A Graph-Based and Metaheuristic Approach to the 2D Shortest Superstring Problem

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

[a] *VinUniversity, Hanoi, Vietnam*

Abstract

We investigate a two-dimensional generalization of the Shortest Superstring Problem (SSP): given a set of 2D strings (symbol arrays), place them on the integer grid with symbol-consistent overlaps to minimize a bounding-box cost. We study two cost variants: a *square* objective minimizing $\max\{H, W\}$, and an *area* objective minimizing $H \cdot W$.

Our key modelling insight is to represent solutions as trees of relative offsets in a *placement graph*, capturing local adjacency structure. We prove that connected placements correspond bijectively to feasible spanning trees, enabling an ILP formulation for exact solutions on small instances. Building on this structure, we develop a greedy heuristic and a tree-based genetic algorithm (GA) whose crossover preserves coherent local neighborhoods.

Experiments show that the GA matches ILP-optimal solutions on small instances and outperforms greedy by 6–12% on larger instances, while remaining orders of magnitude faster than ILP.

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

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.

[*] Corresponding author

Email addresses: dat.tt3@vinuni.edu.vn (Dat Thanh Tran), 23khai.tq@vinuni.edu.vn (Khai Quang Tran), khu.vv@vinuni.edu.vn (Van Khu Vu)

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 $T_1, \ldots, T_n$ (finite 2D arrays over a finite alphabet), collected into a set $\mathcal{T}$. The 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 natural cost variants:

- a *square objective*, which minimizes the side length $\max\{H, W\}$ of the smallest enclosing square;
- an *area objective*, which minimizes the rectangle area $H \cdot W$.

Our central modelling choice is to represent a solution as a *tree of relative offsets between 2D strings* rather than as an explicit 2D array or a vector of absolute coordinates. This choice is motivated by both theoretical and algorithmic considerations.

From a *theoretical* perspective, we prove that every connected placement corresponds to a spanning tree of a naturally defined contact graph, and conversely, every feasible spanning tree induces a valid placement. This equivalence (Theorem 7, Section 3) shows that trees provide a complete and non-redundant encoding of the solution space.

From an *algorithmic* perspective, the tree representation has a crucial advantage for metaheuristic search: it captures *local structure*. Consider a good partial solution where strings A, B, and C form a tightly packed cluster. In a tree representation, this cluster corresponds to a subtree, and the relative offsets within the subtree encode exactly how these strings fit together. A crossover operator can transplant this entire subtree from one parent to another, preserving the beneficial local arrangement—a principle aligned with the building block hypothesis in genetic algorithms (Goldberg, 1989). 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 will almost certainly destroy the cluster's internal structure, since the absolute positions are meaningful only relative to a specific global arrangement.

This observation suggests that tree-based crossover should produce higher-quality offspring than coordinate-based crossover, a hypothesis confirmed by our experiments: our tree-based 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

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

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 Strings and Covering Structures

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

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.

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.

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, 1996) 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. Problem Definition and Structural Properties

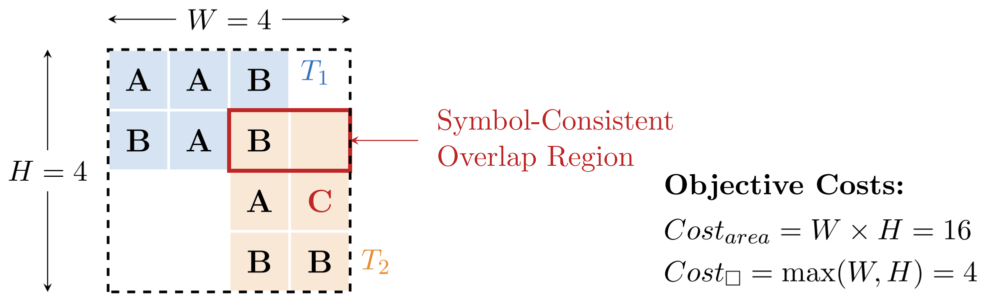
$W = 4$
$H = 4$
A A B
B A B
A C
B B
T_1
T_2
Symbol-Consistent Overlap Region
Objective Costs:
$Cost_{area} = W \times H = 16$
$Cost_{\square} = \max(W, H) = 4$

Figure 1: Illustration of a Symbol-Consistent Placement.

We now formalize the Two-Dimensional Shortest Superstring Problem (2D-SSP), introduce the two objective variants, and develop the structural results underlying our tree-based representation.

3.1. Preliminaries and notation

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 side length of the smallest enclosing square.

3.2. Formal problem definition

Definition 1. Let $\mathcal{T} = \{T_1, \ldots, T_n\}$ 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.,

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

We denote by

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

the *area* and the *square side length* of S, respectively. 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 natural 2D analogue of superstring length (area of the bounding box), while $|S|_{\square}$ focuses on the longest side of the enclosing square.

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 $|S|_{\text{area}}$.
- *Square-based 2D-SSP* (2D-SSP$_{\square}$): find a 2D-superstring S of $\mathcal{T}$ minimizing $|S|_{\square}$.

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

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.

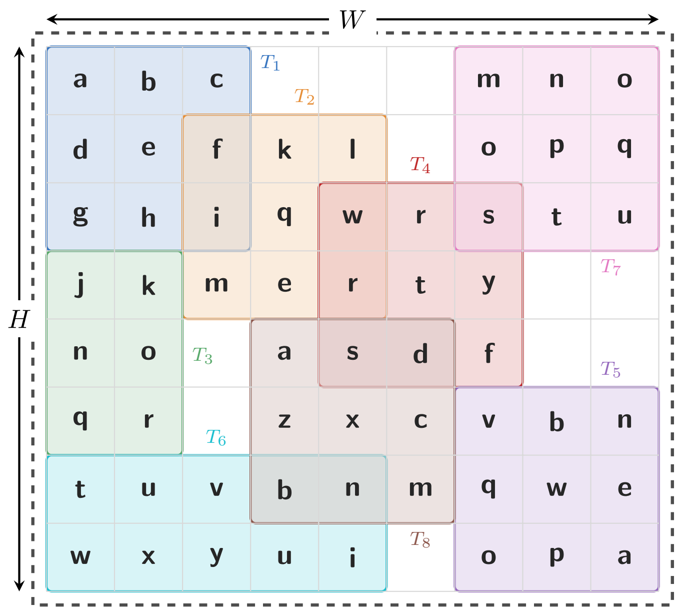

Figure 2: This represents a typical optimal solution where strings cluster to share symbols but extend outwards to fit unique content.

3.3. Placements and symbol consistency

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

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

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

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

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 square side satisfy

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

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$_{\square}$, with cost equal to the chosen bounding-box functional.

Conversely, let S be a 2D-superstring of $\mathcal{T}$ and fix an arbitrary occurrence $(i, j) \in \text{Occ}(T_k, S)$ 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.

3.4. Contact graphs and connectivity

We now focus on connected placements, which is the regime of interest for our applications and structural results; this is formalized in Assumption 1 below.

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 $x = y$.

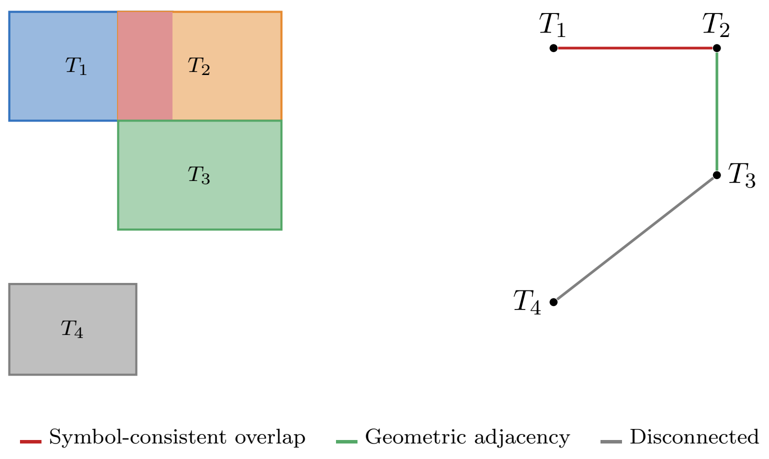

Figure 3: Left: geometric placement. Right: abstract contact graph $G^{ct}(p)$. Nodes correspond to strings; edges indicate symbol-consistent overlap (red) or geometric adjacency (black) or disconnected (gray).

Definition 4. Let p be a symbol-consistent placement of $\mathcal{T}$. The *contact graph* $G^{\mathrm{ct}}(p)$ is the graph with vertex set $\{1, \ldots, n\}$, where two distinct strings i and j are adjacent if either

- T_i and T_j have non-empty symbol-consistent overlap under p, or
- There exist global cells x of T_i and y of T_j with $\|x - y\|_1 = 1$ (that is, x and y are 4-adjacent).

In other words, an edge of $G^{\mathrm{ct}}(p)$ records that T_i and T_j touch in the final arrangement, either by overlapping or by sharing a side.

Remark 1. 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$_{\square}$. This choice mirrors the 1D shortest superstring objective (string length) in the area-based case and a natural "square" version in the other case, and differs from geometric covering formulations that minimize the area of the union $R(p)$.

Assumption 1 (Connectivity of optimal unions)**.** In the structural discussion below we restrict attention to instances for which there exists an optimal symbol-consistent placement p^* (minimizing either $\mathrm{cost}_{\mathrm{area}}$ or $\mathrm{cost}_{\square}$) whose occupied region $R(p^*)$ is 4-connected.

The following lemma shows that this assumption is without loss of generality: any optimal placement that is not 4-connected can be transformed into a 4-connected optimal placement.

Lemma 1. *Let p be an optimal symbol-consistent placement for 2D-SSP (under either* $\mathrm{cost}_{\mathrm{area}}$ *or* $\mathrm{cost}_{\square}$*). 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 $R_1, R_2, \ldots, R_k$. 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 construct p' by translating the components to merge them while ensuring their bounding boxes become adjacent without interpenetration. Without loss of generality, consider merging R_2 into the region containing R_1. Let

$$x_1^{\max} = \max\{x : (x, y) \in B_1 \text{ for some } y\},$$
$$x_2^{\min} = \min\{x : (x, y) \in B_2 \text{ for some } y\}.$$

We choose the translation vector $\tau = (x_1^{\max} - x_2^{\min} + 1, \delta_y)$, where:

- The horizontal component $x_1^{\max} - x_2^{\min} + 1$ is the unique value that places the leftmost column of the translated B_2 immediately to the right of the rightmost column of B_1, ensuring the two bounding boxes are horizontally adjacent but do not overlap.
- The vertical component δ_y is chosen to ensure vertical overlap: if the y-projections of B_1 and B_2 already intersect, we set $\delta_y = 0$; otherwise, we set δ_y to the minimum shift needed to make them share at least one row.

This construction guarantees that the translated B_2 and B_1 share a common edge (or at least a corner), making the combined region 4-connected.

We now verify that:

1. *Symbol-consistency is preserved.* Before translation, strings in $\mathcal{T}_1$ and $\mathcal{T}_2$ had no overlapping cells (since R_1 and R_2 were disjoint). After translation, the bounding boxes B_1 and the translated B_2 are adjacent but non-overlapping by construction of τ. Therefore, no cell can be occupied by both a string from $\mathcal{T}_1$ and a string from $\mathcal{T}_2$, and no symbol conflict arises. Symbol-consistency within each $\mathcal{T}_\ell$ is trivially preserved since internal offsets are unchanged.
2. *The bounding box dimensions do not increase (for either cost variant).* Let $B(p) = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ with width $W = x_{\max} - x_{\min} + 1$ and height $H = y_{\max} - y_{\min} + 1$.
   Key observation: Both cost functions $\text{cost}_{\text{area}}(p) = W \cdot H$ and $\text{cost}_{\square}(p) = \max(W, H)$ are *monotonically increasing* in each dimension. Equivalently, they are monotonically decreasing with respect to the removal of empty space from the bounding box. Since the components $R_1, \ldots, R_k$ are pairwise disjoint and all fit within $B(p)$, there must exist gaps (empty columns or rows) separating them within $B(p)$—otherwise they would not be disconnected. Our translation removes these gaps.
   After merging R_1 and R_2 horizontally, denote the new bounding box dimensions by W' and H'. We have:
   - $W' = W_1 + W_2 \leq W - 1 < W$: the strict inequality holds because B_1 and B_2 originally fit within $[x_{\min}, x_{\max}]$ with at least one empty column between them (the gap that separated the disconnected components).
   - $H' = \max(H_1, H_2) \leq H$: the height of the merged region is the maximum of the two component heights, which cannot exceed the original height. If a vertical shift $\delta_y \neq 0$ was applied, it can only decrease H' further (by aligning the components vertically).

Since $W' < W$ and $H' \leq H$, we have coordinate-wise strict improvement in at least one dimension:

$$\begin{aligned} \text{cost}_{\text{area}}(p') &= W' \cdot H' < W \cdot H = \text{cost}_{\text{area}}(p), \\ \text{cost}_{\square}(p') &= \max(W', H') \leq \max(W, H) = \text{cost}_{\square}(p). \end{aligned}$$

The area inequality is strict; the square inequality may be an equality (if the original max was determined by H and $H' = H$), but in no case does the cost increase.

Repeating this process for all remaining components yields a placement p' with $R(p')$ 4-connected and $\text{cost}(p') \leq \text{cost}(p)$. Since p was optimal, we have $\text{cost}(p') = \text{cost}(p)$, so p' is also optimal. $\square$

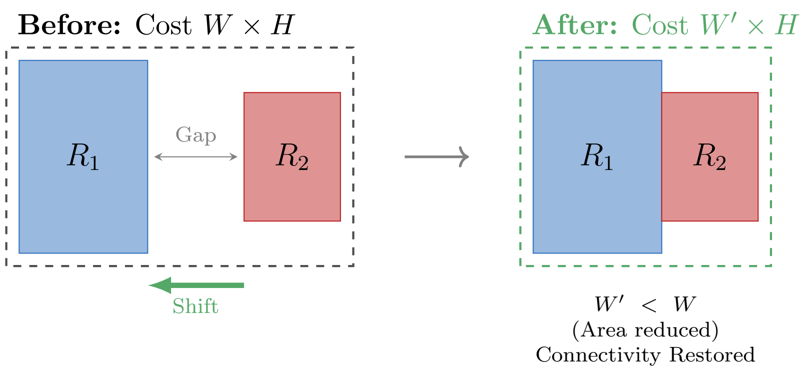

Figure 4: **Merging Logic (Lemma 1).** Any optimal placement with disconnected components (R_1, R_2) can be improved or maintained by shifting components until they touch, eliminating empty gaps and reducing the bounding box.

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

This structural result justifies our focus on connected placements: rather than being a restrictive modelling choice, connectivity is a property that can always be achieved at optimality. 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 3. *Let p be a symbol-consistent placement such that $R(p)$ is 4-connected. Then the contact graph $G^{\text{ct}}(p)$ 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 $R(p)$. □

Under Assumption 1 and Lemma 3, we obtain the following.

Theorem 4. *Let $\mathcal{T}$ be an instance of 2D-SSP (either 2D-SSP$_{\mathrm{area}}$ or 2D-SSP$_{\square}$) that satisfies Assumption 1, and let p^* be an optimal symbol-consistent placement (with respect to the chosen cost) whose occupied region $R(p^*)$ is 4-connected. Then the contact graph $G^{\mathrm{ct}}(p^*)$ is connected and hence admits a spanning tree. Equivalently, there is an ordering of the strings $(i_1, \ldots, i_n)$ such that for every $t > 1$, string i_t is adjacent in $G^{\mathrm{ct}}(p^*)$ to at least one string i_s with $s < t$.*

One can realize p^* by starting from a root string and repeatedly attaching a new string along an edge of this tree.

3.5. Placement graphs and placement trees

For algorithmic purposes it is convenient to encode all symbol-consistent relative offsets between strings in a separate graph.

Definition 5. Given the string set $\mathcal{T}$, the *placement graph* G^{pl} has vertex set $\{1, \ldots, n\}$. 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 $p(j) = p(i) + \delta$ is symbol-consistent with T_i on their overlapping region. In practice, we restrict to offsets that yield non-empty overlap or 4-adjacent contact; multiple offsets may exist for the same unordered pair $\{i, j\}$.

Intuitively, each edge (i, j, δ) in G^{pl} specifies a way of gluing T_j next to T_i. We represent global placements as trees of such relative offsets.

Definition 6. A *placement tree* for $\mathcal{T}$ is a rooted tree $F = (V, E)$ with vertex set $V = \{1, \ldots, n\}$ 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 $\delta_{ji} = -\delta_{ij}$.

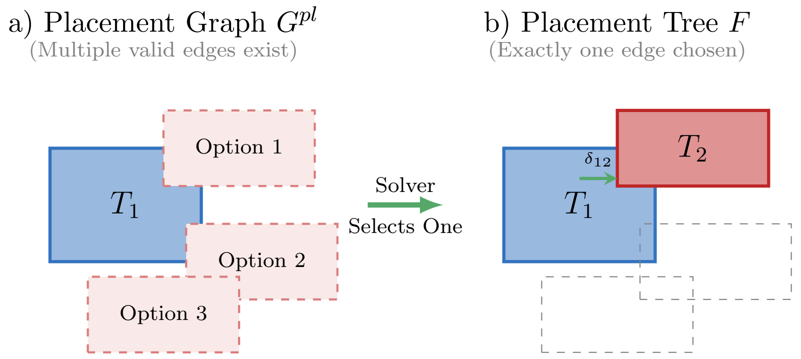

Figure 5: Placement Graph vs. Placement Tree. (a) The graph G^{pl} contains *all* possible valid relative positions (offsets) for T_2 relative to T_1. (b) The tree F selects exactly *one* of these edges to fix the final physical position.

Fix a root $r \in V$. For a vertex $i \in V$, let

$$r = v_0, v_1, \ldots, v_\ell = i$$

be the unique simple path from r to i in F.

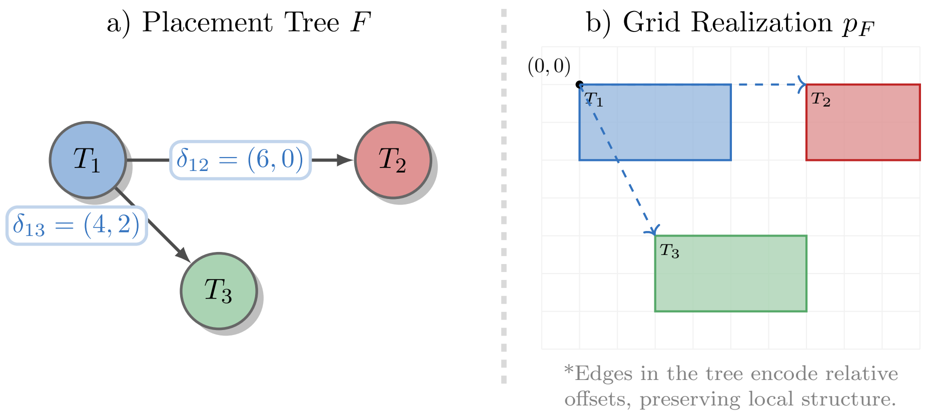

Figure 6: The Bijection between Trees and Placements. (a) A solution represented as a tree. (b) The realization on the grid. The position of T_3 is determined by the vector sum of offsets from the root T_1.

Definition 7. The *realization* of a placement tree F with root position $p(r) \in \mathbb{Z}^2$ is the placement p_F defined by

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

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

We call F *feasible* if, for one (equivalently, for every) choice of root position $p(r)$, the realization p_F is symbol-consistent. In this case we also refer to p_F as a *tree-induced placement.*

Remark 2. Feasibility is a global property of the realized coordinates: 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. We do not attempt to characterize feasibility purely in terms of local constraints on the edge labels (δ_{ij}). In particular, closed walks in the underlying contact graph induce non-trivial "loop-closure" constraints on the offsets, which are difficult to express compactly. Instead, feasibility is treated as an algorithmic property: given a placement tree F, we realize it via p_F and explicitly check symbol-consistency of the resulting placement.

3.6. Structural theorems: equivalence of placements and trees

We now formalize the correspondence between connected placements and feasible placement trees.

Lemma 5. *Let p be a symbol-consistent placement of $\mathcal{T}$ such that $R(p)$ is 4-connected, and let $G^{\mathrm{ct}}(p)$ be its contact graph. Fix any spanning tree F of $G^{\mathrm{ct}}(p)$ and choose a root $r \in V(F)$.*

For each oriented edge (i, j) of F, define

$$\delta_{ij} := p(j) - p(i) \in \mathbb{Z}^2.$$

Then:

1. *The labeled tree F is a feasible placement tree.*
2. *For any choice of $p(r)$, the realization p_F coincides with p up to a global translation.*

Proof. By definition of the labels, for every edge (i, j) the placement p satisfies $p(j) = p(i) + \delta_{ij}$. Consequently, for any vertex i and any path $r = v_0, \dots, v_\ell = i$ in F, we have

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

Thus p is a realization of F for the particular choice of root position $p(r)$. Since p is symbol-consistent, this shows that F is feasible. Changing $p(r)$ to $p(r)+\Delta$ for some $\Delta \in \mathbb{Z}^2$ adds Δ to all positions, yielding a global translation of p. $\square$

Lemma 6. *Let F be a feasible placement tree for $\mathcal{T}$, and let p_F be any realization (for some choice of root and root position). Then p_F is a symbol-consistent placement whose induced array T_{p_F} is a 2D-superstring of $\mathcal{T}$.*

Proof. By feasibility, the realization 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}. Therefore T_{p_F} is a 2D-superstring of $\mathcal{T}$. $\square$

Combining Assumption 1 with Lemmas 3, 5, and 6, we obtain:

Theorem 7. *Let p be a symbol-consistent placement of $\mathcal{T}$ whose occupied region $R(p)$ is 4-connected, and let $G^{\mathrm{ct}}(p)$ be its contact graph. Then any spanning tree F of $G^{\mathrm{ct}}(p)$, equipped with edge labels*

$$\delta_{ij} \;:=\; p(j) - p(i)$$

for each oriented edge (i, j) of F, is a feasible placement tree whose realization p_F coincides with p up to a global translation.

Conversely, any feasible placement tree F induces a symbol-consistent placement p_F and hence a feasible solution to 2D-SSP (under either bounding-box objective), with objective value given by $\mathrm{cost}_{\mathrm{area}}(p_F)$ in the area-based variant or $\mathrm{cost}_{\square}(p_F)$ in the square-based variant.

Remark 3. All lemmas and theorems in this section concern only symbol-consistency, connectivity, and the combinatorial structure of placements. They do not use any specific property of the cost functional beyond the fact that it is derived from the bounding box $B(p)$. Consequently, the structural statements apply verbatim to both 2D-SSP$_{\mathrm{area}}$ and 2D-SSP$_{\square}$.

Theorem 7 establishes that placement trees provide a complete encoding of connected placements, but the theorem's value extends beyond mere equivalence. The tree structure has three 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 greedy construction heuristic (Section 4.2) and ensures that the GA's crossover operator can incrementally build feasible offspring.
3. *Reduced redundancy.* Unlike coordinate-based representations where infinitely many coordinate vectors encode the same placement (differing by global translation), the tree representation is essentially unique for a given contact structure. This reduces the effective search space and avoids wasting computational effort on equivalent solutions.

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.

Table 1: Summary of main notation.

Symbol	Meaning
Σ	Alphabet
$\mathcal{T} = \{T_1, \ldots, T_n\}$	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
$\lvert S\rvert_{\square}$	Side length of minimal enclosing square 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 $W(p)H(p)$
$\mathrm{cost}_{\square}(p)$	Square-based 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 $\{1, \ldots, n\}$
$\delta_{ij} \in \mathbb{Z}^2$	Relative offset from i to j in F
p_F	Realization (tree-induced placement) of F

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 *greedy construction heuristic* (Section 4.2) that builds placements incrementally by always choosing the locally best extension. This approach is extremely fast but may produce suboptimal solutions.
- A *tree-based genetic algorithm* (Section 4.3) 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$_{\square}$.

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 $[0, W_g] \times [0, H_g] \subset \mathbb{Z}^2$, where W_g and H_g are derived from a greedy solution computed as a preprocessing step. Specifically, we first run the greedy heuristic (Section 4.2) to obtain a feasible placement with bounding-box dimensions $W_{\text{greedy}} \times H_{\text{greedy}}$. Since any optimal solution cannot have a larger bounding box than this greedy solution, we set

$$W_g := W_{\text{greedy}}, \qquad H_g := H_{\text{greedy}}.$$

This significantly reduces the number of candidate origins compared to a naive bound of $N \cdot \max_i \max\{w_i, h_i\}$.

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:

$$\mathcal{O}_i := \left\{ o = (x,y) \in \mathbb{Z}^2 \,\middle|\, (x+u, y+v) \in [0, W_g] \times [0, H_g] \text{ for all } (u,v) \in C_i \right\} \tag{1}$$

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

$$\kappa_{ijoo'} = \begin{cases} 1, & \text{if there exist } (u,v) \in C_i,\ (u',v') \in C_j \text{ with} \\ & (x+u, y+v) = (x'+u', y'+v') \text{ and } T_i(u,v) \neq T_j(u',v') \\ & \text{for } o = (x,y),\ o' = (x',y'), \\ 0, & \text{otherwise,} \end{cases} \tag{2}$$

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 $o \in \mathcal{O}_i$:

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

- 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 square side variable (for the square objective):

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

  representing the side length of the smallest enclosing square.

- 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 $W, H, L \in [0, M]$, $A \in [0, M^2]$.

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:

$$\sum_{o \in \mathcal{O}_i} b_{io} \ = \ 1 \qquad \forall i \in \{1, \dots, n\}. \tag{3}$$

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

$$b_{io} + b_{jo'} \ \le \ 1 \quad \forall\, i < j,\ \forall\, o \in \mathcal{O}_i,\ \forall\, o' \in \mathcal{O}_j \text{ with } \kappa_{ijoo'} = 1. \tag{4}$$

(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:

$$X_{\min} \ \le \ x + x_i^{\min} + M(1 - b_{io}), \tag{5}$$
$$X_{\max} \ \ge \ x + x_i^{\max} - M(1 - b_{io}), \tag{6}$$
$$Y_{\min} \ \le \ y + y_i^{\min} + M(1 - b_{io}), \tag{7}$$
$$Y_{\max} \ \ge \ y + y_i^{\max} - M(1 - b_{io}), \tag{8}$$

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

$$W = X_{\max} - X_{\min} + 1, \qquad H = Y_{\max} - Y_{\min} + 1. \tag{9}$$

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

$$W \ \ge \ \max_i w_i, \qquad H \ \ge \ \max_i h_i. \tag{10}$$

4.1.1. Square Objective

The square-based variant minimises the side length of the smallest enclosing square. We link L to W and H via

$$L \;\geq\; W, \qquad L \;\geq\; H. \tag{11}$$

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

The *bounding square* ILP is

$$\begin{aligned} \min \quad & L && (12) \\ \text{s.t.} \quad & (3) - (10),\ (11), \\ & b_{io} \in \{0,1\}\ \forall i, o, \quad X_{\min}, X_{\max}, Y_{\min}, Y_{\max}, W, H, L \in \mathbb{Z}. \end{aligned}$$

4.1.2. Area Objective

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

$$A = W \cdot H. \tag{13}$$

To remain within a ILP 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

$$\begin{aligned} A \;&\geq\; W_{\min}H + H_{\min}W - W_{\min}H_{\min}, && (14) \\ A \;&\geq\; W_{\max}H + H_{\max}W - W_{\max}H_{\max}, && (15) \\ A \;&\leq\; W_{\min}H + H_{\max}W - W_{\min}H_{\max}, && (16) \\ A \;&\leq\; W_{\max}H + H_{\min}W - W_{\max}H_{\min}. && (17) \end{aligned}$$

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

$$\begin{aligned} \min \quad & A && (18) \\ \text{s.t.} \quad & (3) - (10),\ (14) - (17), \\ & b_{io} \in \{0,1\}\ \forall i, o, \quad X_{\min}, X_{\max}, Y_{\min}, Y_{\max}, W, H, A \in \mathbb{Z}. \end{aligned}$$

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 (4) enforce symbol consistency; the big-M constraints (5)–(8) define a global bounding box that contains all chosen placements; and depending on the variant, either (11) or (14)–(17) 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. Greedy Heuristics

We also use greedy construction heuristics that build a placement incrementally on a global *canvas*. Strings are placed one by one, always maintaining symbol-consistency and trying to keep the current bounding-box cost (either side length $\max\{\text{width}, \text{height}\}$ for the square objective or area width $\times$ height for the area objective) as small as possible. All variants start from a chosen "root" string and then attach the remaining strings.

Conceptually, the current solution is represented by:

- a set of occupied global cells with their symbols; and
- the bounding-box coordinates $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$ of these cells.

Given this, the current width and height are

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

and we can define either

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

depending on the chosen objective. For any candidate placement 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 square-based 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.

4.2.1. Deterministic Greedy Placement

The deterministic greedy solver starts from a chosen root string and then iteratively adds the remaining strings. At each step it prefers placements that (i) achieve the smallest possible increase in the chosen bounding-box cost, and (ii) among those, maximize the number of overlapping cells with matching symbols.

By construction, the algorithm always increases the cost threshold until a consistent placement exists for some string, and therefore returns a full placement of all strings.

4.2.2. Stochastic Greedy Placement

The stochastic variant follows the same overall structure, but introduces randomization in the choice among equally good local moves. Instead of picking a single best candidate for the smallest feasible cost increase, it samples one at random from all candidates that achieve the minimal increase.

This stochastic heuristic often yields slightly worse single solutions than the deterministic variant, but it produces a diverse set of layouts, which is useful for initializing the genetic algorithm.

4.3. 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 a finite search window of translations of v relative to u and collect all offsets that yield symbol-consistent overlaps or 4-adjacent contact.

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

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

depending on which objective variant is being optimized.

Each individual in the initial population is obtained by first running a greedy constructive heuristic (deterministic or stochastic) from a given start string, which produces a full placement with absolute coordinates. We then extract a spanning tree of relative offsets from this placement. Thus every individual is a tree that faithfully encodes the relative structure of a greedy solution.

Our crossover operator combines two parent trees by *alternating* their local tree structures while maintaining geometric feasibility. 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 **??**. We maintain a population of trees, initialized from multiple greedy placements with different starting strings and (optionally) stochastic perturbations. 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 **??**, 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.

Our experimental analysis (Section 5, Table 8) 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 square-based cost by simply changing the fitness function to $\text{cost}_{\square}$, and we report results for both objective variants.

5. Experimental Evaluation

Having presented three algorithmic approaches with different design philosophies, we now empirically evaluate their performance on synthetic instances of 2D-SSP. Our experiments address the following questions:

1. How close do the heuristic methods come to optimal solutions on instances where the ILP can be solved?
2. How do the methods scale as instance size increases beyond the ILP's practical limit?
3. What is the trade-off between solution quality and runtime across different methods?

We evaluate under both cost variants introduced in Section 3:

- the *square* objective $f_{\square}(p) := \max\{W(p), H(p)\}$, i.e., the side length of the minimal bounding box, and
- the *area* objective $f_{\text{area}}(p) := W(p) \cdot H(p)$.

Here $W(p)$ and $H(p)$ denote the width and height of the bounding box $B(p)$ induced by placement p.

We compare five algorithms: (i) deterministic greedy (GREEDY), (ii) stochastic greedy (GREEDY (STOCHASTIC)), (iii) the tree-based genetic algorithm initialized from deterministic greedy (GA), (iv) the same GA initialized from stochastic greedy (GA (STOCHASTIC)), and (v) the grid-based ILP solved by CPLEX (CPLEX).

Our primary performance measure is the cost $f(p)$ of the final placement p under the chosen objective (lower is better). We also report the wall-clock runtime in seconds.

5.1. Benchmark instances and protocol

All experiments use 3×3 strings over a fixed finite alphabet. Instance difficulty is controlled by the number of strings N in the input set:

$$N \in \{6, 8, 10, 20, 30, 50, 60, 80, 100\}.$$

For each N we generate multiple random instances and run all algorithms on the same pool of instances.

For the *square objective* we evaluate:

- *small* sets: $N \in \{6, 8, 10\}$, 30 instances per size;
- *medium* sets: $N \in \{20, 30, 50\}$, 20 instances per size;
- *large* sets: $N \in \{60, 80, 100\}$, 10 instances per size.

For the *area objective* we currently report results on *small* sets ($N \in \{6, 8, 10\}$, 10 instances per size); medium and large area-based experiments are left for future work.

For the genetic algorithms we keep the same population size and generation budget across all instance sizes; these settings are chosen such that the GA has time to meaningfully exploit crossover while still remaining far below the CPLEX wall-clock limit (2 000 s in our experiments). For CPLEX we use default parameters with a time limit of 2 000 s per instance; many medium and large instances hit this limit and terminate with a suboptimal incumbent.

All statistics below are averages over successful runs. For each configuration we report the mean and standard deviation of the objective value and runtime.

5.2. Results for the square objective

Tables 2–4 summarize the mean square cost $f_{\square}(p)$ and runtime for each algorithm, aggregated over the small, medium, and large regimes respectively (within each regime we average over all tile counts belonging to that regime).

Algorithm	Mean max side	Mean time [s]
cplex (optimal)	7.32	—
GA (Stochastic)	7.39	0.30
GA	7.64	0.24
greedy	8.18	8.7×10^{-4}
Greedy (Stochastic)	8.33	8.2×10^{-4}

Table 2: Square objective, small instances ($N \in \{6, 8, 10\}$ strings). The ILP provides optimal solutions for verification; its runtime is omitted as it is used only for offline ground-truth generation.

On *small* instances (Table 2), the ILP provides optimal solutions with mean max side 7.32, serving as the ground-truth benchmark. Both genetic algorithms closely approach this optimum (within about 1%–4%). The greedy baselines are essentially instantaneous but incur about 7–13% higher cost on average.

On *medium* instances (Table 3), the ILP is no longer tractable for verification. Both genetic variants achieve mean max side around 12.1, outperforming the greedy heuristics by roughly 7–13%. The GA runtimes (5–9 s) remain practical for these instance sizes.

On *large* instances (Table 4), the tree-based GAs consistently achieve the best max side (around 16.2–16.4), improving on deterministic greedy by about 6% on average. The GA runtimes (tens of seconds) are substantially higher than greedy (tens of milliseconds) but remain practical for offline optimization.

Algorithm	Mean max side	Mean time [s]
GA	12.10	5.08
GA (Stochastic)	12.07	8.73
greedy	13.03	3.3×10^{-3}
Greedy (Stochastic)	13.75	4.2×10^{-3}

Table 3: Square objective, medium instances ($N \in \{20, 30, 50\}$ strings). Mean over all sizes and seeds. ILP verification is not available at this scale.

Algorithm	Mean max side	Mean time [s]
GA	16.20	62.37
GA (Stochastic)	16.40	80.61
greedy	17.20	2.2×10^{-2}
Greedy (Stochastic)	17.97	2.8×10^{-2}

Table 4: Square objective, large instances ($N \in \{60, 80, 100\}$ strings). Mean over all sizes and seeds.

Figure 7 plots the mean square cost as a function of the number of strings, while Figure 8 shows the corresponding runtimes on a logarithmic scale.

These plots highlight a clear Pareto frontier:

- GREEDY and GREEDY (STOCHASTIC) offer the best speed–accuracy trade-off when tight time budgets are required.
- GA and GA (STOCHASTIC) consistently improve the bounding-box side length by roughly 6–12% over greedy once $N \geq 20$, at the cost of two to three orders of magnitude more runtime.
- The ILP (CPLEX) provides verified optimal solutions on small instances ($N \leq 10$) but is computationally intractable beyond this regime.

5.3. Results for the area objective

We now turn to the area-based objective $f_{\text{area}}(p) = W(p) \cdot H(p)$, which is the direct 2D analogue of classical superstring length. Tables 5–7 summarize the average placement cost for each algorithm, grouped by instance scale. Figure 9 visualizes the scaling behavior across all instance sizes.

On small instances (6–10 strings, Table 5), the ILP provides optimal solutions with mean cost 7.32, serving as ground truth. The genetic variants are very close to optimal: GA (STOCHASTIC) and GA are only about 2.3% and 4.2% worse than the optimum, respectively. The purely greedy heuristics sacrifice about 12–14% in solution quality compared to optimal.

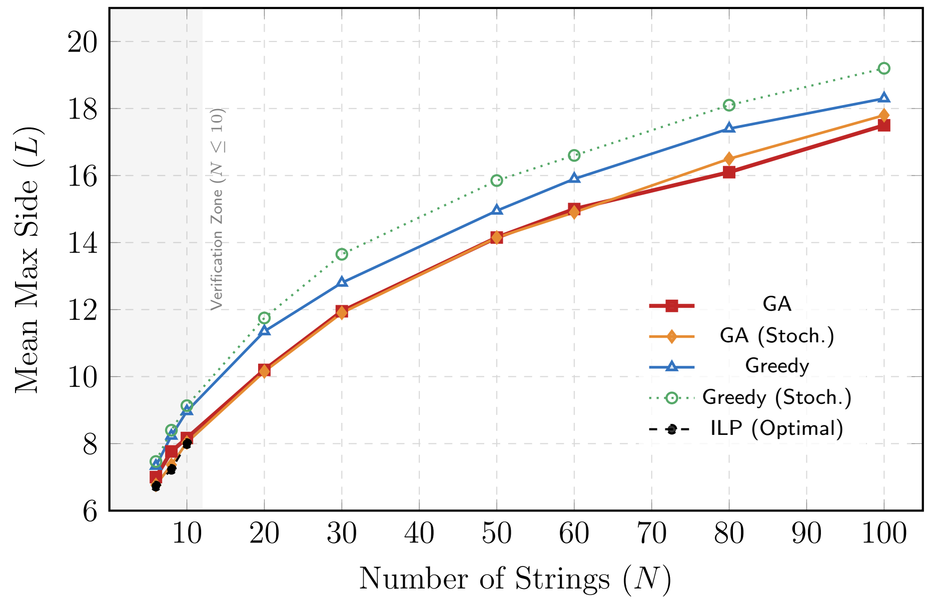

Figure 7: Square Objective: Mean Max Side vs. Number of Strings. The ILP provides optimal ground truth for $N \leq 10$.

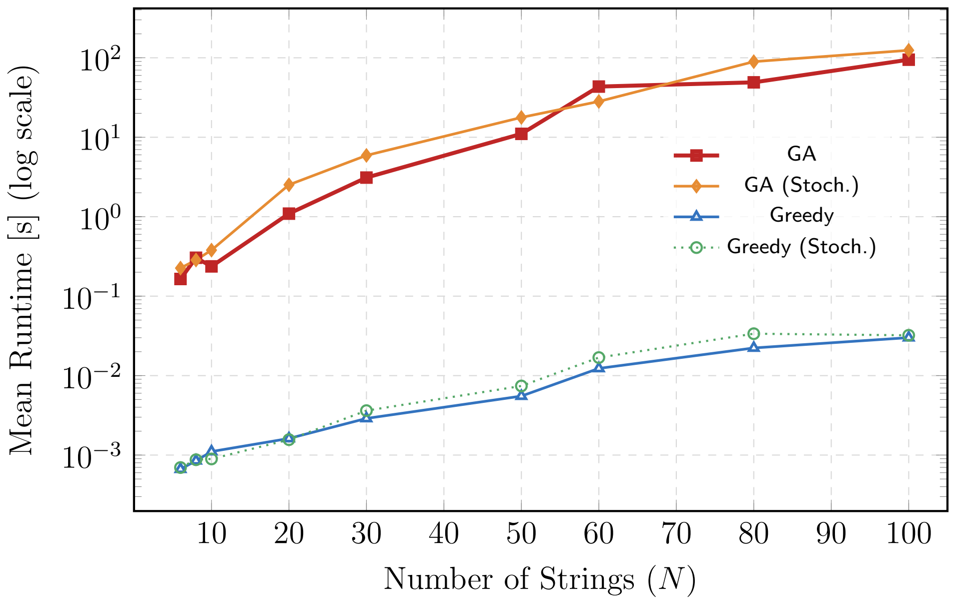

Figure 8: Square objective: mean runtime vs. number of strings (logarithmic scale) for the heuristic methods.

Algorithm	Mean cost	Std. dev.
cplex (optimal)	7.32	0.63
GA	7.63	0.63
GA (Stochastic)	7.49	0.67
greedy	8.24	0.85
Greedy (Stochastic)	8.32	0.91

Table 5: Average placement cost on small string sets (6–10 strings), area-based objective. The ILP provides optimal solutions for verification; runtime is omitted as it is used only for offline ground-truth generation.

Algorithm	Mean cost	Std. dev.
GA	12.17	1.63
GA (Stochastic)	12.16	1.73
greedy	13.17	1.76
Greedy (Stochastic)	13.56	1.77

Table 6: Average placement cost on medium string sets (20–50 strings), area-based objective.

On medium instances (20–50 strings, Table 6), GA (Stochastic) attains the lowest average cost, with GA being essentially indistinguishable (within 0.1%). The greedy baselines are roughly 8% (greedy) and 11.5% (Greedy (Stochastic)) worse in cost.

On large instances (60–100 strings, Table 7), GA consistently achieves the best average cost. The cost gap between the GA and the deterministic greedy baseline is around 6–7% on average, and around 11% for Greedy (Stochastic).

5.4. Scaling with the number of strings

To better understand the scaling behaviour, Figures 9 and 10 plot the mean cost and mean runtime as a function of the number of strings, again using the area-based objective.

Figure 9 shows that for all algorithms the placement cost grows roughly monotonically with the number of strings, as expected. At every size for which optimal solutions are available ($N \leq 10$), the ILP provides the ground truth, and the GA variants remain within 2% of the optimum. For larger instances where the ILP is intractable, the GA consistently achieves the best cost. Both greedy baselines trail the best method by between 6% and 14%,

Algorithm	Mean cost	Std. dev.
GA	16.23	1.09
GA (Stochastic)	16.54	1.21
greedy	17.28	1.25
Greedy (Stochastic)	18.12	1.34

Table 7: Average placement cost on large string sets (60–100 strings), area-based objective.

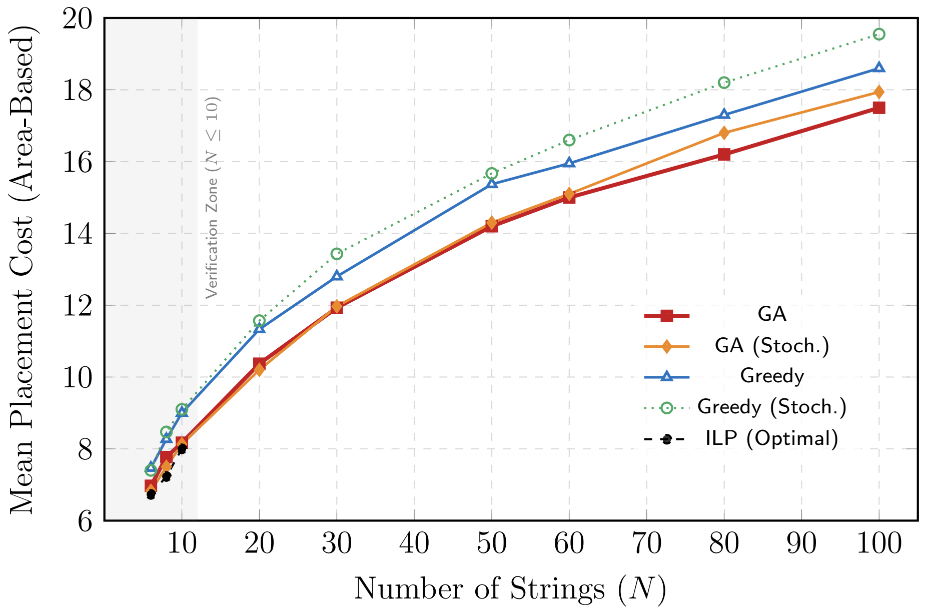

Figure 9: Mean placement cost (area-based objective) as a function of the number of strings. Each point averages over all random instances for the corresponding size.

with an average gap of approximately 7.7% (greedy) and 11.8% (Greedy (Stochastic)) for string sets of size 20 and larger.

Figure 10 (note the logarithmic y-axis) highlights the trade-off between solution quality and runtime for the heuristic methods. The two greedy algorithms are extremely fast and scale almost linearly in the number of strings, remaining below 0.5 seconds for up to 10 strings and below 3.5 seconds even at 100 strings. The GA variants incur one to two orders of magnitude more CPU time, with runtime growing from roughly 4 seconds on 6–10 strings to around 170–225 seconds at 100 strings.

Overall, the experiments suggest a clear Pareto frontier: greedy heuristics provide very fast but moderately suboptimal solutions, while the tree-based genetic algorithms consistently improve the placement cost by roughly 6–12% at the expense of significantly higher runtime. Crucially, when verified against optimal ILP solutions on small instances, the GA achieves near-optimal quality, validating its effectiveness on larger instances where ground truth is unavailable.

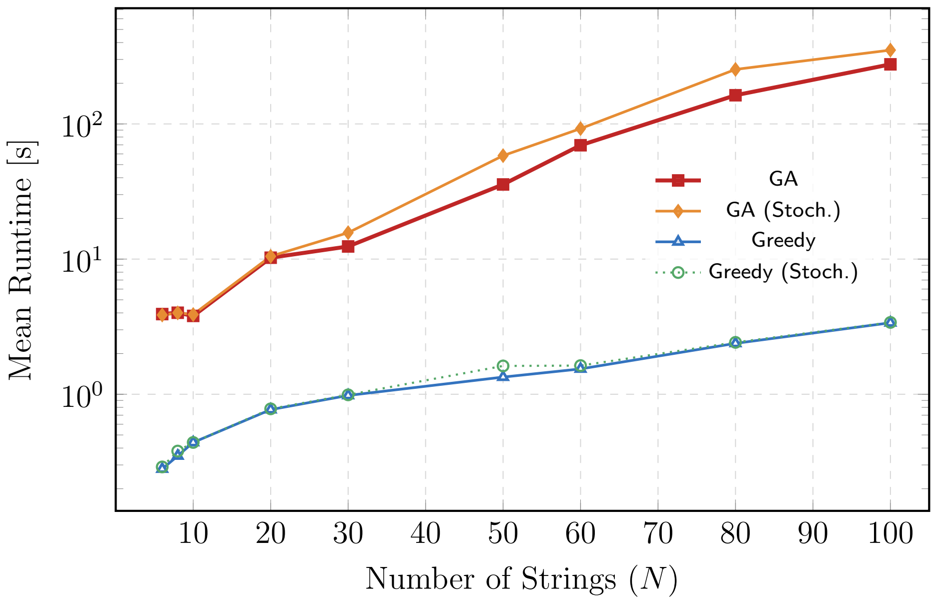

Figure 10: Mean runtime as a function of the number of strings (logarithmic scale) for the heuristic methods. Greedy heuristics are one to two orders of magnitude faster than the genetic algorithms.

5.5. Genetic algorithm dynamics and 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_comp -letion`);
- the total number of strings that are finally placed by the greedy completion step across all incomplete offspring, T_{fix} (`total_strings_complete -d`).

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 8: 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.

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:

- 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 11 plots the average number of crossovers per run as a function of the number of strings. As expected from the fixed population size and generation budget in our implementation, the curves are piecewise constant: roughly 3,150 crossovers for 6–10 strings, 12,600 for 20–50 strings, and 28,350 for 60–100 strings. The two GA variants behave almost identically in this respect.

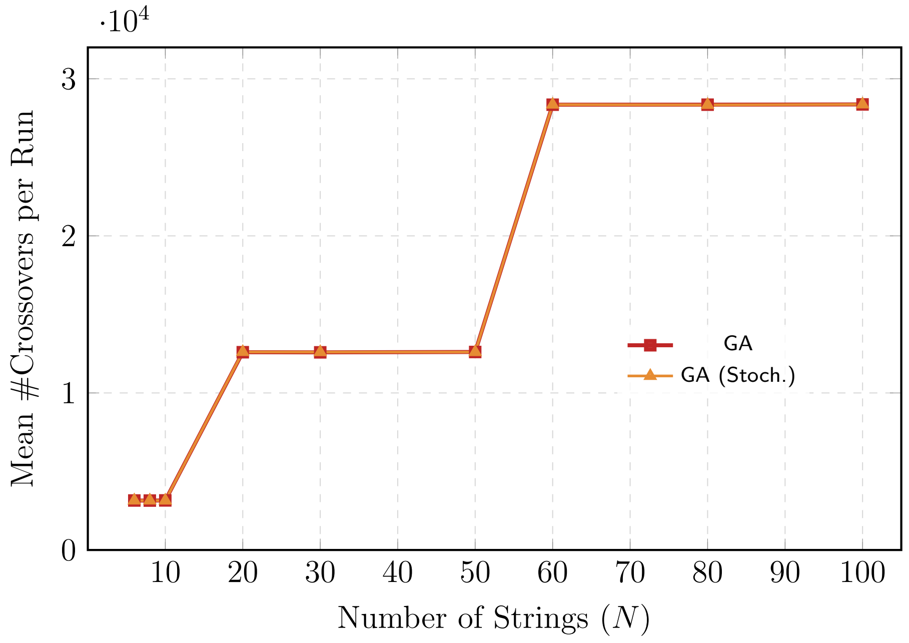

Figure 11: Average number of crossovers per run as a function of the number of strings.

Figure 12 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.

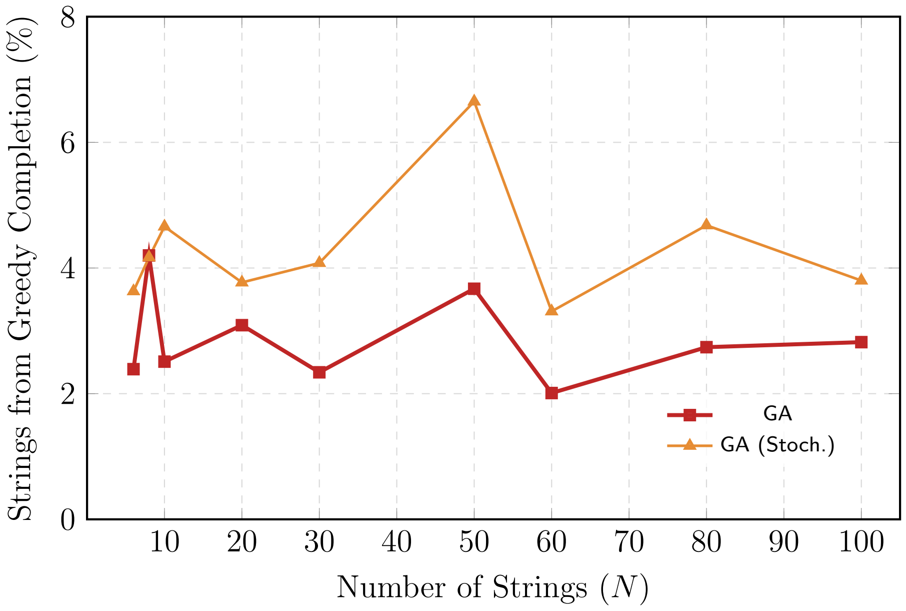

Figure 12: Share of string placements produced by the greedy completion step as a function of the number of strings. The complement to 100% can be interpreted as the completion rate of the crossover operators.

6. Conclusion

We have formulated a two-dimensional version of the Shortest Superstring Problem, in which the goal is to place strings on the integer grid with symbol-consistent overlaps while optimizing a bounding-box objective derived from the minimal axis-aligned rectangle containing all occupied cells. We considered two natural cost variants: a *square* objective, minimizing the side length $\max\{H, W\}$ of the smallest enclosing square, and an *area* objective, minimizing the rectangle area $H \cdot W$.

Our work makes three primary contributions:

1. *Problem formalization.* We introduce the 2D-SSP as a natural two-dimensional generalization of the classical SSP, with two bounding-box cost variants. We establish the equivalence between optimizing over 2D-superstrings and optimizing over symbol-consistent placements.
2. *Structural theory.* We develop a placement-tree representation that encodes solutions as trees of relative offsets rather than raw 2D arrays. Under mild connectivity assumptions, we prove that optimal solutions admit such tree-based representations and that every feasible placement tree induces a valid solution.

3. *Algorithmic framework.* We develop three algorithmic approaches—an exact ILP formulation, a fast greedy heuristic, and a problem-specific tree-based genetic algorithm—that exploit the placement-tree structure. The GA achieves near-optimal quality with substantially lower runtime than ILP and 6–12% better quality than greedy on medium and large instances.

Several limitations of the current work merit discussion. First, our ILP formulation, while exact, does not scale beyond small instances ($n \leq 10$ strings) due to the combinatorial explosion of candidate placements. Second, the connectivity assumption (Assumption 1) excludes instances where optimal solutions consist of disconnected components. Third, our experimental evaluation uses only 3×3 strings over small alphabets; the behavior on larger or more varied string shapes remains unexplored. Finally, the GA's performance depends on parameter choices (population size, crossover rate) that we have not systematically tuned.

Several directions for future research emerge from this work:

- *Approximation algorithms.* Can the greedy 2.5-approximation for 1D-SSP be generalized to the 2D setting? What approximation ratios are achievable for 2D-SSP?
- *Hardness results.* Establishing the precise computational complexity of 2D-SSP (NP-hardness, APX-hardness, or inapproximability bounds) remains open.
- *Alternative objectives.* Beyond bounding-box cost, one could minimize the total area of the union $|R(p)|$ or consider weighted combinations of area and perimeter.
- *Rotations and reflections.* Allowing strings to be rotated or reflected would significantly expand the solution space and potentially improve compression ratios.
- *Applications.* The placement-tree perspective may transfer to related problems in VLSI floorplanning, texture synthesis, and DNA tile assembly where local overlap consistency is central.

More broadly, the placement-tree representation and tree-based GA operators developed here may serve as a template for other two-dimensional placement and tiling problems where local overlap structure plays a central role.

	CPLEX		Greedy		Stoch. Greedy		Genetic Greedy		Genetic Stoch.	
Tiles	Obj	Time	Obj	Time	Obj	Time	Obj	Time	Obj	Time
6	**6.73**	2.32	7.47	**0.28**	7.40	0.29	6.97	3.92	6.83	3.86
8	**7.23**	12.13	8.27	**0.35**	8.47	0.38	7.77	4.01	7.50	4.00
10	**8.00**	43.88	9.00	**0.44**	9.10	0.44	8.17	3.80	8.13	3.89
20	-	-	11.33	**0.77**	11.57	0.78	10.37	10.24	**10.20**	10.49
30	-	-	12.80	**0.98**	13.43	0.99	**11.93**	12.40	11.97	15.68
50	-	-	15.37	**1.34**	15.67	1.62	**14.20**	35.66	14.30	58.26
60	-	-	15.95	**1.54**	16.60	1.63	**15.00**	69.54	15.10	92.34
80	-	-	17.30	**2.38**	18.20	2.42	**16.20**	163.00	16.80	253.13
100	-	-	18.60	**3.38**	19.55	3.40	**17.50**	276.00	17.94	351.74

Figure 13: Performance comparison across varying number of tiles. CPLEX provides optimal baselines for small scales (Tiles ≤ 10), while other methods scale to larger instances.

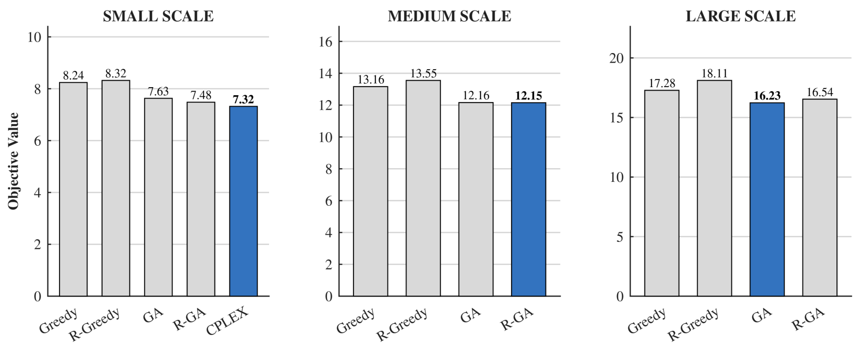

Figure 14: Comparison of objective values across Small, Medium, and Large scales. The blue bars highlight the best-performing algorithm in each category.

References

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

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.

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.

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.

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., 1996. 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.

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.