Verifying Fractal Generating Programs

Dennis Binford [*] **Otto Piramuthu** [*]

Abstract

We present an automated formal verification system for the self-avoidance property in fractals, the property that non-adjacent line segments do not intersect. While mathematical proofs exist for specific fractals, no automated verifier has been developed for this important geometric invariant. We implement a verification framework using the z3 theorem prover that encodes segment intersections as satisfiability problems, correctly distinguishing between permitted endpoint contacts in adjacent segments and true violations. To address the exponential growth in segment count with fractal depth, we utilize two key optimizations: bounding box filtering to eliminate spatially separated segment pairs, and spatial indexing using uniform grid partitioning to reduce comparisons from O(N²) to approximately O(N·k) where k is the average segments per grid cell. We evaluate our system on six well-studied fractals (Koch Curve, Heighway Dragon, Lévy C Curve, Hilbert Curve, Sierpiński Arrowhead, and Gosper Curve) at depths up to 12. Our optimizations achieve substantial verification speedups, enabling analysis at depths where naive z3 verification times out. Our approach generalizes beyond fractals to any geometric structure representable as line segments.

1. Introduction

Background. We define a fractal F as a generating pattern G containing connected line segments $g_0, \ldots, g_k$ with two endpoints, one at the start of g_0 and the other at the end of g_k, a previous state I containing connected line segments $i_0, \ldots, i_l$, and a transition function $T(G, I)$ where each line segment in I is replaced by an affine transformation of G whose endpoints match the endpoints of i. This will be done recursively for n steps with $T_n(G, T_{n-1}(G, I))$, producing a final set of line segments I_n.

[*]Equal contribution . Correspondence to: Dennis Binford <dennisb5@illinois.edu>, Otto Piramuthu <obp2@illinois.edu>.

ECE 584 Class Project, Spring 2025. Instructor: Prof. Sayan Mitra

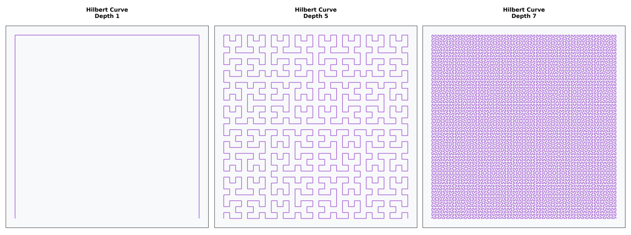

Figure 1. A fractal at different iteration depths

We define a line segment l_1 to *intersect* a line segment l_2 if there is a common point between l_1 and l_2 that is not an endpoint for both segments. Fractals are usually constructed so consecutive segments intersect at one endpoint. We exclude these cases for intersection.

We define a state I to be *self-avoiding* if for all $i, j \in I$, $i \neq j$, i does not intersect j.

A *Lindenmayer System* is a formal grammar we define as a tuple $G = (V, \omega, P)$. V is a finite alphabet containing both replaceable and *variables* non-replaceable *terminals*. The *axiom* $\omega \in V^*$, where V^* denotes the Kleene star of V, is the initial string. $P : V \to V^*$ is a set of *production rules* defining how variables can be replaced by strings of variables and terminals. All symbols $A \in V$ which do not appear on the left hand side of a production rule have an implied rule $A \to A$ denoting no change. As many of these rules as possible are applied simultaneously every iteration.

The problem. We pose the question: Given a G, I, and n, will I_n be self-avoiding? We aim to verify or disprove this for iterations 1 to n for a given fractal.

State of the art. Though many mathematical proofs exist for properties of similarly defined fractal structures, we could find no automated verifier for self-avoidance. Our work provides verification for self-avoidance in fractals.

Our approach. We use Python to generate fractals up to a desired number of iterations in a bounded 2-D plane. Our verifier implemented using the z3 theorem prover either verifies self-avoidance or reports a violation with the involved line segments. The verifier then visualizes the fractals, marking any violations for further inspection.

Existing solutions to verifying invariants of fractals employ manual mathematical proof per fractal. Our solution is more flexible in that it allows any function which takes an iteration as an argument and returns a set of line segments to be verified. This makes it applicable to a large class of mathematical objects beyond fractals.

Contributions Our main contribution is a formal verifier for self-avoidance in fractals. We include various options for optimizations to reduce the runtime of verification.

2. Related work

Fractals have been widely studied in mathematics, both as abstract geometric objects and as computational artifacts. Foundational surveys by Nurujjaman and colleagues (10), along with classical treatments such as those by Mandelbrot (9) and Peitgen, Jürgens, and Saupe (11), consolidate core geometric and analytic properties of fractal constructions. Formal mathematical foundations for self-similar sets and attractors of iterated function systems were established by Hutchinson (6) and later extended in the analyses of Falconer (5) and Edgar (4), providing tools that underpin many contemporary algorithmic treatments of fractal geometry.

One central property relevant to our work is self-avoidance. Bowyer (3) investigates deterministic self-avoiding structures and proposes a constructive algorithm for generating self-avoiding curves within bounded regions. Self-avoidance has also been extensively explored in the stochastic regime. Jia and Heermann (8) examine the scaling behavior and fractal dimensions of random self-avoiding trajectories, linking them to classical polymer-chain models. Related analyses of self-similar curve regularity, such as those discussed by Barnsley (2), further characterize the geometric constraints that arise in recursively generated structures.

Another class of fractals of interest is space-filling curves, which have seen broad application in scientific computing. Many space-filling curves have a locality property: points close on the curve remain a bounded distance apart. This property has been exploited in computational contexts. For instance, Aftosmis, Berger, and Murman (1) analyze how Hilbert- or Peano-based orderings improve cache locality and mesh partitioning, while Tirthapura, Seal, and Aluru (12) study locality guarantees of Hilbert curves in distributed-memory environments. Extensions of these analyses to generalized Peano constructions such as those surveyed by Edgar (4) further formalize continuity, locality, and measure-theoretic behavior relevant to computational use.

Formally verifying systems with fractal or self-similar structure has received comparatively less attention. Ida and Buchberger (7) use the Okito language to model origami folds and verify geometric constraints such as non-intersection. Although origami shares the non-overlap constraint central to self-avoiding fractals, the verification domain is finite and structurally distinct: folds are discrete transitions, whereas fractal rewritings yield unbounded geometric objects. Formal analyses of infinite or self-similar geometric systems are more typically found in measure-theoretic or topological treatments such as those of Falconer (5), but these works do not address automated verification.

3. Methodology

Our methodology consists of three main components: fractal generation, fractal verification, and verification optimizations through the use of bounding boxes and spatial indexing. Each component builds upon the previous to efficiently verify fractal self-avoidance and compare the optimization gains of adding bounding boxes and spatial indexing to the invariant verification.

3.1. Fractal Generation

We implemented the generation of six fractals using a mixture of recursive subdivision and L-systems. Each fractal is defined by an axiom and production rules which are iteratively applied using recursion to generate increasingly complex geometries at greater depths. The fractals include:

Koch Curve: Generated through recursive subdivision, dividing each segment into four parts with a triangular protrusion.

Heighway Dragon: Created by repeatedly folding a line segment at right angles.

Lévy C Curve: Formed by replacing each segment with two perpendicular segments.

Hilbert Curve: Generated via L-system with rules:
$A \to +BF - AFA - FB+$
$B \to -AF + BFB + FA-$
and axiom A.

Sierpiński Arrowhead: Generated via L-system with rules: $X \to YF + XF + Y$, $Y \to XF - YF - X$ and axiom YF.

Gosper Curve: Generated via L-system with rules:
$A \to A - B - -B + A + +AA + B-$
$B \to +A - BB - -B - A + +A + B$
and axiom A.

The above fractals with L-system implementations use symbolic notation in Python code where symbols represent drawing rules and transformations. The letters A and B in the Hilbert and Gosper curves and the letters X and Y for the Sierpiński Arrowhead are variables. The F, $+$, and $-$ symbols are defined as terminals for each curve to move the line segment forward a given amount and then to rotate a specific angle step either counterclockwise or clockwise, respectively.

Each generation depth d produces N_d line segments, where N_d grows exponentially with d. Five of the six fractals we generated grow exponentially as b^d where b is a constant base. The constant base for the Koch Curve, Lévy C Curve, Heighway Dragon, Sierpiński Arrowhead, and Gosper Curve are 4, 2, 2, 3, and 7, respectively. These are derived from each line segment in the fractal splitting into b line segments each iteration. The Hilbert Curve scales differently as it scales with the formula $(2d)^2 - 1$. The exponential nature of fractal scaling is important to consider as it means verification becomes exponentially more complex with increasing depth. This also poses problems for memory as having an exponentially growing number of line segments results in an exponential growth of required memory. These are both important factors to consider in the verification of the self-avoidance property of fractals.

3.2. Fractal Verification

Our verification approach uses the z3 SMT solver in Python to formally verify the self-avoidance property of fractals by checking for intersections between all pairs of non-adjacent line segments. The main challenge is determining whether any two line segments in the fractal intersect at points other than shared adjacent endpoints. We represent each line segment using parametric equations. For a segment s_i with endpoints (x_1, y_1) and (x_2, y_2) any point on the segment can be expressed as $(x(t), y(t)) = (x_1 + t(x_2 - x_1), y_1 + t(y_2 - y_1))$ where $t \in [0, 1]$. A parametric equation can thus be formed for each of the line segments in a pair of line segments for comparison.

Algorithm 1 encodes this intersection test as a satisfiability problem in z3. We introduce real-valued variables for two parametric parameters t and u of the two line segments and constrain the t and u to both lie within $[0, 1]$ to ensure we only consider points on the segments themselves. The intersection constraint requires that both x and y coordinates match at the given (t, u) candidate pair.

A critical distinction in our verification is the treatment of adjacent versus non-adjacent segments. Adjacent segments (where $|i - j| = 1$) naturally share an endpoint by construction of the fractal. Such endpoint-only contact is permissible and does not violate self-avoidance. For adjacent segments, we add an additional constraint that excludes

Algorithm 1 Segment Pair Self-Avoidance z3 Verification

```
1: Input: Line segments $s_i$ and $s_j$ with endpoints
   $[(x_1, y_1), (x_2, y_2)]$ and $[(x_3, y_3), (x_4, y_4)]$
2: Output: Boolean indicating self-avoidance
3: solver = z3.Solver()
4: # parametric equation parameters
5: t = z3.Real('t'), u = z3.Real('u')
6: # make a parameter for each endpoint as a variable
7: # add a constraint for each variable for each endpoint
8: solver.add($x_1 == s_i.x_1, y_1 == s_i.y_1$)) # example
9: # intersection equations
10: $x = x_1 + t(x_2 - x_1) == x_3 + u(x_4 - x_3)$
11: $y = y_1 + t(y_2 - y_1) == y_3 + u(y_4 - y_3)$
12: intersection = z3.And($x, y, t \geq 0, t \leq 1, u \geq 0, u \leq 1$)
13: if $|i - j| = 1$ # adjacent segments
14:     endpoints = $(t = 0 \land u = 0) \lor (t = 0 \land u = 1) \lor (t = 1 \land u = 0) \lor (t = 1 \land u = 1)$
15:     solver.add(z3.And(intersection, z3.Not(endpoints)))
16: else # non-adjacent segments
17:     solver.add(intersection)
18: result = solver.check()
19: return result == z3.sat
```

solutions where the intersection occurs at exactly the endpoints of adjacent line segments. This is only needed for adjacent segments as non-adjacent segment intersections, including those at endpoint connections, are violations of the self-avoiding property.

To verify an entire fractal with N line segments, we invoke Algorithm 1 for all $\binom{N}{2}$ pairs of segments. If the solver returns 'sat' (satisfiable) for any pair, we have found a violation of self-avoidance and make note of the segment pair that created a violation, both for plotting purposes and to count the total number of violations for non-self-avoiding fractals.

3.3. Verification Optimizations

An important realization we take away from the fractal generation is that the exponential nature of generating fractals at greater and greater depths will impact verification times immensely as verifying each pair of segments for a large amount of segments using z3 will scale with $O(N^2)$. In order to get more reasonable verification times at large depths we implement optimizations to prevent line segment pairs from being checked using z3 verification if there is no possibility for intersection. To determine whether there is a possibility for an intersection to even occur we create a bounding box for each line segment which completely contains every point on the segment. If there is no overlap between the bounding boxes of two line segments, then there cannot be a common point between the two line segments and there is no need to check the pair for intersection. We also incorporate these bounding boxes into spatial indexing to prevent bounding box checks for line segment pairs that are not near one another.

3.3.1. Bounding Box

The naive verification approach requires checking all $\binom{N}{2}$ segment pairs using z3, which becomes computationally prohibitive as N grows exponentially with fractal depth. We introduce an optimization using axis-aligned bounding boxes to efficiently eliminate segment pairs that cannot possibly intersect before invoking the expensive z3 solver.

A bounding box for a line segment is the smallest axis-aligned rectangle that completely contains the segment. For a segment with endpoints (x_1, y_1) and (x_2, y_2), the bounding box is defined by: bb $= [\min(x_1, x_2), \max(x_1, x_2)] \times [\min(y_1, y_2), \max(y_1, y_2)]$

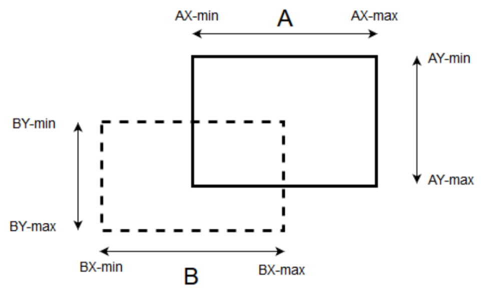

Figure 2. Bounding Boxes for a Line Segment Pair

As can be seen in Figure 7, two line segments can only intersect if their bounding boxes overlap. If the bounding boxes are disjoint that means that the segments themselves must be disjoint. This provides a fast geometric test that can filter out many segment pairs without requiring the more expensive constraint solving in z3 verification.

Algorithm 2 Bounding Box Intersection Check

```
1: Input: Line segments $s_i$ and $s_j$ with endpoints $[(x_1, y_1), (x_2, y_2)]$ and $[(x_3, y_3), (x_4, y_4)]$
2: Output: Boolean indicating bounding box overlap
3: # construct bounding box for $s_i$
4: bb_i.min_x = min($x_1, x_2$), bb_i.max_x = max($x_1, x_2$)
5: bb_i.min_y = min($y_1, y_2$), bb_i.max_y = max($y_1, y_2$)
6: # construct bounding box for $s_j$
7: bb_j.min_x = min($x_3, x_4$), bb_j.max_x = max($x_3, x_4$)
8: bb_j.min_y = min($y_3, y_4$), bb_j.max_y = max($y_3, y_4$)
9: # check for overlap
10: overlap = $\neg((\text{bb}_i.\text{max}_x < \text{bb}_j.\text{min}_x) \vee (\text{bb}_j.\text{max}_x < \text{bb}_i.\text{min}_x) \vee (\text{bb}_i.\text{max}_y < \text{bb}_j.\text{min}_y) \vee (\text{bb}_j.\text{max}_y < \text{bb}_i.\text{min}_y))$
11: return overlap
```

Algorithm 2 shows the implementation of this bounding box test for a given pair of line segments. For a query segment s_0, we construct its bounding box and compare it against the bounding boxes of all other segments in the list S. The overlap test checks whether the two boxes overlap by checking whether they have overlapping x and y intervals. Two intervals $[a, b]$ and $[c, d]$ overlap if and only if $\neg(b < c \vee d < a)$. Using the bounding box intersection test algorithm, we create a list of segments that have bounding boxes overlapping with s_0.

This optimization serves as a pre-processing step in our verification pipeline. For each segment s_i, we first use Algorithm 2 to identify candidate segments whose bounding boxes overlap with s_i. Only these candidates are then passed to Algorithm 1 for precise z3 intersection testing. This two-stage approach may dramatically reduce the number of z3 queries, particularly for fractals where most segments are spatially separated. The computational cost of bounding box tests is $O(1)$ per pair which consists only of simple minimum and maximum computations and comparisons. In contrast, each z3 check involves solving a system of real arithmetic constraints which is substantially more computationally expensive.

While the bounding box optimization significantly reduces the number of z3 checks, $O(N^2)$ bounding box comparisons are still required to check all segment pairs. This optimization thus prevents a lengthy verification time from being performed on each segment pair but does not eliminate the need for a comparison check among all segment pairs. For large fractals number of segments in the thousands or millions, comparing each pair's bounding boxes may still yield high verification times. We introduce spatial indexing as another optimization to prevent checking pairs which are far away from each other in the 2D plane.

3.3.2. Spatial Indexing

We partition the 2D plane into a $G \times G$ uniform grid and precompute which grid cell each line segment occupies based on its bounding box. Segments are then grouped by their grid cell assignment. A segment pair can only have overlapping bounding boxes if the segments occupy the same grid cell or adjacent cells. Conversely, segments in distant grid cells are guaranteed to be disjoint. This allows us to perform bounding box tests only on segment pairs within the same spatial region, dramatically reducing the number of comparisons from $O(N^2)$ to approximately $O(N \cdot k)$, where k is the average number of segments per grid cell. For fractals with relatively uniform spatial distribution, $k \ll N$, yielding substantial performance improvements. We can also optimize the average number of segments per grid cell by using a greater grid size to partition the 2D plane into more cells.

Algorithm 3 constructs this spatial index and generates the

Algorithm 3 Spatial Index Candidate Pairs

```
1:  Input: List of line segments $S$ and grid size $G$
2:  Output: Set of candidate segment pairs
3:  # compute fractal bounding box and grid cell dimensions
4:  cell_width = $(x_{\max} - x_{\min})/G$
5:  cell_height $(y_{\max} - y_{\min})/G$
6:  # Build spatial index: map from cell $(i, j)$ to segment indices
7:  grid = set()
8:  for each segment $s_k$ with index $k$
9:      for each cell $(i, j)$ that $s_k$ passes through:
10:         grid[i][j].append($s_k$)
11: # generate candidate pairs from segments in same cells
12: candidate_pairs = $\emptyset$
13: for each cell $(i, j)$ in grid:
14:     for each pair of distinct segment indices $(k_1, k_2)$ in grid[i][j]:
15:         candidates.add(($k_1, k_2$))
16: return candidate_pairs
```

candidate pairs. After building the grid mapping, we iterate through each non-empty cell and generate pairs from segments that co-occur in that cell. We ensure each pair (k_1, k_2) is only added once by maintaining the invariant $k_1 < k_2$ when adding to the candidate set avoiding redundant comparison. After generating the fractal's N line segments, we first construct the spatial index (Algorithm 3). This produces a set of candidate pairs that are spatially proximate. For each candidate pair, we then perform the bounding box overlap test (Algorithm 2). Only pairs that pass this geometric filter are finally checked for intersection using z3 (Algorithm 1).

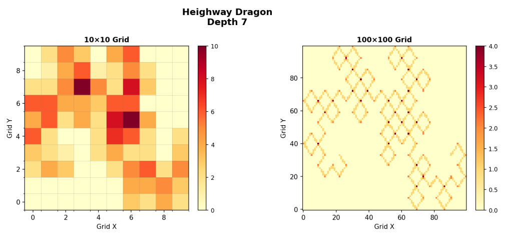

Figure 3. Heat Map of Spatial Indexing

We visualize the difference between using spatial indexing for different-sized grids on the same fractal in Figure 3. The darkness of a grid cell indicates the number of line segments contained in the cell. This comparison shows that as the grid size increases the average number of segments per grid cell decreases, which reduces the number of expensive bounding box and z3 verification checks needed. This, however, comes at the cost of time setting up the spatial index grid and the extra memory cost associated with storing it.

This three-stage optimization pipeline (spatial indexing $\rightarrow$ bounding box test $\rightarrow$ z3 verification) achieves substantial performance gains from naively using z3 for all pairs of segments. The spatial index reduces comparisons from $O(N^2)$ to approximately $O(N \cdot k)$, where k is the average number of segments per grid cell. For fractals with relatively uniform spatial distribution, $k \ll N$, yielding dramatic improvements. The optimal grid size G depends on the fractal structure and may be adjusted in our source code. Larger G creates smaller cells with fewer segments per cell, reducing k and thus the number of candidate pairs. However, this introduces trade-offs: construction time increases as more cells must be populated, and memory overhead grows with G^2 cells. Additionally, very fine grids may create overhead that outweighs the benefits if cells become too small relative to typical segment sizes.

4. Results

Benchmarks and models. We evaluated our verification approach using six well-studied fractals as benchmarks: the Koch Curve, Heighway Dragon, Lévy C Curve, Hilbert Curve, Sierpiński Arrowhead, and Gosper Curve. For each fractal we tested depths ranging from 1 up to 12 with a verification timeout of 2 minutes. The maximum depth we verified for each fractal varied due to their differing growth rates. These fractals served as ideal benchmarks because their self-avoidance properties are mathematically known, allowing us to validate property correctness. Additionally, their exponentially growing complexity at higher depths provides a rigorous test of scalability. We compare three verification configurations:

1. Naive z3 verification checking all segment pairs directly
2. z3 with bounding box filtering
3. Full optimization pipeline with spatial indexing, bounding boxes, and z3

The comparison of these three approaches allowed us to quantify the performance gains achieved by each optimization stage and to understand how verification times scale under different approaches. We also varied the grid size when using the full optimization pipeline to see how the grid size impacts the verification time of fractals over each depth iteration. Each approach was run on the same hardware of a MacBook Air with an Apple M4 chip and 32 GB of memory for consistency in comparison of results. Results may vary depending on the hardware running the fractal verification due to the computationally intensive nature of the verification at greater depths.

Evaluation metrics. We use two primary metrics to evaluate our verification system. First, correctness is measured by whether our verifier correctly determines if a fractal at a given depth is self-avoiding. We validate this against known mathematical properties of each fractal. For instance, the Koch Curve, Hilbert Curve, Sierpiński Arrowhead, and Gosper Curve are known to be self-avoiding at all depths, while the Heighway Dragon and Lévy C Curve exhibit self-intersections starting at certain depths. We also plot the fractals at each given depth and show where self-avoidance violations are detected to ensure through visual inspection that the verification is working correctly. Perfect agreement with these known results and visual inspection of the plots confirms that our z3-based intersection detection correctly handles both adjacent and non-adjacent segment pairs. Second, verification time is our primary performance metric that directly captures the computational efficiency of each verification approach and allows us to quantify the speedup achieved by our optimizations. We report verification times as a function of fractal depth and segment count to demonstrate how each optimization scales with problem size.

Results tables/graphs.

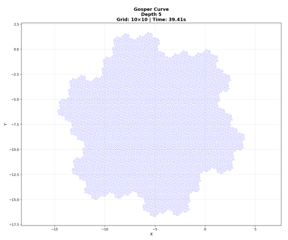

Figure 4. Self-avoiding

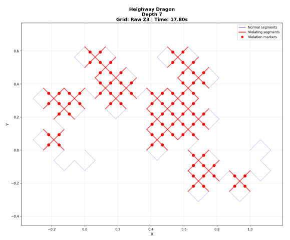

Figure 5. Non-self-avoiding

To validate the correctness of our verification system, we first examine visual evidence from two representative cases. Figure 4 shows the Gosper Curve at depth 5 which our verifier correctly identifies as self-avoiding. The fractal exhibits no intersections between non-adjacent segments with all line segments maintaining spatial separation throughout the structure. This is a fitting example as space-filling curve have lines that do not intersect despite having small distance between them. In contrast, Figure 5 displays the Heighway Dragon at depth 7, where our verifier detects violations of the self-avoiding property. The visualization highlights the detected intersections clearly showing where non-adjacent segments cross.

We performed this visual validation for all six fractals at each tested depth. In every case, our verifier's output matched both the known mathematical properties and our visual inspection of the generated plots. Self-avoiding fractals (Koch Curve, Hilbert Curve, Sierpiński Arrowhead, and Gosper Curve) showed no violations at any tested depth. Fractals with known self-intersections (Heighway Dragon and Lévy C Curve) were correctly identified with violations appearing at the expected depths. This comprehensive visual validation confirms that our z3 intersection detection correctly distinguishes between permitted endpoint contact in adjacent segments and true violations of the self-avoiding property. Having validated that z3 functions correctly through visual inspection of each fractal, we then obtain the benchmark of the amount of time it takes z3 to verify each fractal at each given depth. Figure 6 shows

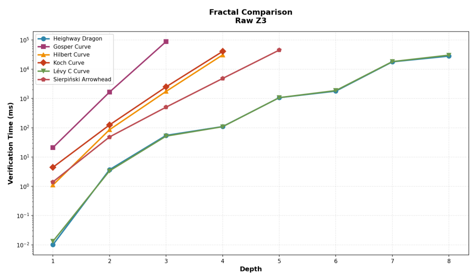

Figure 6. z3 Verification Times

verification times using naive z3 checking on all segment pairs. A timeout of two minutes was given for each fractal. The exponential growth in segment count with fractal depth manifests as rapidly increasing verification times. The Koch Curve and the Hilbert Curve both time out when reaching depth 5. The Sierpiński Arrowhead times out at depth 6. Both the non-self-avoiding fractal tested time out the latest at a depth 9 since they have the least aggressive growth rates. The Gosper Curve, with the most aggressive growth rate, times out earliest at depth 3. These results demonstrate that naive z3 verification times out relatively soon for most fractals not allowing for deeper analysis of the fractals as their structures become more refined. At depths this small it is still possible that the z3 verification cannot accurately compare segments that may be really close together for instance in the space-filling Hilbert and Gosper Curves. These results also give us a benchmark result to compare against the bounding boxes and spatial indexing optimizations.

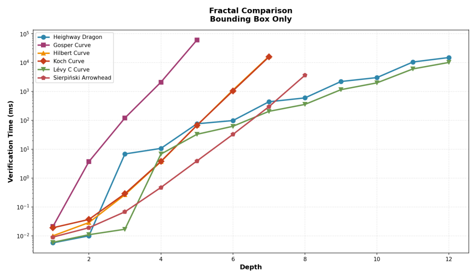

Figure 7. Bounding Boxes Verification Times

Figure 7 demonstrates the substantial performance improvement achieved by incorporating bounding box filtering. A large proportion of expensive z3 checks are eliminated for segment pairs with bounding boxes that do not overlap. Each fractal now has at least one more result at a greater depth when compared with just using z3 for verification of each of the segment pairs. We can also see by comparing Figure 6 and Figure 7 that the verification times are faster with bounding boxes, showing that there is a verification speedup associated with the bounding box optimization.

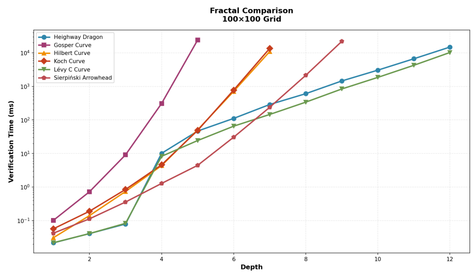

Figure 8. Spatial Indexing Verification Times

Figure 8 shows verification times with the full optimization pipeline incorporating spatial indexing with a 100×100 grid alongside bounding box filtering. This combination yields speedups in verification times when comparing Figure 7 and Figure 8 at higher depths. While not every fractal achieved additional depth iterations, verification times were overall reduced for all fractals at greater depths. However, there is actually a verification time cost associated at the lower depths due to the extra time required to initially set up the spatial index grid optimization. Notably, in the results each fractal was only run up to a depth of 12 so any improvements made on the depth for the non-self-avoiding fractals were not reflected in the results. With a large enough grid size, this could potentially be reduced enough to perform verifications at additional depths on the most exponential fractals such as the space-filling Gosper Curve.

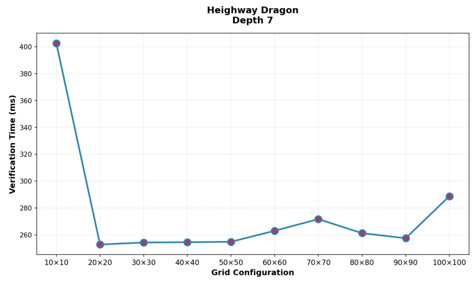

Figure 9. Different Grid Sizes Example

While spatial indexing provides substantial performance gains, the choice of grid size introduces important tradeoffs. Figure 9 illustrates verification times for the Heighway Dragon at depth 7 across grid sizes ranging in increments of 10 from a 10x10 grid to a 100x100 grid. At depth 7, increasing the grid size from 10×10 to 20×20 yields a significant reduction in verification time, demonstrating the benefit of finer spatial partitioning. However, as the grid size continues to increase we actually see that the verification time can increase due to the added time it takes to fit each of the pairs into a grid cell in the spatial indexing grid. This showcases that each fractal at each given depth can have an optimal grid configuration to provide the most improvement in verification time and shows that there is a tradeoff with spatial indexing that means that just because the grid size is larger does not mean it is optimal. Figure

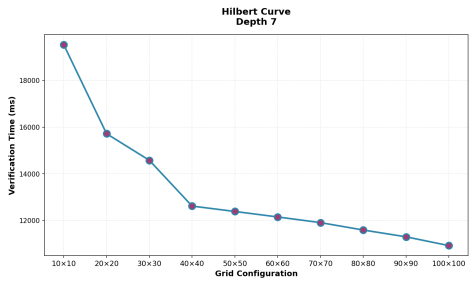

Figure 10. Grid Size Diminishing Returns

10 further illustrates this trade-off by using a space-filling curve example with the Hilbert curve also at a depth of 7. The graph shows us that there are diminishing returns to increasing the grid size, but for this particular example there are still improvements being made all the way up until a grid size of 100x100 (unlike the Heighway Dragon) due to the space-filling nature of the Hilbert Curve.

Code. The code for this project is provided here.

References

[1] Michael J. Aftosmis, Marsha J. Berger, and Scott M. Murman. Applications of space-filling curves to cartesian methods for cfd. In *42nd AIAA Aerospace Sciences Meeting and Exhibit (AIAA 2004)*, 2004. conference paper; DOI verified on AIAA proceedings.

[2] Michael F. Barnsley. *Fractals Everywhere*. Academic Press, San Diego, 1988.

[3] Adrian Bowyer. A new fractal curve. Technical report, Department of Mechanical Engineering, University of Bath, 2002. Available online (PDF).

[4] Gerald A. Edgar. *Measure, Topology, and Fractal Geometry*. Springer, 1990.

[5] Kenneth J. Falconer. *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley & Sons, 1990.

[6] John E. Hutchinson. Fractals and self similarity. *Indiana University Mathematics Journal*, 30(5):713–747, 1981.

[7] Tetsuo Ida, Bruno Buchberger, et al. Computational origami construction of a regular heptagon with automated proof of its correctness. In *Lecture Notes in Computer Science (related volume on computational origami / proving in origami)*. 2004. See related works: computational origami and automated proving (Ida et al.).

[8] Jiying Jia and Dieter W. Heermann. Fractality and topology of self-avoiding walks. *arXiv:2010.15416 [cond-mat.stat-mech]*, 2020.

[9] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, San Francisco, 1982.

[10] M. D. Nurujjaman, A. Hossain, and P. Ahmed. A review of fractal properties: Mathematical approach. *Science Journal of Applied Mathematics and Statistics*, 5(3):98–105, 2017.

[11] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals: New Frontiers of Science*. Springer, 2 edition, 2004.

[12] Srikanta Tirthapura, Sudip K. Seal, and Srinivas Aluru. A formal analysis of space filling curves for parallel domain decomposition. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP'06)*, pages 505–512, 2006. ICPP'06 proceedings.