to a spanning tree. This is the crux of our Search Space Reduction: we need not enumerate all placements (including disconnected ones), but only spanning trees of the placement graph.

Remark 10. Feasibility is a global property: even if all adjacent pairs (i, j) are locally consistent, collisions may occur between distant parts of the tree when their footprints overlap after summing the offsets. In particular, closed walks in the underlying contact graph induce non-trivial "loop-closure" constraints on the offsets. We treat feasibility algorithmically: given a placement tree F, we realize it via p_F and explicitly check symbol-consistency.

The following theorem establishes that optimal solutions can always be represented as placement trees, a crucial fact that justifies restricting our search to tree-based encodings.

Theorem 6 (Optimality-Preserving Equivalence)**.** *The following statements hold:*

(i) ***(Existence of tree-optimal solutions.)*** *For any instance of 2D-SSP, there exists an optimal placement p^* that corresponds to a feasible placement tree. Specifically, by Lemma 3, we may assume $R(p^*)$ is 4-connected; any spanning tree F of $G^{\mathrm{ct}}(p^*)$, equipped with edge labels $\delta_{ij} := p^*(j) - p^*(i)$, is then a feasible placement tree whose realization coincides with p^* up to global translation.*

(ii) ***(Completeness.)*** *Conversely, any feasible placement tree F induces a symbol-consistent placement p_F that is a valid solution to 2D-SSP.*

Proof. *(i) Optimal Placement → Tree.* Let p^* be any optimal placement. By Lemma 3, there exists an optimal placement p with the same cost such that $R(p)$ is 4-connected. By Lemma 5, $G^{\mathrm{ct}}(p)$ is connected, so it admits a spanning tree F. For any edge (i, j) of F, we have $p(j) = p(i) + \delta_{ij}$ by definition of the labels. For any vertex i with path $r = v_0, \dots, v_\ell = i$ from root r:

$$p(i) = p(r) + \sum_{t=0}^{\ell-1} \delta_{v_t v_{t+1}} = p_F(i).$$

Thus p equals the realization p_F (for root position $p(r)$), which is symbol-consistent, so F is feasible and optimal.

(ii) Tree → Placement. By feasibility, p_F is symbol-consistent. Every string T_i is placed exactly once, so T_i occurs as a 2D-substring of the induced array T_{p_F}, making it a 2D-superstring of $\mathcal{T}$. □

The significance of this theorem is that *no optimal solution is lost* by restricting to tree-based representations. While not every placement corresponds to a tree—disconnected placements correspond to forests of multiple trees—the Connectivity Principle ensures that at least one optimal solution has a connected contact graph, and hence corresponds to a single spanning tree. Combined with the bounded edge labels (Lemma 2), this establishes that the search space is both *finite* (bounded labels, finite tree topologies) and *complete* (contains at least one optimum). This justifies designing algorithms that search exclusively over placement trees.

Remark 11. All results in this section concern only symbol-consistency, connectivity, and combinatorial structure. They apply verbatim to both 2D-SSP$_{\text{area}}$ and 2D-SSP$_{\text{bal}}$.

Remark 12 (Extension to higher dimensions)**.** The theoretical framework developed in this section—the Connectivity Principle (Lemma 3), the Optimality-Preserving Equivalence (Theorem 6), and the bounded offset property (Lemma 2)—relies solely on translation operations and adjacency relations, with no dependence on the specific dimensionality of the underlying lattice. Consequently, these results extend directly to d-dimensional SSP for any $d \geq 1$: the contact graph, placement graph, and spanning-tree representation generalize naturally to $\mathbb{Z}^d$, with $2d$-adjacency replacing 4-adjacency and offset labels becoming d-tuples. The core insight—that optimal solutions can be restricted to connected placements representable as spanning trees with bounded labels—holds in arbitrary dimension, making this framework a principled foundation for higher-dimensional generalizations.

3.3. Algorithmic implications

Theorem 6 has two key algorithmic consequences. First, it guarantees that *optimal solutions are always reachable* via tree-based search: since at least one optimal placement corresponds to a feasible placement tree, any algorithm that exhaustively searches over placement trees is guaranteed to find an optimum. Second, combined with the bounded offset property (Lemma 2), it yields a *finite discrete search space*: while the number of spanning trees can be exponential in n (see Remark 6), this is a dramatic improvement over the infinite space of absolute coordinate vectors. For small instances, this enables exact ILP enumeration; for larger instances, it provides a well-structured combinatorial space amenable to metaheuristic exploration.

Remark 13 (Existence vs. heuristic reachability)**.** Theorem 6 establishes the *existence* of an optimal placement tree, but our GA does not exhaustively enumerate all trees. In particular, the crossover operator (Algorithm 4) uses a *greedy completion* step: when recombination produces an incomplete offspring (missing some strings), the missing strings are attached via a locally-optimal greedy heuristic. This means the GA effectively searches the subspace of "greedily-completable" trees.

A natural question arises: *can the optimal tree be "ungreedy"?* That is, might the globally optimal placement require a locally suboptimal edge—one that a greedy heuristic would never select—to achieve the minimum cost? In principle, yes: the optimal tree might contain an edge (i, j, δ) that is dominated by another edge (i, j, δ') with smaller local cost, yet δ enables a globally superior arrangement for subsequent strings.

Two design choices mitigate this gap. First, the stochastic variant of greedy completion randomly samples among equally-good candidates rather than deterministically choosing one, introducing exploration of alternative attachment points. Second, and more substantially, empirical analysis of solution provenance (Table 5) reveals that *crossover dominates*: typically 95–97.5% of string placements in offspring are inherited directly from parents, with greedy completion responsible for only 3–5% of placements. This means the vast majority of structural decisions are made by recombining parent subtrees—which themselves encode accumulated evolutionary "wisdom"—rather than by local greedy choices. The greedy-completion bias thus affects only a small fraction of the solution structure, substantially reducing the risk that it blocks access to globally optimal configurations.

Empirically, this mitigation appears effective: our experimental results (Section 5) show that the GA matches or closely approaches exact ILP solutions on small instances, suggesting that for typical inputs, the greedy-completion subspace contains near-optimal solutions. Nevertheless, closing this theoretical gap—proving conditions under which greedy completion preserves optimality, or designing non-greedy repair operators—remains an interesting direction for future work.

The tree structure has three additional properties that make it particularly suitable for metaheuristic optimization:

1. *Locality preservation.* In a placement tree, strings that are geometrically close in the final arrangement are typically close in the tree (separated by few edges). This means that a subtree corresponds to a spatially coherent cluster of strings. When crossover transplants a subtree from one parent to another, it preserves the internal structure of this cluster, the relative offsets that make these strings fit together well.
2. *Incremental realizability.* A placement tree can be "grown" by adding one string at a time along tree edges, always maintaining a valid partial placement. This property directly enables our tree-growing greedy heuristic (Section 4.3) and ensures that the GA's crossover operator can incrementally build feasible offspring.
3. *Reduced redundancy (symmetry breaking).* A coordinate-based representation assigns each of n strings an (x, y) offset, yielding a $2n$-dimensional search space. However, due to translational invariance, infinitely many

Table 1
Summary of main notation.

Symbol	Meaning
Σ	Alphabet
$\mathcal{T} = \{T_1, \dots, T_n\}$	Input set of strings
$C_T \subset \mathbb{Z}^2$	Local cell coordinates of string T
P	Pattern 2D-string (for occurrences)
$\mathrm{Occ}(P, T)$	Set of occurrences of P in T
S	2D-superstring of $\mathcal{T}$
$\lvert S\rvert_{\mathrm{area}}$	Area of minimal bounding box of S
$\lvert S\rvert_{\mathrm{bal}}$	Balanced side length $\max\{m, n\}$ of S
$p(i) = (x_i, y_i)$	Placement (offset) of string T_i
$R(p) \subset \mathbb{Z}^2$	Union of occupied cells under placement p
$B(p)$	Minimal axis-aligned bounding box of $R(p)$
$W(p), H(p)$	Width and height of $B(p)$
$\mathrm{cost}_{\mathrm{area}}(p)$	Area-based cost $W(p)H(p)$
$\mathrm{cost}_{\mathrm{bal}}(p)$	Balanced-area cost $\max\{W(p), H(p)\}$
$G^{\mathrm{ct}}(p)$	Contact graph induced by placement p
G^{pl}	Placement graph of symbol-consistent offsets
$F = (V, E)$	Placement tree on $\{1, \dots, n\}$
$\delta_{ij} \in \mathbb{Z}^2$	Relative offset from i to j in F
p_F	Realization (tree-induced placement) of F

coordinate vectors encode the same placement: shifting all coordinates by any vector $(\Delta x, \Delta y)$ produces an equivalent solution. A naïve discretization to a $W \times H$ grid still yields $O((WH)^n)$ candidate placements.
The tree representation achieves principled *symmetry breaking*. By encoding only *relative offsets* between adjacent strings, global translation is factored out: the placement is determined up to a single global shift, which can be canonicalized by fixing one string at the origin. The effective search space is the set of spanning trees of the placement graph G^{pl}, each with $n-1$ labeled edges. While this space can still be exponential (up to n^{n-2} trees by Cayley's formula for complete graphs), it is dramatically smaller than the coordinate space and, crucially, *non-redundant*: distinct trees correspond to distinct relative arrangements.

These properties motivate our choice to design a genetic algorithm whose individuals are placement trees and whose crossover operates on subtrees, rather than using a more conventional coordinate-based representation.

4. Algorithms

Having established the structural foundation linking placements and trees, we now present three algorithmic approaches for solving 2D-SSP. These methods span the spectrum from exact to heuristic, offering different trade-offs between solution quality and computational cost:

- An *exact ILP formulation* (Section 4.1) that enumerates all candidate placements on a discrete grid and optimizes over them using mixed-integer linear programming. This approach guarantees optimal solutions but is limited to small instances.
- A *merge-based greedy heuristic* (Section 4.2) adapted from the classical 1D Shortest Superstring algorithm. This baseline repeatedly merges pairs of partial superstrings with maximum overlap.
- A *tree-growing greedy heuristic* (Section 4.3) that builds a placement tree incrementally, motivated by Theorem 6. At each step it attaches a new string to minimize the bounding-box cost.
- A *tree-based genetic algorithm* (Section 4.4) that represents individuals as placement trees and uses crossover operators that recombine subtrees. This approach aims to combine the quality of exact methods with the scalability of heuristics.

All three methods can be instantiated with either of the two bounding-box cost variants, 2D-SSP$_{\text{area}}$ or 2D-SSP$_{\text{bal}}$.

The genetic algorithm uses placement trees rather than the more obvious coordinate-based representation (where each individual is a vector of (x, y) coordinates for each string). This choice is motivated by several considerations. First, in the coordinate representation, an offspring produced by crossover rarely inherits good local structure from its parents: if parent 1 places strings A and B in a well-overlapping configuration, and parent 2 places strings B and C similarly, a crossover that takes A from parent 1 and C from parent 2 will likely place them far apart, destroying both favorable overlaps. In contrast, our tree-based crossover transplants entire subtrees, preserving the relative offsets among all strings in the subtree. Second, the coordinate representation is highly redundant: any global translation of a placement yields the same objective value, so the search space contains infinitely many equivalent solutions. The tree representation eliminates this redundancy by encoding only the pairwise offsets that matter. Third, the tree structure aligns naturally with the connectivity requirement: a spanning tree automatically ensures that all strings are geometrically connected, whereas the coordinate representation requires additional constraints or repair operators to enforce connectivity.

4.1. Exact Verification via ILP

To validate the solution quality of our heuristic approaches, we formulate a direct grid-based mixed-integer linear program (Wolsey, 1998). We emphasize that this formulation is *not* intended as a scalable solver for general instances, but strictly as a *ground-truth oracle* for small-scale verification ($N \leq 10$). This allows us to measure exactly how close our genetic algorithm comes to the global optimum.

The model works with discrete candidate placements of each 2D string on a finite grid: it enumerates a finite set of allowed origins for each 2D string, chooses exactly one origin per string, forbids symbol conflicts between strings, and then optimizes the axis-aligned bounding box enclosing all occupied cells according to the chosen objective variant.

Throughout this subsection we reuse the notation from Section 3. In particular, we have a finite multiset of 2D strings $\mathcal{T} = \{T_1, \dots, T_n\}$, each represented by a finite set of local cells $C_i \subset \mathbb{Z}^2$ and a symbol function $T_i : C_i \to \Sigma$. For each i we denote the local bounding box of T_i by

$$x_i^{\min} = \min_{(u,v)\in C_i} u, \qquad x_i^{\max} = \max_{(u,v)\in C_i} u,$$
$$y_i^{\min} = \min_{(u,v)\in C_i} v, \qquad y_i^{\max} = \max_{(u,v)\in C_i} v,$$

and its width and height by

$$w_i = x_i^{\max} - x_i^{\min} + 1, \qquad h_i = y_i^{\max} - y_i^{\min} + 1.$$

We embed all strings into a common rectangular grid $[0, W_g] \times [0, H_g] \subset \mathbb{Z}^2$. To guarantee that the ILP is a *true exact solver*, the grid must be large enough to accommodate *any* possible optimal arrangement, including those with extreme aspect ratios.

Remark 14 (Grid bounds and exactness)**.** A naïve approach would set $W_g = \sum_i w_i$ and $H_g = \sum_i h_i$ (the maximum possible dimensions if all strings are placed in a line with no overlap). This guarantees global optimality but creates an enormous grid.

A tempting optimization is to use greedy bounds: run a heuristic to obtain dimensions $W_{\text{greedy}} \times H_{\text{greedy}}$, then set $W_g := W_{\text{greedy}}$ and $H_g := H_{\text{greedy}}$. However, this is *not* sound for all objectives:

- *For 2D-SSP*$_{\text{area}}$*:* An optimal solution may have a different aspect ratio than the greedy solution. For example, if greedy yields a 10×10 placement (area 100), the optimal might be 2×40 (area 80). If $W_g = H_g = 10$, the ILP would exclude the 2×40 solution since $40 > 10$.
- *For 2D-SSP*$_{\text{bal}}$*:* The greedy bound *is* sound. If greedy achieves $\max\{W_{\text{greedy}}, H_{\text{greedy}}\} = L$, any optimal solution has $\max\{W^*, H^*\} \leq L$, so both dimensions are bounded by L.

To ensure global exactness for both objectives, we use:

$$W_g := \min\Big(\sum_i w_i,\ \text{cost}_{\text{area}}^{\text{greedy}}\Big), \qquad H_g := \min\Big(\sum_i h_i,\ \text{cost}_{\text{area}}^{\text{greedy}}\Big).$$

This exploits the fact that if the greedy area is A_{greedy}, any optimal area satisfies $W^* \cdot H^* \leq A_{\text{greedy}}$, so $W^* \leq A_{\text{greedy}}$ (since $H^* \geq 1$). The grid remains bounded by the greedy area rather than the sum of all dimensions, while accommodating all aspect ratios.

For small verification instances, this grid size is manageable. For larger instances, the ILP becomes intractable regardless of grid bounds.

To eliminate redundant symmetric solutions arising from translation invariance, we apply *symmetry breaking*: we fix the first string T_1 at the origin by restricting its set of allowed origins to $\mathcal{O}_1 := \{(0,0)\}$. This constraint removes all translated copies of any given solution from the search space without affecting optimality.

For each string $i \geq 2$ we define a finite set of allowed *origins* $\mathcal{O}_i \subset \mathbb{Z}^2$ such that translating T_i by o keeps all its local cells inside the global grid:

$$\mathcal{O}_i := \Big\{ o = (x,y) \in \mathbb{Z}^2 \,\Big|\, (x+u, y+v) \in [0, W_g] \times [0, H_g] \text{ for all } (u,v) \in C_i \Big\}. \tag{3}$$

Recall that $\mathcal{O}_1 = \{(0,0)\}$ by the symmetry-breaking constraint. When T_i is placed at an origin $o = (x,y)$, each local cell $(u,v) \in C_i$ is mapped to global coordinates $(x+u, y+v)$.

Two candidate placements (i,o) and (j,o') are incompatible if they assign different symbols to the same global coordinate. Formally, we precompute a Boolean conflict indicator

$$\kappa_{ijoo'} = \begin{cases} 1, & \text{if there exist } (u,v) \in C_i,\ (u',v') \in C_j \\ & \text{with } (x+u, y+v) = (x'+u', y'+v') \\ & \text{and } T_i(u,v) \neq T_j(u',v'), \\ & \text{for } o = (x,y),\ o' = (x',y'), \\ 0, & \text{otherwise,} \end{cases} \tag{4}$$

for all $i < j$, $o \in \mathcal{O}_i$, and $o' \in \mathcal{O}_j$. This preprocessing reduces symbol consistency to simple pairwise constraints in the ILP.

The model uses the following variables.

- For each string $i \in \{1, \dots, n\}$ and each origin $o \in \mathcal{O}_i$:

  $b_{io} \in \{0,1\}$ (1 if T_i is placed at origin o, 0 otherwise).

- Integer coordinates of the global bounding box:

  $X_{\min}, X_{\max}, Y_{\min}, Y_{\max} \in \mathbb{Z}$,

  which represent the minimum and maximum global row/column indices among all occupied cells.

- The width and height of the bounding box:

  $W, H \in \mathbb{Z}_{\geq 0}$.

- A maximum side variable (for the balanced-area objective):

  $L \in \mathbb{Z}_{\geq 0}$,

  representing the maximum of width and height.

- An area variable (for the area objective):

  $A \in \mathbb{Z}_{\geq 0}$,

  used to model or approximate the bounding-box area $W \cdot H$.

In the implementation we bound these variables by a constant

$$M \;:=\; \max\{W_g, H_g\} + \max_i \max\{w_i, h_i\},$$

so that $X_{\min}, X_{\max}, Y_{\min}, Y_{\max} \in [-M, M]$ and $W, H, L \in [0, M]$, $A \in [0, M^2]$.

Remark 15 (Big-M calibration)**.** The choice of M involves a trade-off. If M is too small, valid placements may be incorrectly excluded; if M is excessively large, the LP relaxation becomes weak (the big-M constraints provide little tightening when $b_{io} = 0$), leading to slow branch-and-bound convergence.

Our choice $M = \max\{W_g, H_g\} + \max_i \max\{w_i, h_i\}$ is valid given the grid bounds from Remark 14: since the grid accommodates all optimal solutions, no coordinate can exceed $\max\{W_g, H_g\}$ plus the maximum string dimension. The bound is instance-adaptive and typically much smaller than a naïve bound like $n \cdot \max_i \max\{w_i, h_i\}$, improving LP relaxation quality.

Both objective variants share the following groups of constraints.

(i) Exactly one origin per string. Each 2D string must be placed at exactly one origin:

$$\sum_{o \in \mathcal{O}_i} b_{io} \;=\; 1 \qquad \forall i \in \{1, \dots, n\}. \tag{5}$$

(ii) No symbol conflicts. If two candidate placements (i, o) and (j, o') conflict, they cannot be chosen simultaneously:

$$b_{io} + b_{jo'} \;\le\; 1 \quad \forall\, i < j,\ \forall\, o \in \mathcal{O}_i,\ \forall\, o' \in \mathcal{O}_j \text{ with } \kappa_{ijoo'} = 1. \tag{6}$$

(iii) Bounding box must contain all placed strings. Let $o = (x, y) \in \mathcal{O}_i$ be a candidate origin for T_i. If $b_{io} = 1$, then the global footprint of T_i is

$$[x + x_i^{\min},\ x + x_i^{\max}] \times [y + y_i^{\min},\ y + y_i^{\max}],$$

and this rectangle must lie inside $[X_{\min}, X_{\max}] \times [Y_{\min}, Y_{\max}]$. We encode these implications with big-M constraints:

$$X_{\min} \;\le\; x + x_i^{\min} + M(1 - b_{io}), \tag{7}$$
$$X_{\max} \;\ge\; x + x_i^{\max} - M(1 - b_{io}), \tag{8}$$
$$Y_{\min} \;\le\; y + y_i^{\min} + M(1 - b_{io}), \tag{9}$$
$$Y_{\max} \;\ge\; y + y_i^{\max} - M(1 - b_{io}), \tag{10}$$

for all i and all $o = (x, y) \in \mathcal{O}_i$. When $b_{io} = 1$ these reduce to the desired inequalities $X_{\min} \le x + x_i^{\min}$, $X_{\max} \ge x + x_i^{\max}$, etc., and when $b_{io} = 0$ they are relaxed by the big-M terms.

(iv) Definition of width and height. The bounding box dimensions are defined by

$$W = X_{\max} - X_{\min} + 1, \qquad H = Y_{\max} - Y_{\min} + 1. \tag{11}$$

In addition, W and H must be at least as large as the widest and tallest individual 2D string:

$$W \;\ge\; \max_i w_i, \qquad H \;\ge\; \max_i h_i. \tag{12}$$

4.1.1. Balanced-Area Objective

The balanced-area variant minimises the maximum side length $\max\{W, H\}$. We link L to W and H via

$$L \;\ge\; W, \qquad L \;\ge\; H. \tag{13}$$

At optimality, $L = \max\{W, H\}$.

The *balanced-area* ILP is

$$\begin{aligned} \min \quad & L \\ \text{s.t.} \quad & (5) - (12),\ (13), \\ & b_{io} \in \{0, 1\}\ \forall i, o, \quad X_{\min}, X_{\max}, Y_{\min}, Y_{\max}, W, H, L \in \mathbb{Z}. \end{aligned} \tag{14}$$

4.1.2. Area Objective

The area-based variant uses the product $W \cdot H$ as its cost. Conceptually we want

$$A = W \cdot H. \tag{15}$$

Remark 16 (Linearization of the bilinear term)**.** The product $W \cdot H$ makes the area objective inherently *non-convex*. A direct formulation would yield a Mixed-Integer *Nonlinear* Program (MINLP), which is significantly harder to solve than a Mixed-Integer *Linear* Program (MILP).

We linearize this bilinear term using the *McCormick envelope*—the tightest convex relaxation of the product over a box. This is a standard technique in global optimization (McCormick, 1976). The McCormick envelope provides the *convex hull* of points $(W, H, W \cdot H)$ when W and H are continuous, but for integer variables, it is only a relaxation: intermediate points satisfying the envelope constraints may not correspond to integer solutions with $A = W \cdot H$.

For our small verification instances, this relaxation is tight in practice: the branch-and-bound solver finds integer solutions where the McCormick constraints are binding. Alternative approaches for tighter formulations include:

- *Logarithmic discretization:* Introduce binary variables for each bit of W and H, yielding $O(\log W_{\max} \cdot \log H_{\max})$ auxiliary variables and an exact linearization.
- *SOS2 piecewise-linear approximation:* Approximate $W \cdot H$ using special ordered sets of type 2, trading exactness for tighter LP relaxations.
- *Surrogate minimization:* Since we minimize area, we could minimize $(W + H)$ as a surrogate (perimeter), which is linear. However, this changes the objective and may yield suboptimal area solutions.

We chose McCormick for simplicity, as it sufficed for our small test cases. For larger instances requiring tighter relaxations, logarithmic discretization would be preferred.

In contrast, the balanced objective $\max\{W, H\}$ is trivially linearized via auxiliary variable $L \geq W$, $L \geq H$—no relaxation is needed.

To remain within a MILP framework we linearize this product over a known box $W \in [W_{\min}, W_{\max}]$, $H \in [H_{\min}, H_{\max}]$. We take

$$W_{\min} := \max_i w_i, \quad H_{\min} := \max_i h_i, \quad W_{\max}, H_{\max} \leq M,$$

and impose the standard McCormick envelope

$$A \ \geq \ W_{\min} H + H_{\min} W - W_{\min} H_{\min}, \tag{16}$$
$$A \ \geq \ W_{\max} H + H_{\max} W - W_{\max} H_{\max}, \tag{17}$$
$$A \ \leq \ W_{\min} H + H_{\max} W - W_{\min} H_{\max}, \tag{18}$$
$$A \ \leq \ W_{\max} H + H_{\min} W - W_{\max} H_{\min}. \tag{19}$$

These constraints define the convex hull of all triples (W, H, A) with $A = W \cdot H$ and (W, H) in the given box. On small instances one can further tighten this relaxation by introducing a discrete piecewise-linear approximation of $W \cdot H$, but the simple McCormick envelope above already proved sufficient for our experiments.

The *bounding area* ILP is

$$\begin{aligned} \min \quad & A \\ \text{s.t.} \quad & (5) - (12),\ (16) - (19), \\ & b_{io} \in \{0,1\}\ \forall i, o, \quad X_{\min}, X_{\max}, Y_{\min}, Y_{\max}, W, H, A \in \mathbb{Z}. \end{aligned} \tag{20}$$

The objective value $A^\star$ coincides with the area of the minimal axis-aligned bounding rectangle $B(p)$ enclosing all occupied cells in the induced placement p.

This formulation is conceptually straightforward: each b_{io} encodes a specific choice of origin for string T_i; the conflict constraints (6) enforce symbol consistency; the big-M constraints (7)–(10) define a global bounding box that contains all chosen placements; and depending on the variant, either (13) or (16)–(19) expresses the chosen cost. The greedy-based grid bounds and symmetry breaking significantly reduce the search space, but the number of variables and conflict constraints still grows quickly with the number of strings. Consequently, this ILP is practically limited to small instances ($N \leq 10$) and serves exclusively as a verification tool to certify the optimality gap of our heuristic methods.

4.2. Merge-Based Greedy Heuristic

Before presenting our tree-growing heuristic, we describe a natural baseline adapted from the classical 1D Shortest Superstring Problem: the *merge-based greedy* algorithm.

In the 1D setting, the Greedy Superstring Algorithm (Blum et al., 1994) is remarkably effective: it repeatedly merges the pair of strings with maximum overlap until a single superstring remains. This simple strategy achieves a 4-approximation for 1D-SSP (Blum et al., 1994), later improved to 2.5 (Kaplan and Shafrir, 2005). Empirically, the algorithm performs far better than these worst-case bounds suggest: extensive experiments on both random and biological sequences show approximation ratios consistently below 1.05 (Cazaux and Rivals, 2018), making it a strong practical baseline.

We adapt this approach to the 2D setting as follows. Given a multiset $\mathcal{T}$ of 2D strings, we maintain a set of *partial superstrings* (initially, each string is its own partial superstring). At each step, we identify the pair (S_i, S_j) of partial superstrings that can be merged with maximum symbol-consistent overlap, merge them into a single partial superstring, and repeat until only one remains.

Crucially, the overlap function measures *geometrical overlap*—the area of the overlapping region—rather than the number of matching non-wildcard characters:

$$\text{overlap}(S_i, S_j) = \max_{\delta \in \mathcal{D}_{ij}} |R_i \cap (R_j + \delta)|,$$

where R_i, R_j are the cell sets of S_i, S_j, the offset δ ranges over all translations that yield symbol-consistent overlap, and $|\cdot|$ denotes the cardinality of the intersection (i.e., the number of overlapping cells, regardless of whether those cells contain alphabet symbols or wildcards).

Geometrical vs. character overlap. In the 1D setting, maximizing geometrical overlap and maximizing character overlap are equivalent: every overlapping position contains exactly one character. In the 2D setting, however, these objectives diverge. Consider two 3×3 strings that can overlap in two ways: (a) a 2×2 region with 4 matching characters, or (b) a 1×3 strip with 3 matching characters. A character-based criterion would prefer (a), but this choice might force the merged component into an unfavorable shape that increases the final bounding box.

We use geometrical overlap because it directly relates to the bounding-box objective: maximizing the area of overlap is equivalent to minimizing the area increase when merging. Formally, if S_i has area A_i and S_j has area A_j, then the merged component has area

$$A_{\text{merged}} = A_i + A_j - |R_i \cap (R_j + \delta)|,$$

so maximizing geometrical overlap minimizes A_{merged}. This observation motivates our choice of overlap function. The merge operation places S_j at the offset δ^* achieving this maximum, creating a new partial superstring whose bounding box contains both.

Hypothesis: 1D vs. 2D performance. We hypothesize that the merge-based greedy performs well when the input strings are *nearly one-dimensional* (i.e., have aspect ratios close to $1 \times m$ or $m \times 1$), since this regime closely resembles the classical 1D setting where the algorithm has provable guarantees. However, as strings become *genuinely two-dimensional* (aspect ratios closer to 1), the merge-based approach may struggle: the "best overlap" criterion optimizes locally for overlap size but ignores how the merge affects the global bounding-box shape. In contrast, our tree-growing heuristic (Section 4.3) explicitly optimizes the bounding-box cost at each step, which we expect to yield better solutions for 2D instances.

The procedure is summarized in Algorithm 1.

We test this hypothesis experimentally in Section 5: Table **??** compares merge-based greedy against our methods on instances with 1×8 strings (nearly 1D), while Tables 2 and **??** evaluate performance on 3×3 strings (genuinely 2D).

4.3. Tree-Growing Greedy Heuristic

Motivated by Theorem 6, which establishes that every connected placement corresponds to a spanning tree of relative offsets, we design a greedy heuristic that directly constructs a *placement tree*. Rather than merging pairs of partial superstrings as in the merge-based approach, we grow a single tree by iteratively attaching strings to the current structure.

The algorithm maintains:

Algorithm 1 Merge-Based Greedy for 2D-SSP

```
Require: Set of 2D strings $\mathcal{T} = \{T_1, \dots, T_n\}$
Ensure: 2D superstring $S$
1: $\mathcal{S} \leftarrow \{T_1, \dots, T_n\}$                                  ▷ Set of partial superstrings
2: while $|\mathcal{S}| > 1$ do
3:     $(S_i^*, S_j^*, \delta^*) \leftarrow \arg\max_{S_i, S_j \in \mathcal{S}, \delta} \text{overlap}(S_i, S_j, \delta)$    ▷ Find best symbol-consistent overlap
4:     $S_{\text{merged}} \leftarrow \textsc{Merge}(S_i^*, S_j^*, \delta^*)$          ▷ Create merged superstring
5:     $\mathcal{S} \leftarrow (\mathcal{S} \setminus \{S_i^*, S_j^*\}) \cup \{S_{\text{merged}}\}$
6: end while
7: return the single element of $\mathcal{S}$
```

- A *placement tree* $F = (V_F, E_F)$ with $V_F \subseteq \{1, \dots, n\}$ representing the strings placed so far;
- A *canvas* of occupied global cells with their symbols;
- The bounding-box coordinates $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$ of the current placement.

At each iteration, we select an unplaced string $T_j \notin V_F$ and attach it to some string $T_i \in V_F$ via an edge (i, j, δ) from the placement graph G^{pl}, choosing the pair (i, j, δ) that minimizes the resulting bounding-box cost while maintaining symbol-consistency. This directly mirrors the structural insight of Theorem 6: we are constructing an optimal placement by "growing" a spanning tree one edge at a time.

Crucially, by Lemma 2, we need only consider offsets δ within the bounded window $|\Delta x| \le w_i + w_j - 1$ and $|\Delta y| \le h_i + h_j - 1$. This transforms what might seem like an unbounded search into a tractable enumeration: for each candidate parent T_i in the current tree and each unplaced string T_j, we examine $O(w_i h_i + w_j h_j)$ candidate attachment positions.

The current width and height are

$$w(C) = x_{\max} - x_{\min} + 1, \qquad h(C) = y_{\max} - y_{\min} + 1,$$

and we define either

$$\text{size}_{\text{bal}}(C) = \max\{w(C), h(C)\} \quad \text{or} \quad \text{size}_{\text{area}}(C) = w(C) \cdot h(C)$$

depending on the chosen objective.

For any candidate attachment of a string, we can (i) check whether overlapping cells agree symbolically, and (ii) compute the resulting bounding-box cost.

For a given target side length L in the balanced-area case, we enumerate translations $(\Delta x, \Delta y)$ of a string T that would result in a bounding box of side length exactly L when combined with the current canvas. Intuitively:

- if the canvas is empty, we place the first string so that the bounding box matches its own width/height;
- otherwise, when L equals the current size s, we slide the string in all ways that keep the bounding box within a virtual $s \times s$ square;
- when $L > s$, we consider placements that extend this square along one of its four sides so that the new size becomes exactly L.

Symbol-consistency of these candidate placements is checked against the canvas. For the area-based variant we similarly enumerate candidate positions and evaluate them according to the area-based cost.

4.3.1. Deterministic Tree-Growing

The deterministic variant starts from a chosen root string T_r (placed at the origin) and iteratively attaches the remaining strings. At each step, it considers all candidate edges (i, j, δ) where T_i is already in the tree and T_j is not, and selects the edge that:

(i) achieves the smallest possible increase in the chosen bounding-box cost, and
(ii) among those, maximizes the *character overlap*—the number of overlapping cells containing matching non-wildcard symbols—as a tie-breaker.

Rationale for the two-level criterion. The primary criterion (minimizing bounding-box cost increase) is equivalent to maximizing geometrical overlap: as shown in Section 4.2, if the current canvas has area A and the new string has area A_j, then

$$\Delta A = A_j - |\text{geometrical overlap}|,$$

so minimizing ΔA is the same as maximizing geometrical overlap. The secondary criterion (character overlap) serves as a tie-breaker when multiple attachments achieve the same bounding-box cost. This design reflects a key insight: in the 2D setting, controlling the bounding-box shape is the primary concern, but among equally good placements (in terms of bounding box), preferring those with more character matches encourages compression and may improve solution quality.

The algorithm terminates when all n strings have been added to the tree. By construction, it always increases the cost threshold until a consistent attachment exists for some string, and therefore returns a complete placement tree spanning all strings.

This procedure directly implements the "growing" perspective of Theorem 6: we construct a spanning tree of the contact graph by adding one vertex at a time, always maintaining a valid partial placement.

4.3.2. Stochastic Tree-Growing

The stochastic variant follows the same tree-growing structure but introduces randomization in the selection among equally good candidate edges. Instead of deterministically picking the single best attachment, it uses *roulette wheel selection* among all candidates that achieve the minimal cost increase, with selection probabilities proportional to their *character overlap* (number of matching non-wildcard symbols). Specifically, for candidates $\{(i_1, j_1, \delta_1), \dots, (i_k, j_k, \delta_k)\}$ all achieving cost c^*, the probability of selecting candidate ℓ is

$$P(\ell) = \frac{\text{char\_overlap}(i_\ell, j_\ell, \delta_\ell)}{\sum_{m=1}^{k} \text{char\_overlap}(i_m, j_m, \delta_m)},$$

where char_overlap(i, j, δ) denotes the number of overlapping cells containing matching non-wildcard symbols when T_j is placed at offset δ relative to T_i. This weighting biases the selection toward attachments with more character matches (favoring compression) while still allowing exploration of alternative placements.

This stochastic heuristic often yields slightly worse single solutions than the deterministic variant, but it produces a diverse set of placement trees, which is useful for initializing the genetic algorithm population. The diversity arises because different runs may grow the tree in different orders, exploring distinct regions of the solution space.

The tree-growing procedure is summarized in Algorithm 2.

4.4. Tree-based genetic algorithm

We now describe the tree-based genetic algorithm (GA) used in our experiments. Recall that a solution is represented as a *placement tree* $F = (V, E, r)$ whose vertices are strings, whose directed edges $(u \to v)$ are annotated by integer offsets $(\Delta x, \Delta y)$, and whose root r is the string placed at the origin. Decoding such a tree yields a concrete placement and an objective value.

As a preprocessing step we build a *placement graph* $G = (V, E_G)$ on the strings, whose directed edges encode all locally valid relative placements between string pairs. For each ordered pair of distinct strings (u, v) we enumerate translations of v relative to u within the bounded search window established by Lemma 2: $|\Delta x| \le w_u + w_v - 1$ and $|\Delta y| \le h_u + h_v - 1$. We collect all offsets within this window that yield symbol-consistent overlaps or 4-adjacent contact. This bounded enumeration is crucial for efficiency: for uniform $w \times h$ strings, we examine $O(wh)$ candidate offsets per pair rather than an unbounded search space.

To evaluate a tree $F = (V, E, r)$ we traverse it from the root r, assign absolute coordinates $p_F(i)$ to every 2D string T_i by summing the edge offsets along the unique path from r to i, and construct the global canvas, rejecting any edge that would create a symbol conflict. The fitness is then the bounding-box cost of the resulting placement, either

$$\text{cost}_{\text{area}}(p_F) = W(p_F) \cdot H(p_F) \quad \text{or} \quad \text{cost}_{\text{bal}}(p_F) = \max\{W(p_F), H(p_F)\},$$

depending on which objective variant is being optimized.

Algorithm 2 Tree-Growing Greedy for 2D-SSP

```
Require: Set of 2D strings $\mathcal{T} = \{T_1, \dots, T_n\}$, root index $r$, objective cost $\in \{\text{area}, \text{bal}\}$, mode $\in \{\textsc{Det}, \textsc{Stoch}\}$
Ensure: Placement tree $F$ with placement $p_F$
 1: $V_F \leftarrow \{r\}$; $E_F \leftarrow \emptyset$; $p_F(r) \leftarrow (0,0)$
 2: $canvas \leftarrow$ cells of $T_r$ at origin
 3: while $|V_F| < n$ do
 4:     $candidates \leftarrow \emptyset$
 5:     for each $i \in V_F$, each $j \notin V_F$, each valid offset $\delta$ do
 6:         if attaching $T_j$ at $p_F(i) + \delta$ is symbol-consistent with $canvas$ then
 7:             $c \leftarrow \text{cost}(canvas \cup \text{cells of } T_j \text{ at } p_F(i) + \delta)$
 8:             $candidates \leftarrow candidates \cup \{(i, j, \delta, c)\}$
 9:         end if
10:     end for
11:     $c^* \leftarrow \min\{c : (i, j, \delta, c) \in candidates\}$
12:     $best \leftarrow \{(i, j, \delta) : (i, j, \delta, c^*) \in candidates\}$
13:     if mode = $\textsc{Det}$ then
14:         $(i^*, j^*, \delta^*) \leftarrow$ element of $best$ maximizing overlap
15:     else
16:         // Roulette wheel selection weighted by overlap
17:         $w_\ell \leftarrow \text{overlap}(i_\ell, j_\ell, \delta_\ell)$ for each $(i_\ell, j_\ell, \delta_\ell) \in best$
18:         $(i^*, j^*, \delta^*) \leftarrow$ sample from $best$ with $P(\ell) \propto w_\ell$
19:     end if
20:     $V_F \leftarrow V_F \cup \{j^*\}$; $E_F \leftarrow E_F \cup \{(i^*, j^*, \delta^*)\}$
21:     $p_F(j^*) \leftarrow p_F(i^*) + \delta^*$
22:     Update $canvas$ with cells of $T_{j^*}$ at $p_F(j^*)$
23: end while
24: return $(F, p_F)$
```

Each individual in the initial population is obtained by running a greedy constructive heuristic from a given start string, producing a full placement with absolute coordinates; we then extract a spanning tree of relative offsets. Thus every individual faithfully encodes the relative structure of a greedy solution.

We consider two variants that differ in their use of deterministic versus stochastic greedy:

- **T-GA** (Tree-based GA): Uses *deterministic* tree-growing greedy (DET mode) for both population initialization and greedy completion. Since deterministic greedy produces the same tree for a given root, initial population members differ *only* in their choice of starting root. This limits initial diversity to n distinct individuals.

- **ST-GA** (Stochastic Tree-based GA): Uses *stochastic* tree-growing greedy (STOCH mode) for both population initialization and greedy completion. Each initial individual is generated by an independent stochastic run, introducing diversity at the population level. Greedy completion during crossover is also stochastic, allowing different repair trajectories for incomplete offspring.

Our *locality-preserving crossover* operator combines two parent trees by *alternating* their local tree structures while maintaining geometric feasibility. This design directly implements the building block hypothesis: rather than mixing raw coordinates, we transplant entire subtrees, thereby preserving the *schema*, the pattern of relative offsets that makes a cluster of strings fit together well. Starting from the root, we expand a child tree by re-using parent edges whenever they can be realized without conflicts on the canvas. If some strings cannot be connected using parent edges alone, we perform a final *greedy completion* step to attach all remaining strings. The greedy completion ensures that every crossover produces a complete tree containing all strings, even when parental structures are incompatible. We additionally keep track of how many strings had to be attached via greedy completion, which serves as a diagnostic of how often the parental structures alone suffice.

The full GA is summarized in Algorithm 3, and the crossover operator in Algorithm 4. We maintain a population of trees, initialized from greedy placements with different starting strings (T-GA) or independent stochastic runs (ST-GA).

Algorithm 3 Tree-Based Genetic Algorithm for 2D-SSP

```
Require: Strings $\mathcal{T}$, population size $N$, generations $G$, crossover rate $\rho$, elite fraction $\epsilon$, mode $\in \{\textsc{Det}, \textsc{Stoch}\}$
Ensure: Best placement tree $F^*$
 1: // Initialization
 2: for $k = 1$ to $N$ do
 3:     $r_k \leftarrow$ random root from $\{1, \dots, n\}$
 4:     $P[k] \leftarrow$ TreeGrowingGreedy($\mathcal{T}, r_k$, mode)        ▷ Det for T-GA, Stoch for ST-GA
 5: end for
 6: // Main loop
 7: for $g = 1$ to $G$ do
 8:     Evaluate fitness $f[k] \leftarrow \mathrm{cost}(P[k])$ for all $k$
 9:     Sort population by fitness (ascending)
10:     $P' \leftarrow$ copy of top $\lfloor \epsilon N \rfloor$ individuals        ▷ Elitism
11:     while $|P'| < N$ do
12:         if rand() $< \rho$ then
13:             Select parents $F_1, F_2$ via tournament selection
14:             $F_{\text{child}} \leftarrow$ TreeCrossover($F_1, F_2$, mode)
15:         else
16:             $F_{\text{child}} \leftarrow$ copy of tournament-selected individual
17:         end if
18:         $P' \leftarrow P' \cup \{F_{\text{child}}\}$
19:     end while
20:     $P \leftarrow P'$
21: end for
22: return $\arg\min_{F \in P} \mathrm{cost}(F)$
```

In each generation we decode all individuals, rank them by fitness, copy the best few (elitism (De Jong, 1975)), and fill the remaining slots by crossover or by copying fit parents.

Our GA deliberately omits a dedicated mutation operator. This design choice is motivated by two considerations. First, when crossover is carefully designed to recombine meaningful building blocks, as our subtree-based crossover does, explicit mutation often provides diminishing returns or can even be counterproductive by disrupting well-structured solutions. This observation aligns with findings in other combinatorial optimization domains where problem-specific crossover operators dominate the search dynamics (Sholomon et al., 2013). Second, maintaining feasibility under traditional mutation is non-trivial in our setting: a random perturbation of edge offsets in a placement tree can easily introduce symbol conflicts or break connectivity.

However, we observe that the *greedy completion* step in our crossover operator (Algorithm 4, lines 18–22) implicitly serves as a *feasibility-preserving mutation mechanism*. When crossover produces an incomplete offspring, one that does not contain all strings, the greedy completion step attaches the missing strings using fresh, locally-optimal placements that differ from both parents. In the stochastic variant (GA (Stochastic)), this completion step uses *stochastic greedy* placement, which randomly samples among equally-good candidate positions rather than deterministically choosing one. This randomization introduces genuine variation: even when the same set of strings must be completed, different runs may attach them at different positions, exploring alternative regions of the solution space.

This design offers several advantages over traditional mutation:

1. *Guaranteed feasibility.* Unlike random perturbations of tree edges, greedy completion always produces symbol-consistent placements by construction.
2. *Adaptive intensity.* The amount of "mutation" adapts to the compatibility of the parents: when parent structures are highly compatible, few strings require completion and offspring closely resemble parents; when parents are incompatible, many strings are completed afresh, introducing substantial variation.
3. *Local optimization.* Newly attached strings are placed greedily, ensuring that the mutated portion of the solution is locally reasonable rather than random.

Algorithm 4 Locality-Preserving Tree Crossover with Greedy Completion

```
Require: Parent trees $F_1 = (V, E_1, r_1)$, $F_2 = (V, E_2, r_2)$, mode $\in$ {DET, STOCH}
Ensure: Child tree $F_c$
 1: $r_c \leftarrow r_1$                                          ▷ Inherit root from first parent
 2: $V_c \leftarrow \{r_c\}$; $E_c \leftarrow \emptyset$; $p_c(r_c) \leftarrow (0,0)$
 3: $canvas \leftarrow$ cells of $T_{r_c}$ at origin
 4: $Q \leftarrow [r_c]$                                          ▷ Queue for BFS expansion
 5: while $Q \neq \emptyset$ do
 6:     $i \leftarrow Q$.DEQUEUE()
 7:     for each child $j$ of $i$ in $F_1$ or $F_2$ with $j \notin V_c$ do
 8:         $\delta \leftarrow$ offset of edge $(i, j)$ in the parent tree
 9:         if attaching $T_j$ at $p_c(i) + \delta$ is symbol-consistent with $canvas$ then
10:             $V_c \leftarrow V_c \cup \{j\}$; $E_c \leftarrow E_c \cup \{(i, j, \delta)\}$
11:             $p_c(j) \leftarrow p_c(i) + \delta$
12:             Update $canvas$; $Q$.ENQUEUE($j$)
13:         end if
14:     end for
15: end while
16: // Greedy completion for missing strings
17: if $|V_c| < n$ then
18:     $missing \leftarrow V \setminus V_c$
19:     for each $j \in missing$ do
20:         $(i^*, \delta^*) \leftarrow$ best attachment via greedy (mode)          ▷ DET or STOCH
21:         $V_c \leftarrow V_c \cup \{j\}$; $E_c \leftarrow E_c \cup \{(i^*, j, \delta^*)\}$
22:         $p_c(j) \leftarrow p_c(i^*) + \delta^*$; update $canvas$
23:     end for
24: end if
25: return $F_c = (V_c, E_c, r_c)$
```

Remark 17 (Population diversity and the super-individual problem)**.** A legitimate concern with heavy repair operators is that they can create *super-individuals*—offspring that dominate the population and cause premature convergence to local optima (De Jong, 1975). This risk is particularly acute when the repair operator imposes a strong bias (e.g., always placing strings in the same canonical positions). We address this concern with both theoretical and empirical arguments.

Theoretical mitigation. Our greedy completion is *context-sensitive*: the placement of each missing string depends on the *current* canvas state, which varies across offspring because different parental subtrees leave different strings already placed. Thus, even when two offspring require the same set of strings to be completed, they typically receive different greedy placements because the existing partial solutions differ. In the stochastic variant, ties are broken randomly, further diversifying outcomes.

Empirical evidence. Table 5 and Figure 3 provide direct evidence against super-individual formation:

1. *Consistent offspring structure.* The direct placement rate ρ_{direct} (share of strings placed by crossover alone) remains remarkably stable across instance sizes, hovering between 95% and 97.5% for both GA variants. This consistency indicates that crossover—not the repair operator—dominates solution construction, and that offspring inherit diverse parental structures rather than collapsing to a uniform "greedy template."
2. *Moderate and stable repair intensity.* The repair rate r_{repair} (fraction of crossovers needing completion) ranges from 3% to 13%, remaining bounded even as instances scale from 6 to 100 strings. If super-individuals were forming, we would expect r_{repair} to collapse toward zero as the population converges to nearly identical trees that always produce complete offspring.
3. *Variance in greedy share.* The per-instance ρ_{greedy} values in Figure 3 scatter between 2% and 7% rather than clustering at a single value, indicating that different runs (and different instances) exercise the repair mechanism to varying degrees—a signature of genuine population diversity.

These observations confirm that the tree representation supports diverse topologies: crossover combines structurally distinct parental subtrees, and the lightweight greedy completion fills gaps without homogenizing the population. The GA converges toward good solutions through progressive refinement of diverse building blocks, not through domination by repair-generated super-individuals.

Our experimental analysis (Section 5, Table 5) shows that approximately 3–5% of string placements arise from greedy completion, providing a consistent but moderate level of exploration that complements the exploitation performed by crossover.

In all experiments we fix the crossover rate $\rho = 0.7$ and an elite fraction of 10% of the population. Unless otherwise stated, the objective used in selection is the area-based cost of the decoded placement. The same GA can be run with the balanced-area cost by simply changing the fitness function to cost_{bal}, and we report results for both objective variants.

5. Experiments

This section presents our experimental evaluation. We first describe the experimental setup (Section 5.1), then report results comparing algorithms across different problem scales and string geometries (Section 5.2), and finally analyze the internal dynamics of the genetic algorithm (Section 5.3).

5.1. Experimental Setup

Hardware and implementation. All experiments were conducted on a system with an Intel Core i5-13400F processor and 32 GB RAM. The ILP formulation was solved using IBM ILOG CPLEX 22.1. All heuristic algorithms were implemented in Python.

Instance generation. For each configuration, we generated 10 synthetic instances consisting of random binary 2D strings (alphabet $\Sigma = \{0, 1\}$), where each cell is assigned 0 or 1 uniformly at random with probability 0.5. The same 10 instances were used across all algorithms within each configuration to ensure fair comparison.

Table 2 presents results for configurations mixing 1D-like and 2D string shapes with the *area* objective. Table 3 reports results for genuinely 2D strings with the *balanced-area* objective.

Experimental configurations. We designed four experimental configurations to systematically evaluate algorithm performance across different scales and string geometries:

1. **1D-like instances** (`1d`): Strings with extreme aspect ratios (1×2, 1×4, 1×8) that closely resemble classical 1D strings. Instance sizes: $n \in \{10, 20, 50, 100, 200, 300\}$ strings. This configuration uses the *area* objective ($H \cdot W$) and tests whether merge-based greedy retains its 1D effectiveness. GA parameters: population size 150, 300 generations.
2. **Small instances** (`small`): Genuinely 2D strings (3×3, 2×4, 5×5, 4×6) with $n \in \{6, 8, 10\}$ strings. This configuration includes CPLEX as a baseline (time limit: 300 s) to validate heuristic quality against optimal solutions. Uses the *balanced-area* objective ($\max\{H, W\}$). GA parameters: population size 100, 200 generations.
3. **Medium instances** (`medium`): Same string shapes as `small`, but with $n \in \{20, 30, 50\}$ strings. CPLEX is excluded due to computational intractability. Uses the *balanced-area* objective. GA parameters: population size 150, 300 generations.
4. **Large instances** (`large`): Same string shapes, with $n \in \{60, 80, 100\}$ strings for scalability testing. Uses the *balanced-area* objective. GA parameters: population size 200, 400 generations.

Algorithms compared. We evaluate five heuristic algorithms plus an exact solver:

- `CPLEX`: Exact ILP solver (small instances only, 300 s time limit).
- `M-Greedy`: Merge-based greedy (Algorithm 1).
- `T-Greedy`: Tree-growing greedy with deterministic tie-breaking.
- `ST-Greedy`: Tree-growing greedy with stochastic tie-breaking.

Config	`M-Greedy`	`T-Greedy`	`ST-Greedy`	`T-GA`	`ST-GA`
`T6_n3_m3`	48.30 ± 4.50 (0.008s)	43.63 ± 5.17 (0.002s)	44.13 ± 4.75 (**0.002**s)	41.07 ± 3.82 (0.108s)	**40.03 ± 3.60** (0.133s)
`T6_n5_m5`	172.10 ± 14.19 (0.028s)	144.00 ± 3.00 (**0.025**s)	143.00 ± 2.45 (0.027s)	143.00 ± 2.45 (0.491s)	**142.50 ± 2.50** (1.066s)
`T8_n3_m3`	59.40 ± 6.53 (0.014s)	56.23 ± 4.39 (**0.003**s)	57.31 ± 5.12 (0.003s)	53.31 ± 4.01 (0.142s)	**51.46 ± 3.20** (0.183s)
`T8_n5_m5`	223.50 ± 12.34 (0.084s)	195.20 ± 10.02 (0.053s)	196.20 ± 9.43 (**0.053**s)	190.50 ± 2.69 (0.847s)	**190.00 ± 3.16** (2.472s)
`T10_n1_m2`	**4.50 ± 0.50** (0.004s)	4.60 ± 0.66 (0.001s)	4.70 ± 0.78 (**0.001**s)	**4.50 ± 0.50** (0.089s)	**4.50 ± 0.50** (0.075s)
`T10_n1_m4`	14.00 ± 2.10 (0.015s)	13.80 ± 1.99 (**0.001**s)	13.90 ± 2.02 (0.002s)	13.40 ± 1.56 (0.140s)	**13.20 ± 1.54** (0.152s)
`T10_n1_m8`	45.90 ± 4.30 (0.057s)	49.50 ± 3.56 (**0.007**s)	48.20 ± 4.98 (0.007s)	45.30 ± 4.20 (0.288s)	**44.80 ± 4.38** (0.400s)
`T10_n3_m3`	71.80 ± 7.90 (0.019s)	69.20 ± 5.10 (0.005s)	69.10 ± 5.72 (**0.005**s)	62.30 ± 4.80 (0.195s)	**59.00 ± 2.72** (0.266s)
`T10_n5_m5`	277.00 ± 19.61 (0.202s)	240.00 ± 5.92 (**0.089**s)	241.00 ± 5.83 (0.089s)	237.00 ± 5.57 (1.530s)	**236.00 ± 5.39** (4.976s)
`T20_n1_m2`	5.10 ± 0.54 (0.013s)	5.20 ± 0.60 (**0.001**s)	5.10 ± 0.54 (0.001s)	**4.90 ± 0.30** (0.162s)	**4.90 ± 0.30** (0.154s)
`T20_n1_m4`	18.10 ± 1.87 (0.046s)	18.30 ± 1.10 (0.003s)	19.10 ± 1.97 (**0.003**s)	16.80 ± 0.75 (0.331s)	**16.50 ± 0.50** (0.509s)
`T20_n1_m8`	**71.40 ± 6.41** (0.415s)	77.00 ± 7.97 (**0.025**s)	77.30 ± 5.92 (0.045s)	74.00 ± 5.67 (0.880s)	**71.40 ± 6.23** (1.416s)
`T20_n3_m3`	113.40 ± 8.89 (0.130s)	120.30 ± 8.26 (**0.010**s)	122.60 ± 10.76 (0.013s)	104.20 ± 5.06 (1.041s)	**96.10 ± 3.81** (1.628s)
`T20_n5_m5`	518.40 ± 21.12 (3.390s)	470.00 ± 11.40 (0.562s)	467.50 ± 9.01 (**0.552**s)	459.00 ± 8.31 (18.089s)	**452.00 ± 6.00** (32.016s)
`T30_n3_m3`	143.10 ± 12.51 (0.396s)	154.70 ± 9.95 (**0.021**s)	164.80 ± 11.70 (0.024s)	138.40 ± 7.14 (2.821s)	**131.90 ± 7.02** (3.518s)
`T30_n5_m5`	752.10 ± 25.58 (15.051s)	28.40 ± 0.66 (**0.017**s)	29.00 ± 0.45 (0.024s)	**27.10 ± 0.30** (58.713s)	**27.10 ± 0.30** (90.580s)
`T50_n1_m2`	5.50 ± 0.50 (0.148s)	5.20 ± 0.40 (**0.003**s)	5.40 ± 0.49 (0.003s)	**5.00 ± 0.00** (0.653s)	**5.00 ± 0.00** (0.656s)
`T50_n1_m4`	19.30 ± 1.10 (0.626s)	21.10 ± 1.92 (0.004s)	20.90 ± 1.70 (**0.004**s)	18.50 ± 0.67 (1.258s)	**18.40 ± 0.49** (1.716s)
`T50_n1_m8`	**123.60 ± 13.51** (5.470s)	135.10 ± 14.82 (**0.148**s)	136.50 ± 13.03 (0.191s)	131.30 ± 13.84 (7.273s)	127.10 ± 13.40 (6.853s)
`T50_n3_m3`	211.10 ± 16.83 (1.806s)	227.30 ± 14.45 (**0.086**s)	236.60 ± 15.92 (0.097s)	197.90 ± 6.70 (13.097s)	**191.33 ± 9.01** (18.018s)
`T50_n5_m5`	1168.00 ± 16.00 (109.201s)	1127.50 ± 12.50 (**6.604**s)	1122.50 ± 12.50 (6.623s)	1102.50 ± 7.50 (214.734s)	**1100.00 ± 0.00** (311.548s)
`T100_n1_m2`	**5.00 ± 0.00** (1.188s)	5.20 ± 0.40 (**0.004**s)	5.40 ± 0.49 (0.004s)	**5.00 ± 0.00** (1.628s)	**5.00 ± 0.00** (1.619s)
`T100_n1_m4`	20.60 ± 1.43 (4.948s)	20.00 ± 1.10 (0.007s)	21.30 ± 1.55 (**0.006**s)	19.30 ± 0.46 (3.078s)	**19.00 ± 0.00** (4.048s)
`T100_n1_m8`	**178.20 ± 11.41** (37.071s)	198.50 ± 11.72 (0.399s)	196.10 ± 10.15 (**0.380**s)	185.10 ± 8.94 (25.846s)	183.40 ± 9.32 (25.341s)
`T200_n1_m2`	5.33 ± 0.47 (9.418s)	**5.00 ± 0.00** (**0.008**s)	5.33 ± 0.47 (0.009s)	**5.00 ± 0.00** (4.230s)	**5.00 ± 0.00** (4.374s)

Table 2
Objective type: `area`. Each cell shows objective (top) as mean ± std and runtime is mean only (bottom).

- `T-GA`: Genetic algorithm using deterministic `T-Greedy` for both population initialization and greedy completion. Initial population members differ only in the choice of starting root.
- `ST-GA`: Genetic algorithm using stochastic `ST-Greedy` for both population initialization and greedy completion. This introduces diversity at both stages: each initial individual is generated by an independent stochastic greedy run, and incomplete offspring are completed stochastically.

Config	`M-Greedy`	`T-Greedy`	`ST-Greedy`	`T-GA`	`ST-GA`
T6_n3_m3	7.70 ± 0.90 (0.007s)	7.33 ± 0.54 (**0.001**s)	7.47 ± 0.56 (**0.001**s)	7.00 ± 0.37 (0.165s)	$\mathbf{6.77 \pm 0.42}$ (0.225s)
T6_n5_m5	15.30 ± 1.27 (0.030s)	14.90 ± 1.04 (**0.001**s)	14.80 ± 0.60 (0.001s)	13.50 ± 0.50 (0.476s)	$\mathbf{13.10 \pm 0.30}$ (1.376s)
T6_n10_m10	33.00 ± 1.73 (0.681s)	31.80 ± 1.25 (**0.009**s)	31.10 ± 1.22 (0.009s)	29.60 ± 0.66 (5.311s)	$\mathbf{28.90 \pm 0.30}$ (14.190s)
T8_n3_m3	8.60 ± 0.92 (0.013s)	8.23 ± 0.56 (**0.001**s)	8.40 ± 0.55 (0.001s)	7.77 ± 0.42 (0.304s)	$\mathbf{7.37 \pm 0.48}$ (0.285s)
T8_n5_m5	17.40 ± 1.28 (0.088s)	16.30 ± 0.64 (**0.002**s)	16.60 ± 0.66 (0.002s)	15.10 ± 0.30 (1.561s)	$\mathbf{15.00 \pm 0.00}$ (2.793s)
T8_n10_m10	42.40 ± 3.93 (2.267s)	34.30 ± 1.27 (0.016s)	34.10 ± 1.51 (**0.016**s)	31.70 ± 0.90 (24.422s)	$\mathbf{30.60 \pm 0.49}$ (40.423s)
T10_n3_m3	9.20 ± 0.40 (0.020s)	8.97 ± 0.41 (0.001s)	9.13 ± 0.43 (**0.001**s)	8.17 ± 0.37 (0.237s)	$\mathbf{8.03 \pm 0.18}$ (0.380s)
T10_n5_m5	19.50 ± 2.91 (0.202s)	17.90 ± 0.70 (**0.002**s)	18.20 ± 0.40 (0.003s)	16.70 ± 0.46 (1.786s)	$\mathbf{16.50 \pm 0.50}$ (6.692s)
T10_n10_m10	51.50 ± 5.00 (5.939s)	39.60 ± 1.20 (**0.024**s)	39.50 ± 0.50 (0.027s)	37.10 ± 0.54 (58.876s)	$\mathbf{36.90 \pm 0.30}$ (111.969s)
T20_n3_m3	11.80 ± 0.75 (0.127s)	11.33 ± 0.47 (0.002s)	11.77 ± 0.56 (**0.002**s)	10.27 ± 0.44 (1.280s)	$\mathbf{10.13 \pm 0.34}$ (2.314s)
T20_n5_m5	26.40 ± 3.32 (3.331s)	24.00 ± 0.45 (**0.008**s)	24.00 ± 0.77 (0.010s)	$\mathbf{22.60 \pm 0.49}$ (16.813s)	22.80 ± 0.40 (27.894s)
T20_n10_m10	105.50 ± 2.50 (129.009s)	53.60 ± 1.11 (**0.113**s)	53.40 ± 0.92 (0.135s)	50.30 ± 0.64 (739.766s)	$\mathbf{50.00 \pm 0.00}$ (1088.163s)
T30_n3_m3	13.80 ± 1.72 (0.391s)	12.83 ± 0.52 (**0.003**s)	13.57 ± 0.56 (0.004s)	11.93 ± 0.36 (2.897s)	$\mathbf{11.90 \pm 0.30}$ (5.085s)
T30_n5_m5	30.60 ± 2.76 (15.057s)	28.70 ± 1.00 (**0.019**s)	29.10 ± 0.70 (0.030s)	$\mathbf{27.10 \pm 0.30}$ (75.373s)	27.40 ± 0.49 (108.513s)
T50_n3_m3	16.30 ± 1.19 (1.793s)	15.00 ± 0.52 (**0.006**s)	15.77 ± 0.80 (0.008s)	$\mathbf{14.13 \pm 0.34}$ (12.208s)	14.20 ± 0.48 (18.474s)
T50_n5_m5	35.50 ± 0.50 (109.531s)	35.00 ± 0.00 (**0.055**s)	36.00 ± 0.00 (0.076s)	$\mathbf{33.50 \pm 0.50}$ (236.194s)	35.00 ± 0.00 (358.765s)

Table 3
Objective type: `square`. Each cell shows objective (top) as mean ± std and runtime is mean only (bottom).

All GA variants use tournament selection with size 3, crossover probability 0.9, and mutation probability 0.1.

Performance metrics. For each algorithm and configuration, we report the objective value (mean ± standard deviation over 10 runs) and mean runtime. Best results per configuration are shown in bold.

5.2. Results

Key observations from Table 2 (area objective). *(i) Genuinely 2D instances favor tree-based methods.* On square and near-square string shapes (3×3, 5×5), the tree-based methods (`T-Greedy`, `ST-Greedy`, `T-GA`, `ST-GA`) consistently outperform merge-based greedy. For example, on `T20_n3_m3`, `ST-GA` achieves 96.10 versus `M-Greedy`'s 113.40—a 15% improvement. The advantage grows with instance size: on `T50_n3_m3`, the gap widens to 9% (191.33 vs. 211.10).

(ii) 1D-like instances favor merge-based greedy. On highly elongated strings (1×8), merge-based greedy matches or outperforms tree-based methods, confirming our hypothesis from Section 4.2. For `T100_n1_m8`, `M-Greedy` achieves 178.20 while `ST-GA` obtains 183.40. This validates merge-greedy as an appropriate baseline: it excels precisely in the regime where classical 1D-SSP theory applies.

(iii) GA variants provide consistent improvement over greedy. Across all 2D configurations, `T-GA` and `ST-GA` improve upon their greedy counterparts by 6–12%. The stochastic variant `ST-GA` typically achieves the best objective values, suggesting that tie-breaking randomization aids exploration.

Key observations from Table 3 (balanced-area objective). *(iv) Larger improvements on balanced-area.* The tree-based methods show even stronger relative performance under the balanced-area objective. On `T20_n10_m10`, `ST-GA`

Config	`CPLEX`	`M-Greedy`	`T-Greedy`	`ST-Greedy`	`T-GA`	`ST-GA`
T6_n3_m3	$\mathbf{6.60 \pm 0.49}$ (1.194s)	7.70 ± 0.90 (0.007s)	7.33 ± 0.54 (**0.001**s)	7.47 ± 0.56 (**0.001**s)	7.00 ± 0.37 (0.165s)	6.77 ± 0.42 (0.225s)
T8_n3_m3	$\mathbf{7.30 \pm 0.46}$ (18.737s)	8.60 ± 0.92 (0.013s)	8.23 ± 0.56 (**0.001**s)	8.40 ± 0.55 (0.001s)	7.77 ± 0.42 (0.304s)	7.37 ± 0.48 (0.285s)
T10_n3_m3	$\mathbf{8.00 \pm 0.00}$ (127.740s)	9.20 ± 0.40 (0.020s)	8.97 ± 0.41 (0.001s)	9.13 ± 0.43 (**0.001**s)	8.17 ± 0.37 (0.237s)	8.03 ± 0.18 (0.380s)
T20_n3_m3	11.00 ± 0.53 (201.079s)	11.80 ± 0.75 (0.127s)	11.33 ± 0.47 (0.002s)	11.77 ± 0.56 (**0.002**s)	10.27 ± 0.44 (1.280s)	$\mathbf{10.13 \pm 0.34}$ (2.314s)
T30_n3_m3	14.00 ± 0.00 (203.087s)	13.80 ± 1.72 (0.391s)	12.83 ± 0.52 (**0.003**s)	13.57 ± 0.56 (0.004s)	11.93 ± 0.36 (2.897s)	$\mathbf{11.90 \pm 0.30}$ (5.085s)

Table 4
Objective type: `square`. Each cell shows objective (top) as mean ± std and runtime is mean only (bottom).

achieves 50.00 versus `M-Greedy`'s 105.50—a reduction of more than 52%. This dramatic gap suggests that merge-greedy's overlap-maximization strategy is particularly ill-suited for the balanced-area objective, where controlling aspect ratio is critical.

(v) Scalability. All heuristics scale to $n = 50$ strings on 5×5 shapes, though GA runtimes grow substantially (up to ∼230 s for `T-GA` on `T50_n5_m5`). Greedy methods remain fast (under 1 s) across all configurations.

Comparison with CPLEX (Table 4). Table 4 compares heuristics against CPLEX on small instances where optimal solutions are tractable. The GA variants closely approach CPLEX results, with `ST-GA` achieving gaps under 3% on most configurations. This validates the GA's effectiveness: despite searching a heuristically-restricted subspace, it reaches near-optimal solutions on instances where optimality is verifiable.

Runtime trade-offs. Greedy methods complete in milliseconds to seconds, while GA variants require seconds to minutes. For instance, on `T50_n5_m5` (area objective), `T-Greedy` runs in 6.6 s while `ST-GA` requires 311.5 s—a 47× slowdown for a 2.4% objective improvement (1100 vs. 1127.5). This trade-off is acceptable when solution quality is paramount.

5.3. Genetic Algorithm Dynamics

5.3.1. Crossover statistics

To better understand how the genetic algorithms construct their solutions, we instrument GA and GA (STOCHASTIC) with additional counters. For each run we record:

- the total number of crossover operations, C (`total_crossovers`);
- the number of crossovers that produce an incomplete placement and therefore require greedy completion, C_{repair} (`crossovers_needing_completion`);
- the total number of strings that are finally placed by the greedy completion step across all incomplete offspring, T_{fix} (`total_strings_completed`).

From these quantities we derive:

$r_{\text{repair}} = \dfrac{C_{\text{repair}}}{C}$	repair rate: fraction of crossovers that need greedy fix
$\rho_{\text{greedy}} = \dfrac{T_{\text{fix}}}{Cn}$	share of string placements coming from greedy completion
$\rho_{\text{direct}} = 1 - \rho_{\text{greedy}}$	"completion rate": share of strings placed directly by crossover
$\bar{t}_{\text{repair}} = \dfrac{T_{\text{fix}}}{C_{\text{repair}}}$	average #missing strings per repaired offspring .

Several trends are apparent:

Algorithm	Scale	Mean #crossovers	Repair rate r_{repair}	Direct strings ρ_{direct}	Avg. strings/repair $\bar{t}_{\text{repair}}$
GA	small	3 152	9.2%	97.0%	3.0
GA	medium	12 594	6.7%	97.0%	22.6
GA	large	28 357	3.1%	97.5%	68.3
GA (Stochastic)	small	3 151	12.8%	95.9%	2.7
GA (Stochastic)	medium	12 602	8.9%	95.2%	22.0
GA (Stochastic)	large	28 354	6.3%	96.1%	60.8

Table 5
Internal statistics of the genetic algorithms. "Direct strings" is the fraction of string placements coming directly from crossover ($\rho_{\text{direct}} = 1 - \rho_{\text{greedy}}$); the remainder are filled by the greedy completion procedure. All experiments here use the area-based objective.

- The number of crossovers per run grows with instance size. On small instances each GA performs about 3.1×10^3 crossovers; this increases to roughly 1.26×10^4 on medium instances and 2.8×10^4 on large instances.
- The repair rate r_{repair} is modest: between 3% and 13% of crossovers produce offspring that are not complete placements. The GA (Stochastic) variant tends to trigger repairs slightly more often than GA.
- At the string level, only a very small fraction of the solution is delegated to the greedy completion phase. Across all scales and both GA variants, the mean ρ_{greedy} is about 3.6%, so approximately 96% of string placements come directly from crossover. In other words, the GA's recombination operators are doing almost all of the constructive work.
- When a repair is needed, it can be substantial on large instances: on small instances, an incomplete offspring is missing on average 3 strings out of 8; on medium instances, around 22 out of 33 strings; and on large instances, around 61–68 out of roughly 80 strings. This reflects the increasing difficulty of producing fully consistent placements purely by recombination when the search space grows.

Figure 3 focuses on the string-level interaction between crossover and greedy repair by plotting ρ_{greedy} (the fraction of strings placed by the greedy completion step) as a function of the number of strings. For both GA variants this fraction remains between roughly 2% and 7% across all sizes, confirming that the vast majority of strings in the final placements are produced by crossover rather than by the repair heuristic.

Taken together, these diagnostics indicate that the GA operates in a regime where (i) crossover is the dominant constructive mechanism, responsible for more than 95% of string placements, and (ii) greedy completion acts as a lightweight but essential repair operator that corrects the relatively small fraction of offspring that violate feasibility, especially on large instances where missing strings can be numerous.

6. Conclusion

This paper has introduced and systematically investigated the Two-Dimensional Shortest Superstring Problem (2D-SSP), establishing it as a rich combinatorial optimization problem that unifies classical string sequencing with geometric packing. Our work makes three principal contributions that together advance both the theoretical understanding and practical solvability of this problem class.

Theoretical foundations. We have provided the first formal complexity analysis of 2D-SSP, establishing NP-hardness and APX-hardness for both the area and balanced-area objectives via approximation-preserving reductions from the classical 1D-SSP. More significantly, we have developed a *Discrete Combinatorial Reduction* that transforms 2D-SSP from a geometric optimization problem over an infinite Cartesian coordinate space into a finite combinatorial problem over spanning trees. The *Connectivity Principle* (Lemma 3) proves that optimal solutions can always be made 4-connected without increasing cost—extending classical VLSI compaction arguments to the novel regime of symbol-consistent overlaps. Combined with the *Optimality-Preserving Equivalence* (Theorem 6), these results establish that the search space can be reduced to spanning trees of the placement graph with bounded edge labels. This reduction is the theoretical linchpin enabling both our ILP formulation and metaheuristic approach.

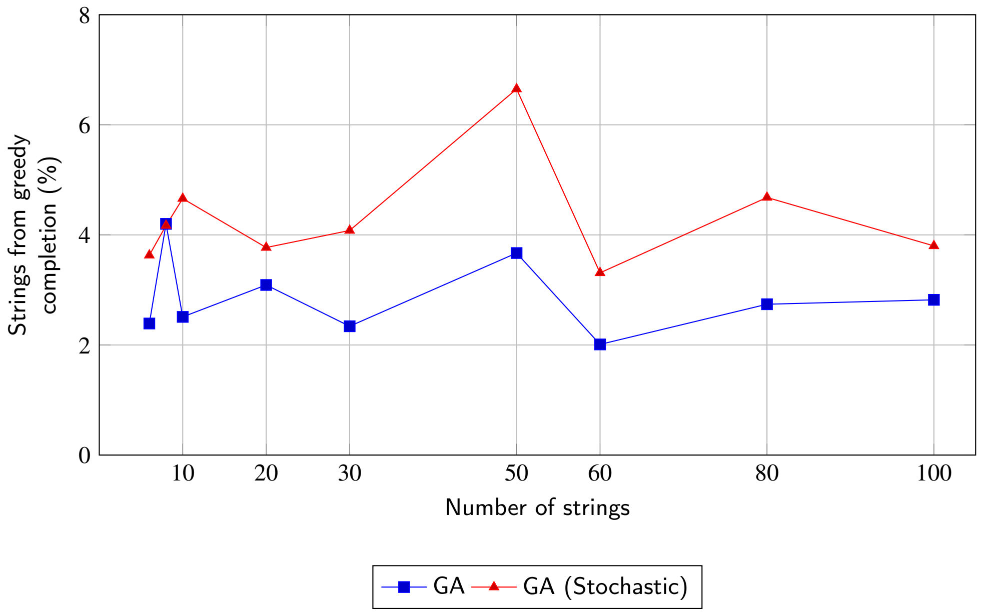

Figure 3: Share of string placements produced by the greedy completion step as a function of the number of strings. The complement to 100% can be interpreted as the "completion rate" of the crossover operators.

Algorithmic innovation. Building on the tree-based representation, we developed a Tree-Based Genetic Algorithm (T-GA) featuring a *locality-preserving crossover* operator that directly implements the building block hypothesis. By operating on subtrees rather than raw coordinates, our crossover preserves the *schema*—the pattern of relative offsets—that makes clusters of strings fit together well. Empirical analysis confirms that crossover dominates solution construction (responsible for $> 95\%$ of string placements), while greedy completion provides lightweight, context-sensitive repair that maintains population diversity without inducing premature convergence. The T-GA matches ILP-optimal solutions on small instances and outperforms greedy heuristics by 6–12% on larger instances, effectively bridging the gap between fast but suboptimal heuristics and optimal but non-scalable exact methods.

Broader implications. Our results suggest that 2D-SSP occupies a distinctive position in the landscape of combinatorial optimization: it inherits the sequencing structure of 1D-SSP (exploited by our tree encoding) while introducing geometric packing constraints (handled by symbol-consistency verification on the canvas). The problem thus serves as a natural testbed for understanding how classical stringology techniques can be extended to higher-dimensional settings. Importantly, our theoretical framework—the Connectivity Principle, the Optimality-Preserving Equivalence, and the tree-based representation—relies solely on translation operations and adjacency relations, with no dependence on the specific dimensionality of the lattice. This means the entire framework extends directly to d-dimensional SSP for any $d \geq 1$, with $2d$-adjacency replacing 4-adjacency. The placement graph formalism and locality-preserving crossover may find application in related problems involving 2D patterns, such as texture synthesis, 2D barcode design, and structured-light pattern generation, as well as their higher-dimensional analogs in 3D printing, volumetric data compression, and crystal structure prediction.

Limitations and future directions. Several limitations of the current work merit acknowledgment. First, our ILP formulation does not scale beyond small instances ($n \leq 10$), limiting its role to validation rather than practical optimization. Second, while the Connectivity Principle guarantees lossless restriction to connected placements, the gap between Theorem 6's existence proof and the GA's greedy-completion subspace remains theoretically open. Third, our experimental evaluation focuses on identical size strings over binary alphabets; behavior on larger, irregularly shaped set of strings or higher-entropy alphabets is unexplored.

These limitations suggest several directions for future research:

- *Approximation algorithms.* Can the classical 2.5-approximation for 1D-SSP be generalized to 2D? What constant-factor guarantees are achievable, and how do they depend on string shape and alphabet size?
- *Tighter inapproximability.* While APX-hardness rules out a PTAS, the precise approximation threshold for 2D-SSP remains open. Establishing whether 2D-SSP admits better or worse approximation than 1D-SSP would clarify the computational landscape.
- *Alternative objectives.* Beyond bounding-box cost, one could minimize the union area $|R(p)|$ or weighted combinations of area and perimeter, each inducing different algorithmic challenges.
- *Scaling to larger instances.* Developing decomposition methods, column-generation ILP approaches, or hybrid exact-metaheuristic strategies could extend exact solvability to medium-scale instances.
- *Non-rectangular strings.* Extending the formalism to handle L-shaped, T-shaped, or arbitrarily shaped 2D patterns would broaden applicability to real-world fabrication and design problems.

In conclusion, we have established 2D-SSP as a well-defined, theoretically grounded optimization problem with practical relevance and rich structure. The discrete combinatorial reduction, tree-based encoding, and locality-preserving genetic algorithm provide a coherent framework for attacking this and related problems, opening new avenues at the intersection of stringology, computational geometry, and metaheuristic optimization.

References

Bennell, J.A., Oliveira, J.F., 2009. A tutorial in irregular shape packing problems. Journal of the Operational Research Society 60, S93–S105.

Blum, A., Jiang, T., Li, M., Tromp, J., Yannakakis, M., 1994. Linear approximation of shortest superstrings. Journal of the ACM 41, 630–647.

Cazaux, B., Rivals, E., 2018. Hierarchical overlap graph. Information Processing Letters 136, 78–84.

Chang, Y.C., Chang, Y.W., Wu, G.M., Wu, S.W., 2000. B*-trees: a new representation for non-slicing floorplans, in: Proceedings of the 37th Design Automation Conference, pp. 458–463.

Charalampopoulos, P., Pissis, S.P., Radoszewski, J., Waleń, T., Zuba, W., 2021. Computing covers of 2d strings, in: 32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021), Schloss Dagstuhl–Leibniz-Zentrum für Informatik. pp. 12:1–12:20.

De Jong, K.A., 1975. An Analysis of the Behavior of a Class of Genetic Adaptive Systems. Ph.D. thesis. University of Michigan.

Efros, A.A., Freeman, W.T., 2001. Image quilting for texture synthesis and transfer, in: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pp. 341–346.

Gallant, J., Maier, D., Storer, J.A., 1980. On finding minimal length superstrings. Journal of Computer and System Sciences 20, 50–58.

Gilmore, P.C., Gomory, R.E., 1965. Multistage cutting stock problems of two and more dimensions. Operations Research 13, 94–120.

Goldberg, D.E., 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley.

Golomb, S.W., 1994. Polyominoes: Puzzles, Patterns, Problems, and Packings. 2nd ed., Princeton University Press.

Kaplan, H., Shafrir, N., 2005. The greedy algorithm for shortest superstrings. Information Processing Letters 93, 13–17.

Korf, R.E., 2003. Optimal rectangle packing: Initial results , 287–295.

Kwatra, V., Schödl, A., Essa, I., Turk, G., Bobick, A., 2003. Graphcut textures: image and video synthesis using graph cuts. ACM Transactions on Graphics (ToG) 22, 277–286.

Leung, J.Y.T., Tam, T.W., Wong, C.S., Young, G.H., Chin, F.Y.L., 1990. Packing squares into a square. Journal of Parallel and Distributed Computing 10, 271–275.

Lodi, A., Martello, S., Monaci, M., 2002. Two-dimensional packing problems: A survey. European Journal of Operational Research 141, 241–252.

McCormick, G.P., 1976. Computability of global solutions to factorable nonconvex programs: Part i—convex underestimating problems. Mathematical Programming 10, 147–175.

Morabito, R., Morales, S., 1998. A simple and effective recursive procedure for the manufacturer's pallet loading problem. Journal of the Operational Research Society 49, 819–828.

Mucha, M., 2013. Lyndon words and short superstrings, in: Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM. pp. 958–972.

Murata, H., Fujiyoshi, K., Nakatake, S., Kajitani, Y., 1996a. Vlsi module placement based on rectangle-packing by the sequence-pair. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 15, 1518–1524.

Murata, H., Fujiyoshi, K., Nakatake, S., Kajitani, Y., 1996b. Vlsi module placement based on rectangle-packing by the sequence-pair. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 15, 1518–1524.

Papadimitriou, C.H., Yannakakis, M., 1991. Optimization, approximation, and complexity classes. Journal of Computer and System Sciences 43, 425–440.

Sholomon, D., David, O.E., Netanyahu, N.S., 2013. A genetic algorithm-based solver for very large jigsaw puzzles. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition , 1767–1774.

Winfree, E., Liu, F., Wenzler, L.A., Seeman, N.C., 1998. Algorithmic self-assembly of dna. Nature 394, 539–544.

Wolsey, L.A., 1998. Integer Programming. Wiley-Interscience.

Wong, D., Liu, C., 1986. A New Algorithm for Floorplan Design. Design Automation Conference.

Yehezkeally, Y., Schwartz, M., 2025. Constructions of covering sequences and 2d-sequences. Designs, Codes and Cryptography 93, 1–25.