EE449 HW3
Evolutionary Algorithms

1: The Evolutionary Algorithm

2: Experimental Work

Number of individuals experiments

Discussion of num_inds:

As the number of individuals increases, the algorithm gains diversity that speeds up early discovery and raises the eventual plateau. In the fitness plots titled "num_inds = 5" the curve rockets up but stalls near 0.83 SSIM, and the montage shows patchy images, the horizon band is vague, and the colors are muddled. With "num_inds = 10" and "20" the full-run curves climb higher (≈0.87), and the images sharpen—note the emerging triangular skyline in the montages. At "num_inds = 50" the fitness still rises smoothly and the final SSIM nears 0.89; the montage shows clearer layering and lighter color wash. Going to "num_inds = 75" gives the highest curve (≈0.90 SSIM) and the most coherent horizon/sky gradient, but the gain over 50 is small while computation time grows 50 → 75 ≈ 1.5×. **Best trade-off** for this dataset: num_inds = 75 provides the top SSIM and visibly best image, with diminishing returns beyond 50.

<num_inds> = 5

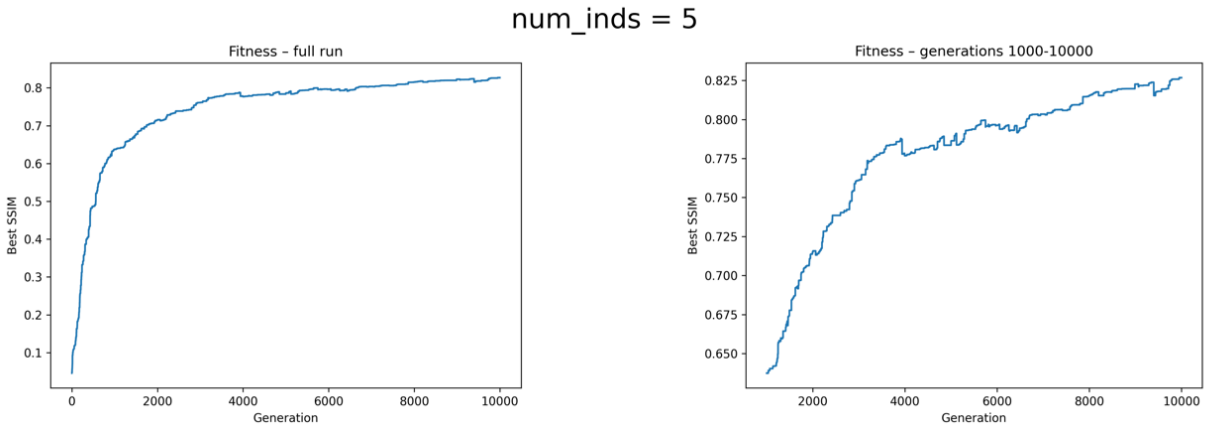

Figure 1: Fitness Plots for num_inds = 5

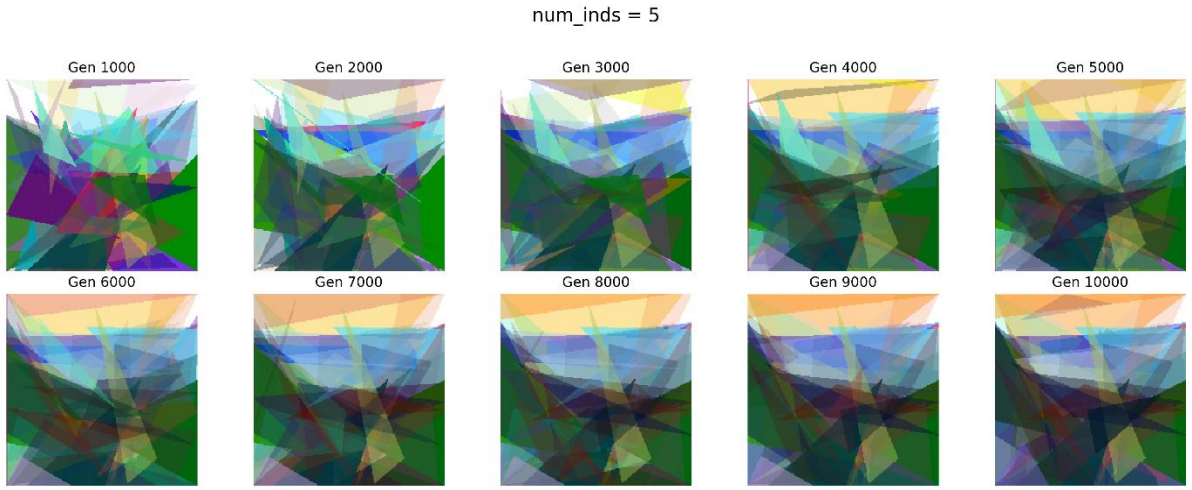

Figure 2: Best Individual in the Population for num_inds = 5

<num_inds> = 10

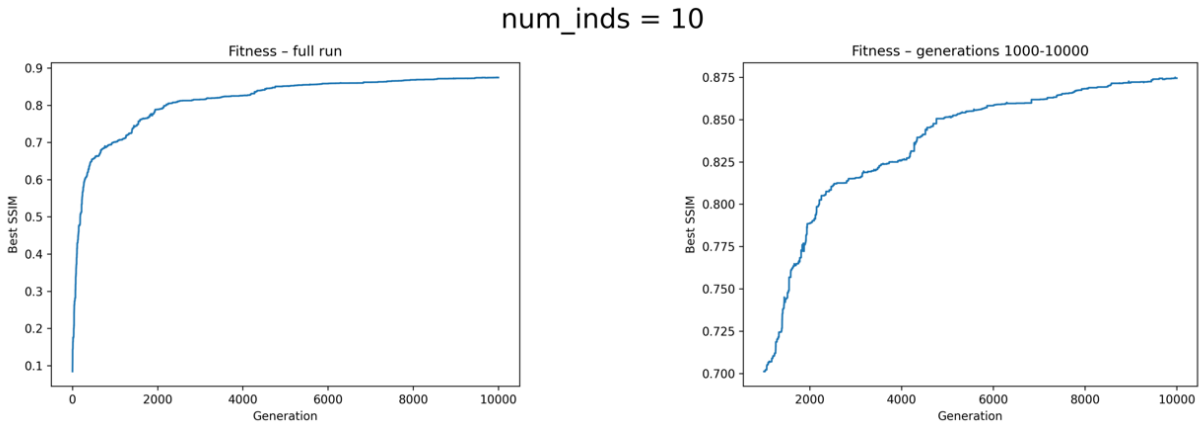

Figure 3: Fitness Plots for num_inds = 10

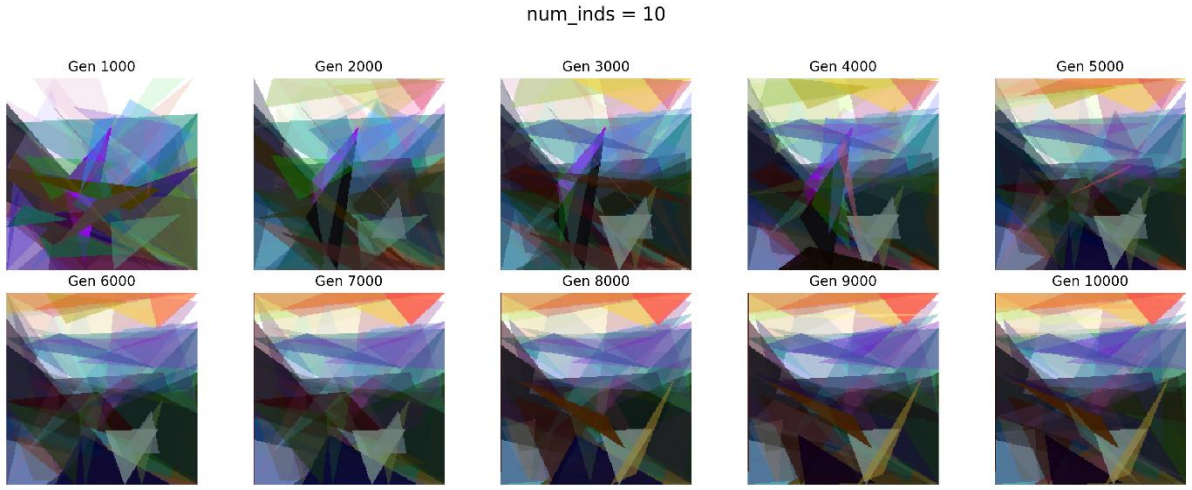

Figure 4: Best Individual in the Population for num_inds = 10

<num_inds> = 20

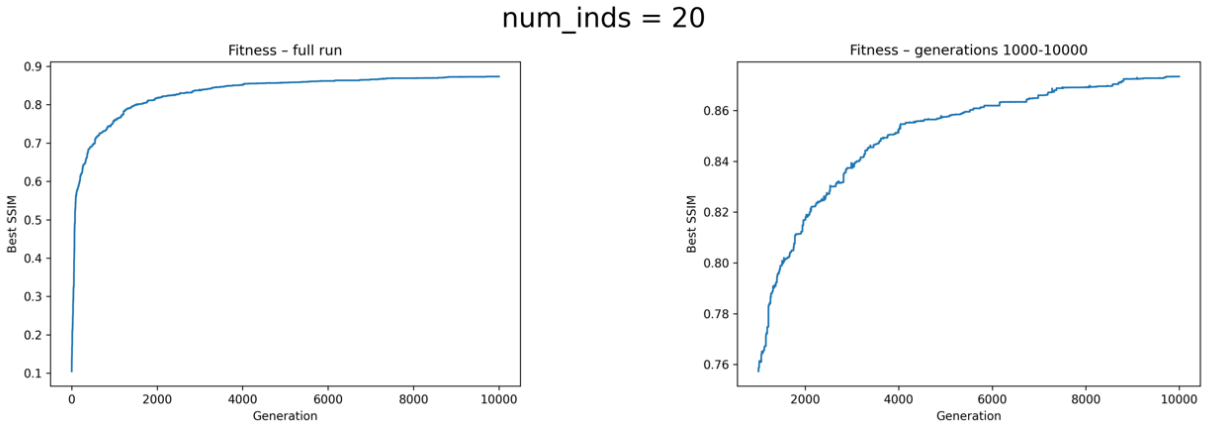

Figure 5: Fitness Plots for num_inds = 20

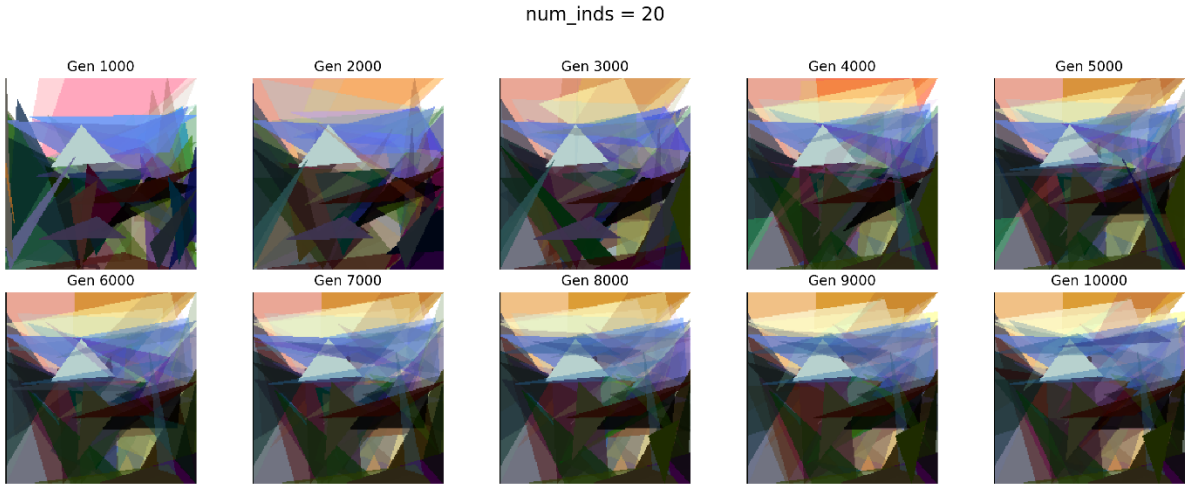

Figure 6: Best Individual in the Population for num_inds = 20

<num_inds> = 50

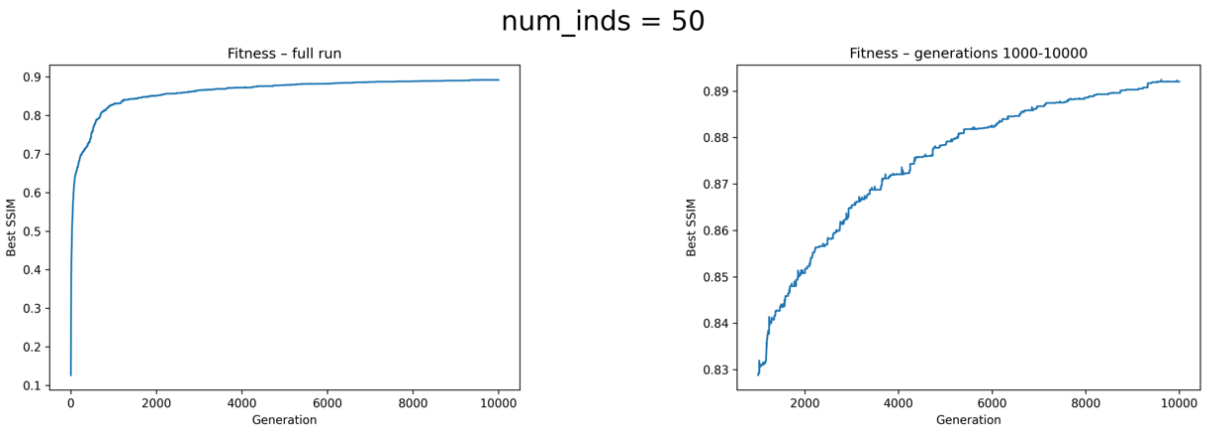

Figure 7: Fitness Plots for num_inds = 50

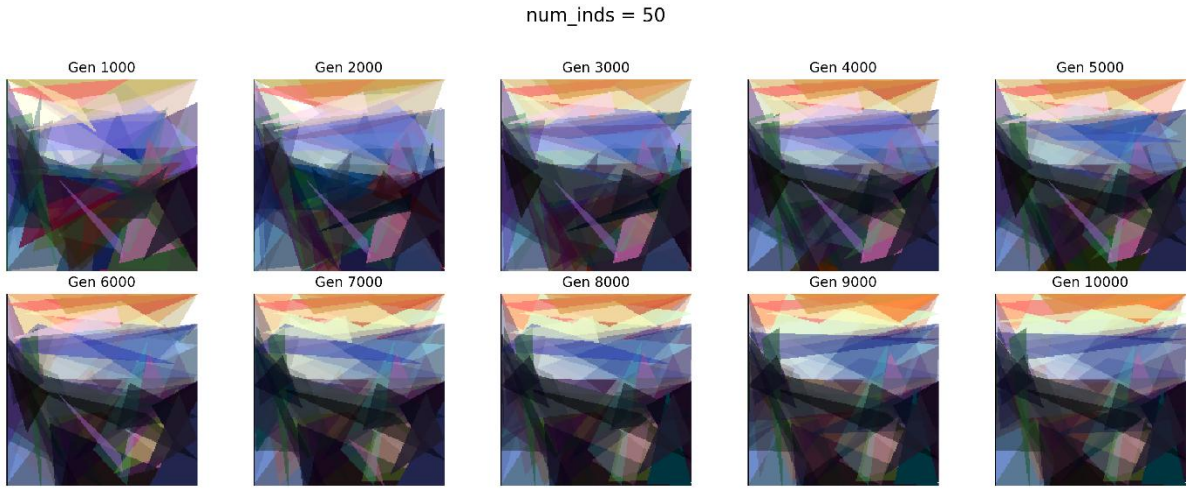

Figure 8: Best Individual in the Population for num_inds = 50

<num_inds> = 75

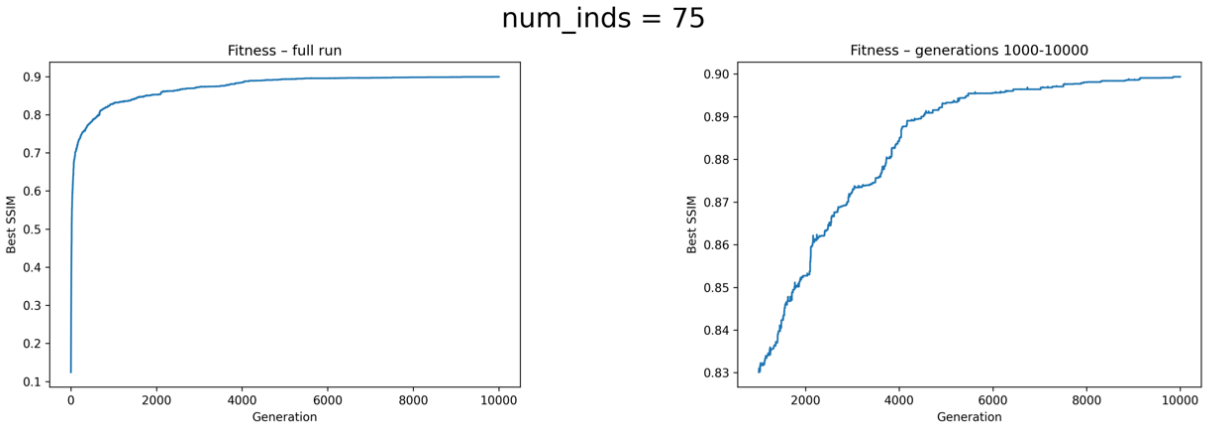

Figure 9: Fitness Plots for num_inds = 75

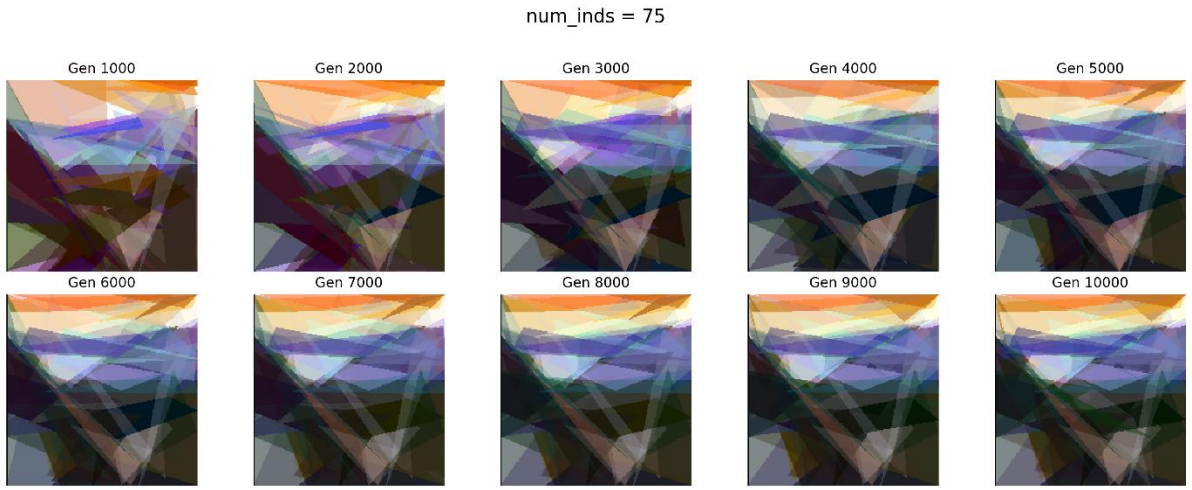

Figure 10: Best Individual in the Population for num_inds = 75

Number of genes experiments

Discussion of num_genes:

Adding more triangles per individual steadily lifts the ceiling on visual fidelity, because each genome can approximate finer detail. In the fitness panels, “num_genes = 10” plateaus early around 0.75 SSIM and its montage stays blocky. As we move through “= 25” and “= 50” the curves climb higher, and the horizon line becomes clearer. With “num_genes = 100” the SSIM pushes past 0.88 and the sky/foreground separation is smooth, while “num_genes = 150” reaches the top score (~0.89) and the montage shows the cleanest gradients. The last step yields only a marginal gain for 50 % more genes, so the **best cost-benefit** point is num_genes = 100—highest quality before diminishing returns set in and the **best visual image** is from num_genes = 150.

<num_genes> = 10

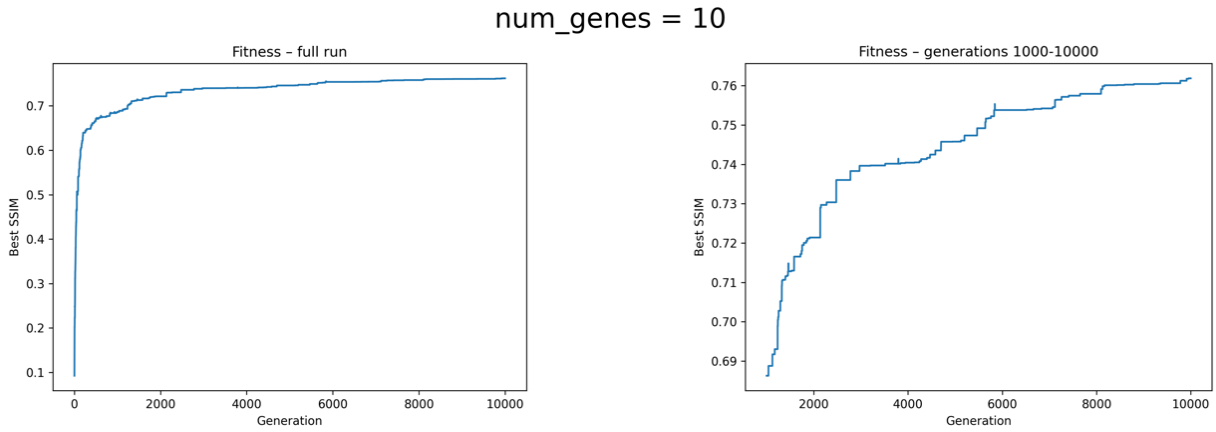

Figure 11: Fitness Plots for num_genes = 10

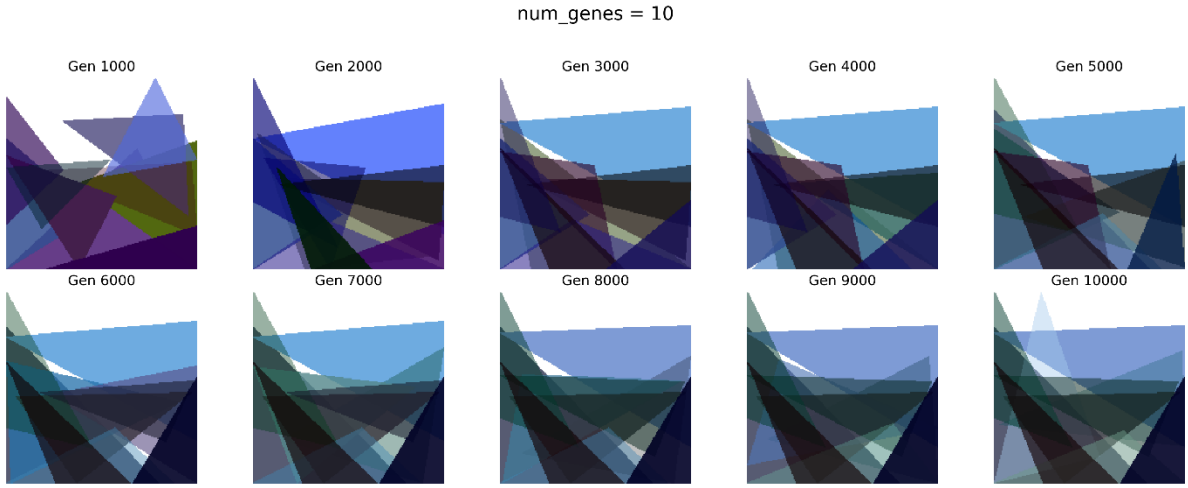

Figure 12: Best Individual in the Population for num_genes = 10

<num_genes> = 25

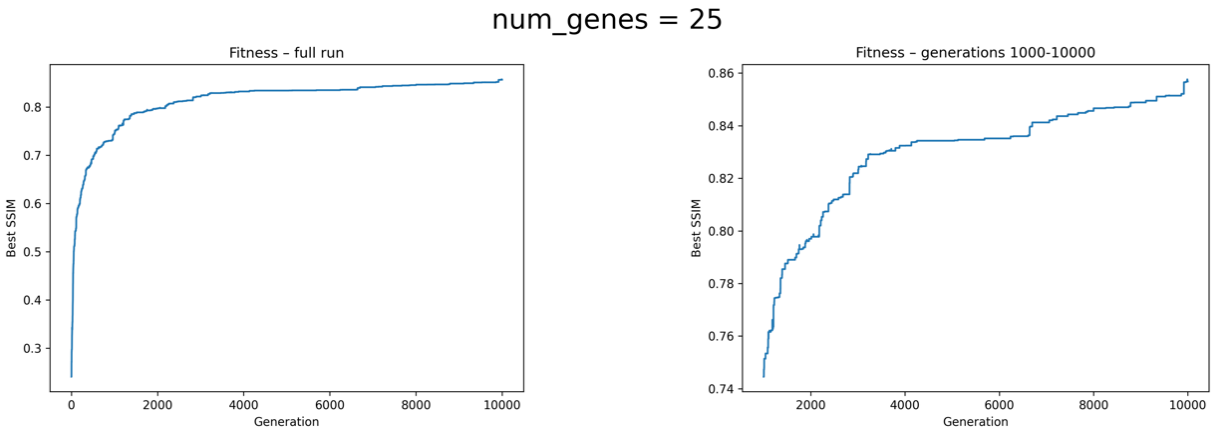

Figure 13: Fitness Plots for num_genes = 25

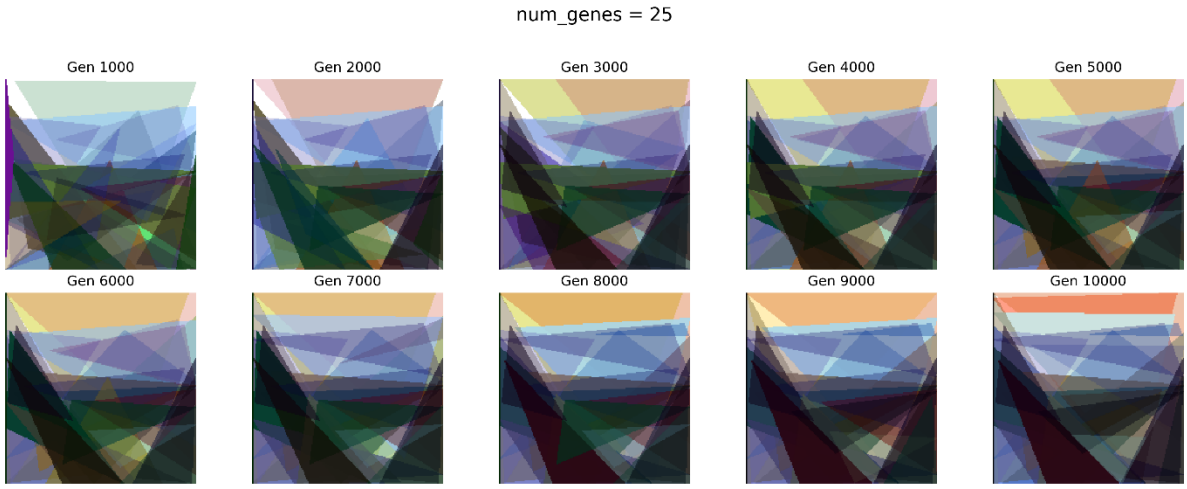

Figure 14: Best Individual in the Population for num_genes = 25

<num_genes> = 50

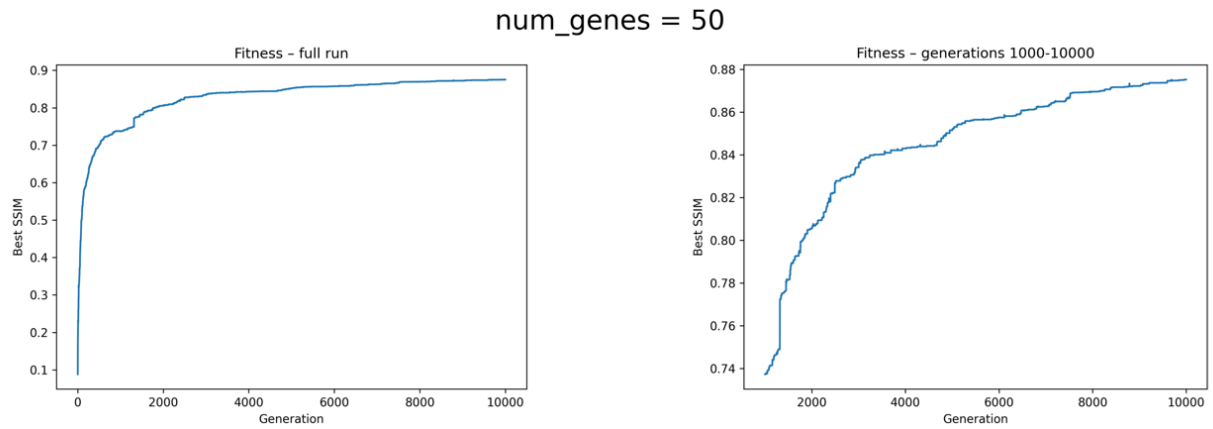

Figure 15: Fitness Plots for num_genes = 50

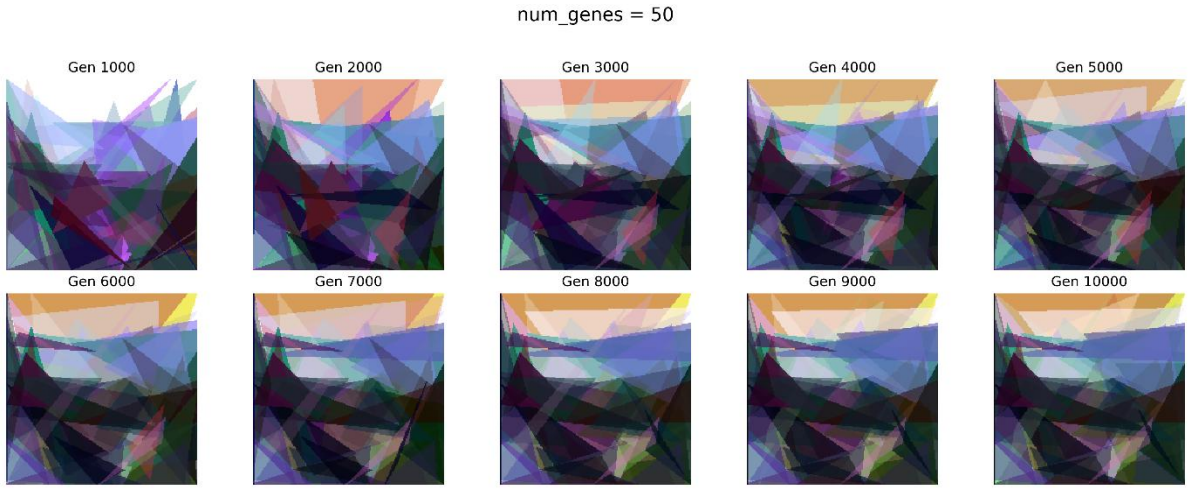

Figure 16: Best Individual in the Population for num_genes = 50

<num_genes> = 100

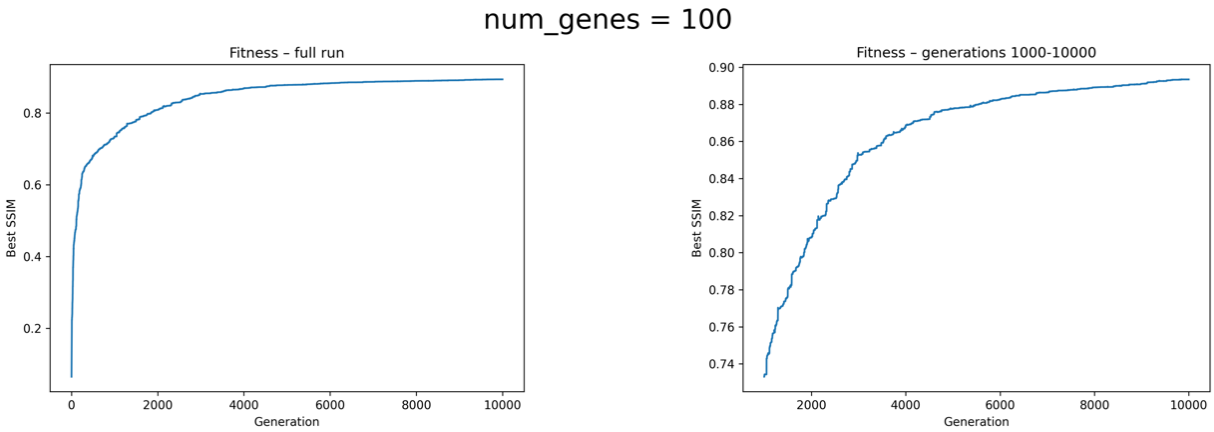

Figure 17: Fitness Plots for num_genes = 100

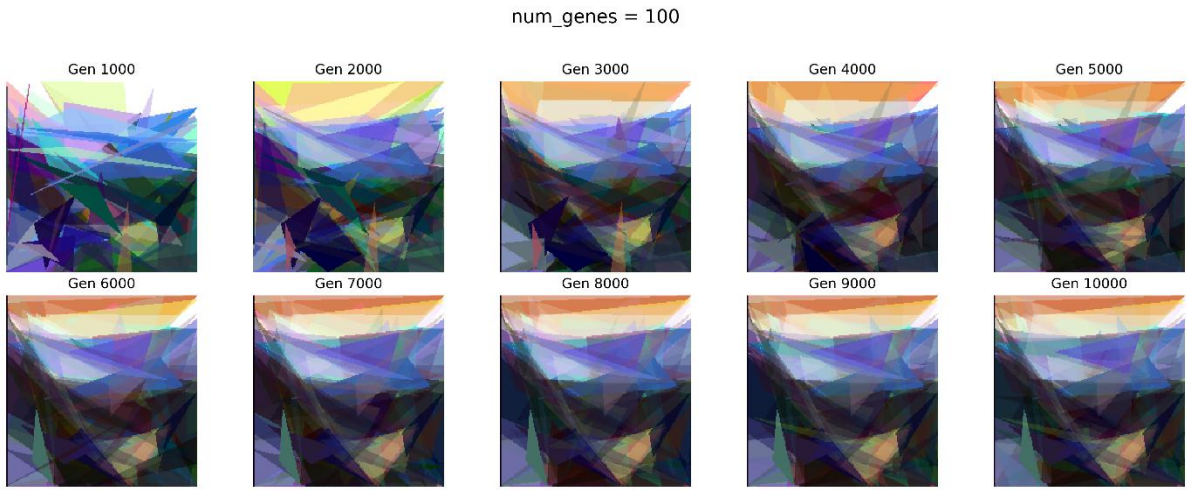

Figure 18: Best Individual in the Population for num_genes = 100

<num_genes> = 150

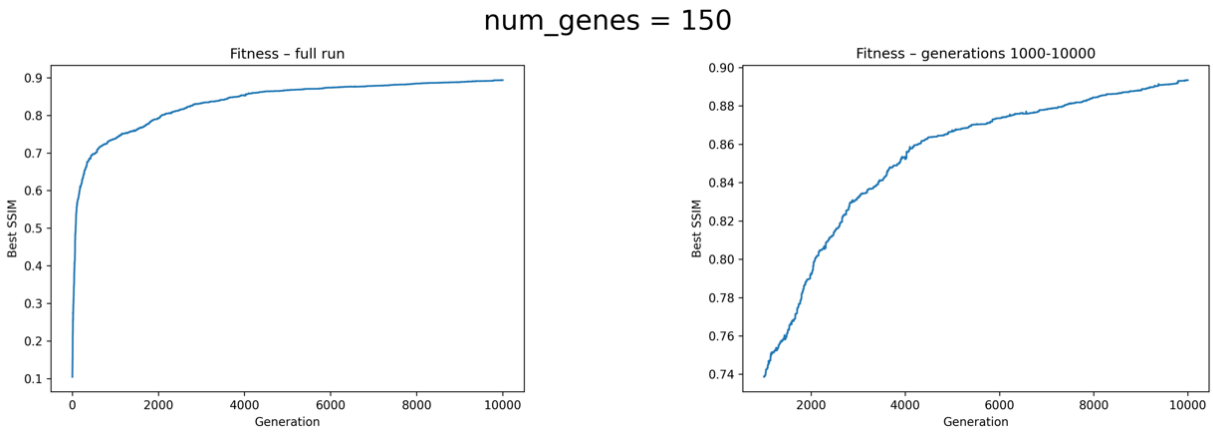

Figure 19: Fitness Plots for num_genes = 150

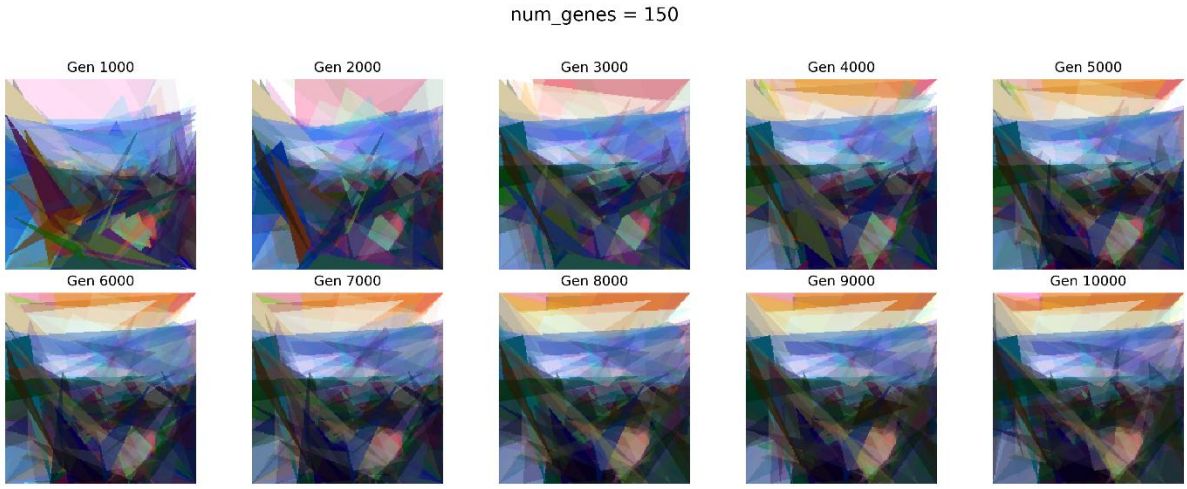

Figure 20: Best Individual in the Population for num_genes = 150

Tournament size experiments

Discussion for tm_size:

Selection pressure rises with tournament size: in "tm_size = 2" the weak pressure keeps diversity high, so fitness climbs but jitters and settles around 0.86 SSIM, and the montage still looks noisy. Raising the pool to "tm_size = 5" strikes a balance—the curve in that plot is the steepest and peaks highest (≈0.89), while the montage shows a crisp horizon and smoother color bands. Pushing pressure further with "tm_size = 10" and "20" reduces jitter but also slows late-stage improvement and yields slightly lower plateaus (~0.88), hinting at premature convergence.
Best value: tm_size = 5 delivers the highest SSIM and the cleanest image with no evident loss of exploration.

<tm_size> = 2

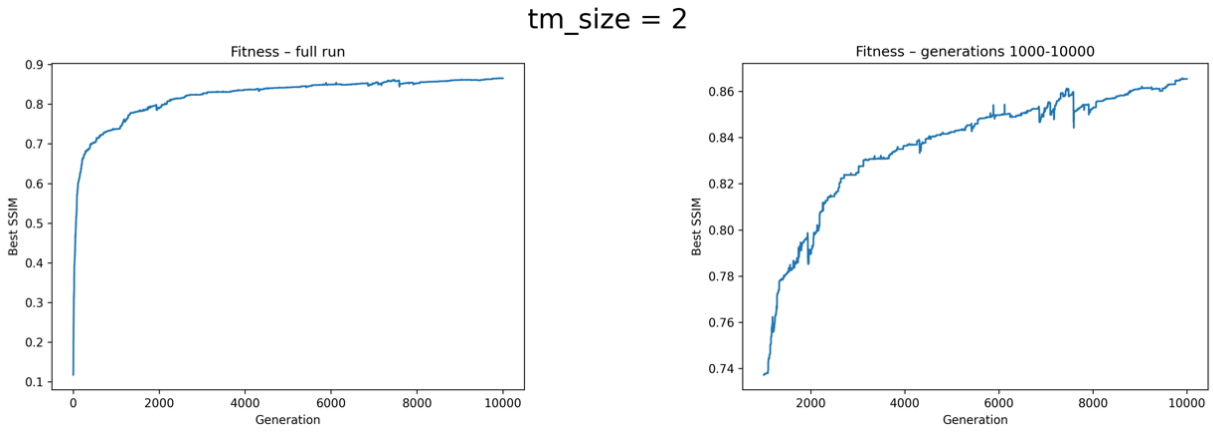

Figure 21: Fitness Plots for tm_size = 2

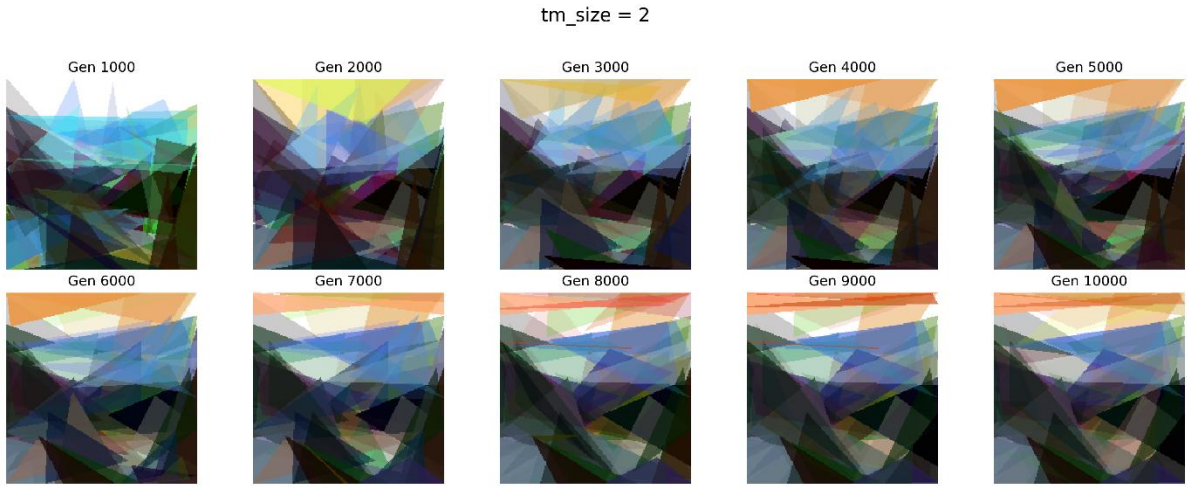

Figure 22: Best Individual in the Population for tm_size = 2

<tm_size> = 5

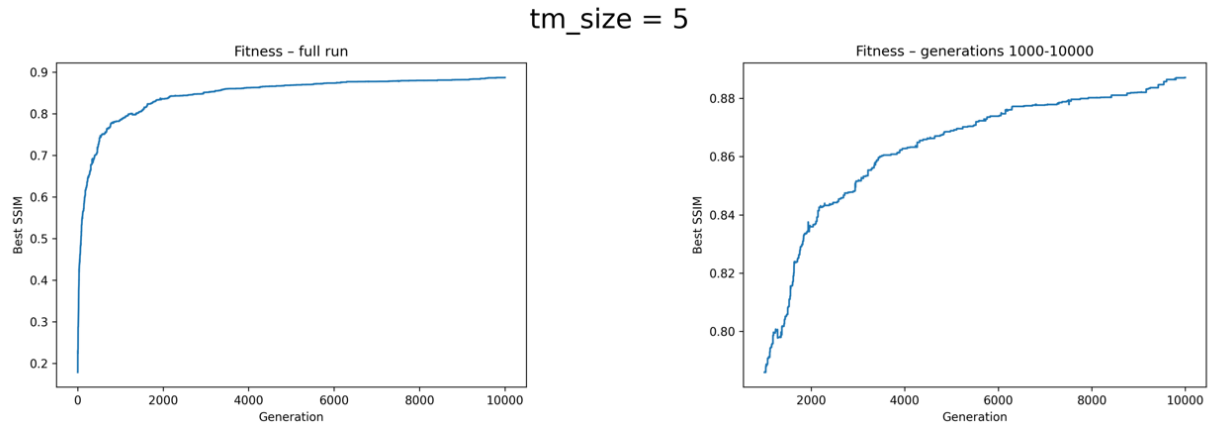

Figure 23: Fitness Plots for tm_size = 5

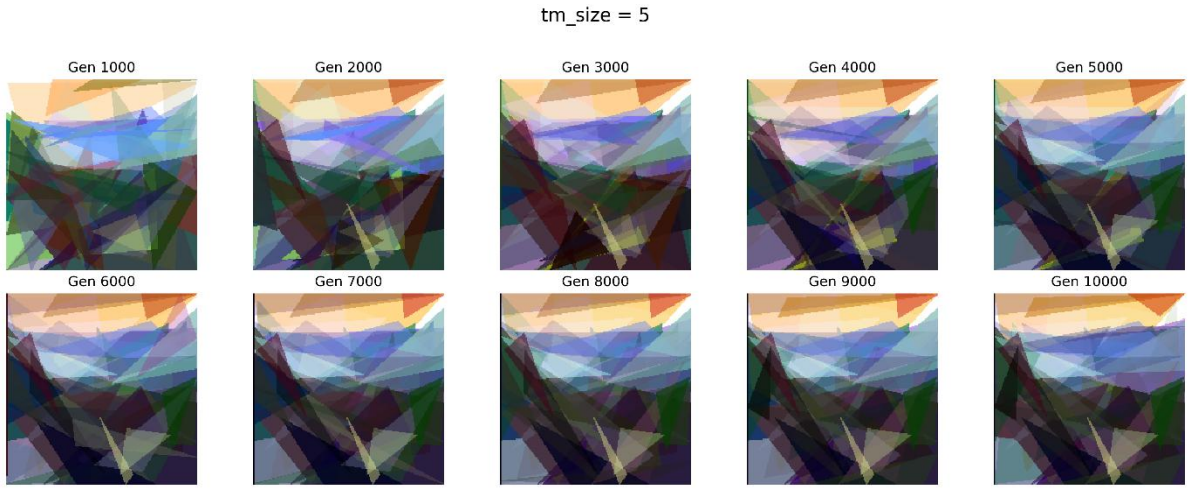

Figure 24: Best Individual in the Population for tm_size = 5

<tm_size> = 10

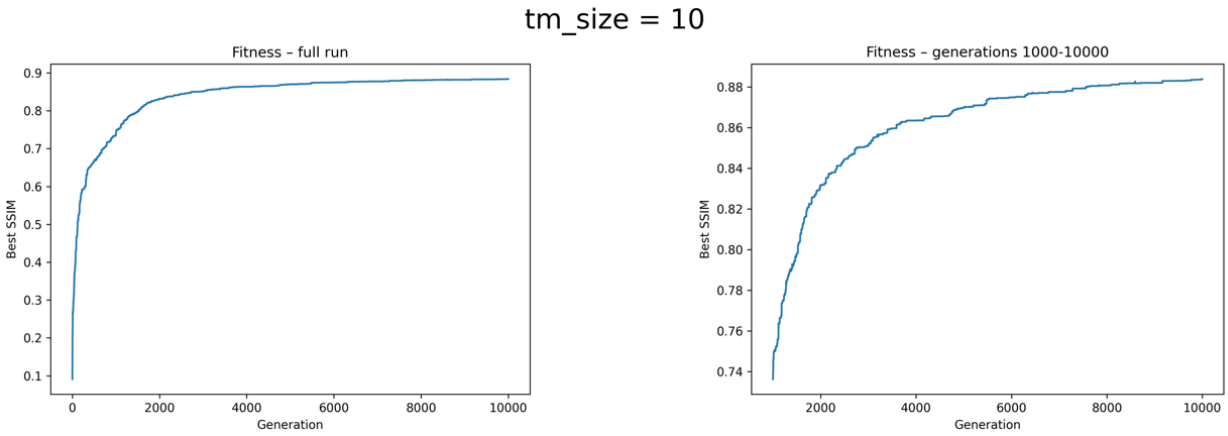

Figure 25: Fitness Plots for tm_size = 10

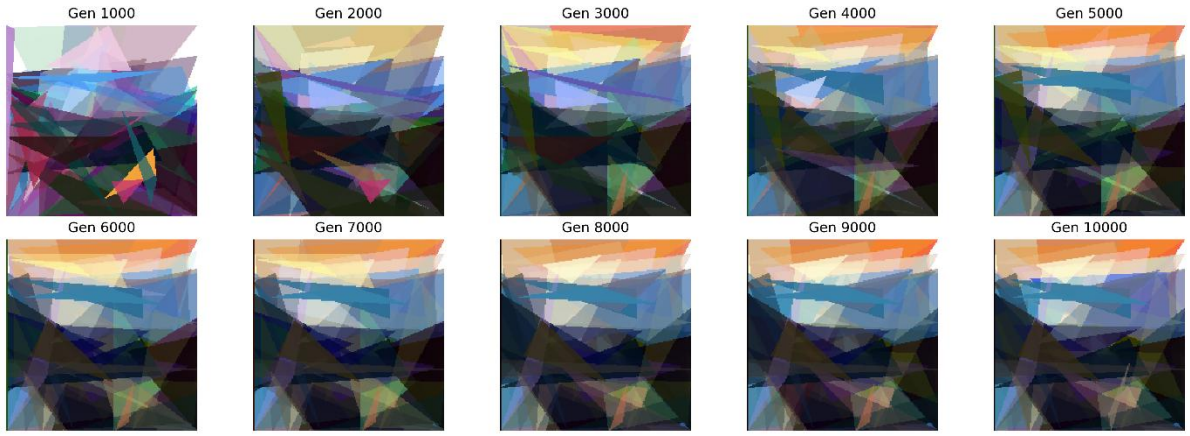

Figure 26: Best Individual in the Population for tm_size = 10

<tm_size> = 20

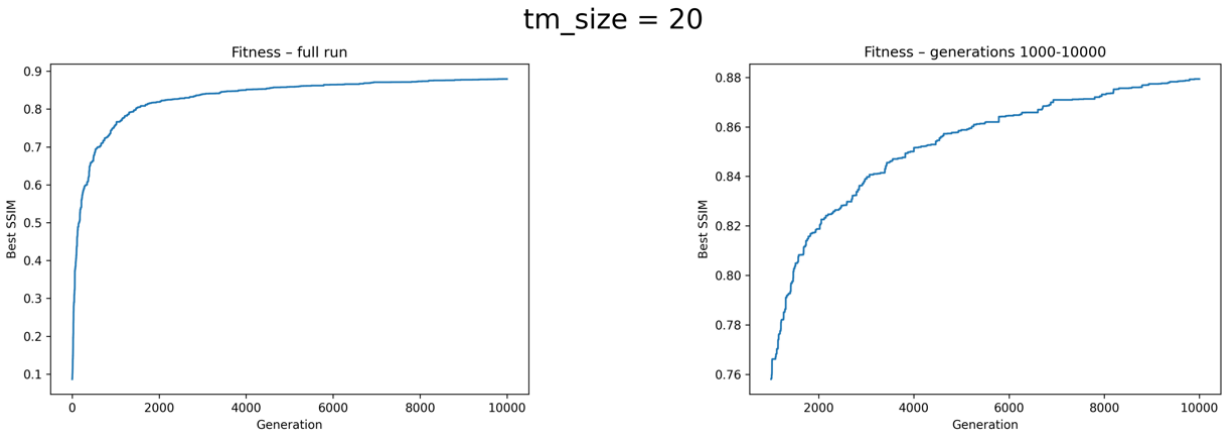

Figure 27: Fitness Plots for tm_size = 20

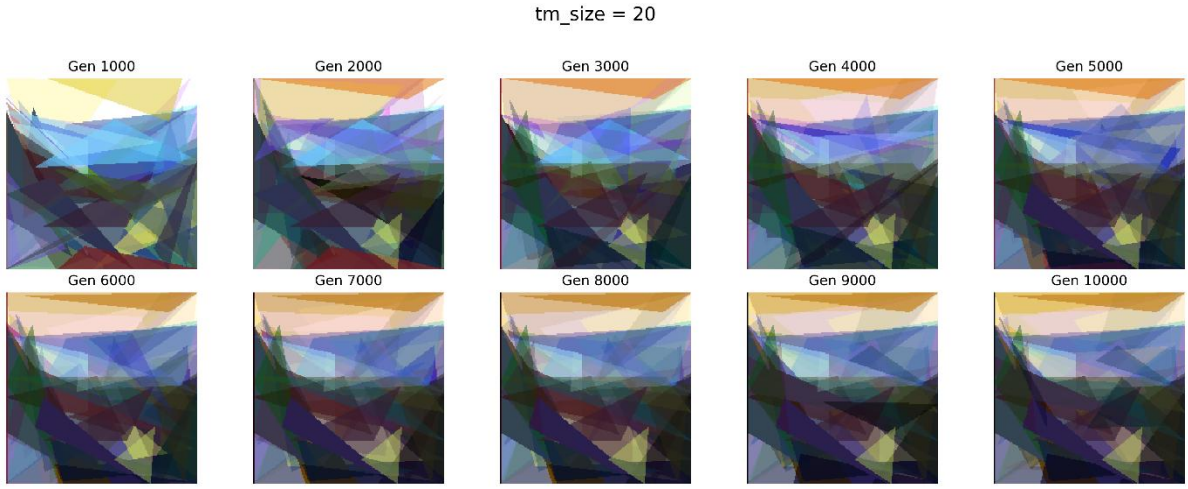

Figure 28: Best Individual in the Population for tm_size = 20

Number of individuals advancing without changing experiments

Discussion for frac_elites

With elite survival the trade-off is exploitation versus exploration. Keeping just 5 % of the population "frac_elites = 0.05" preserves diversity, so the run keeps climbing and ultimately tops out around 0.88 SSIM, the best of the three curves, while the montage shows a consistently improving, well-defined shoreline. Raising the quota to 20 % "frac_elites = 0.2" still converges quickly but the plateau stalls a few hundredths lower as exploration begins to dry up. Pushing elitism to 40 % "frac_elites = 0.4" locks the population too early—fitness climbs more slowly and levels off near 0.86 SSIM, and the images retain blotchy artefacts.
Best choice: frac_elites = 0.05, which gives the highest final fitness without sacrificing stability.

<frac_elites> = 0.05

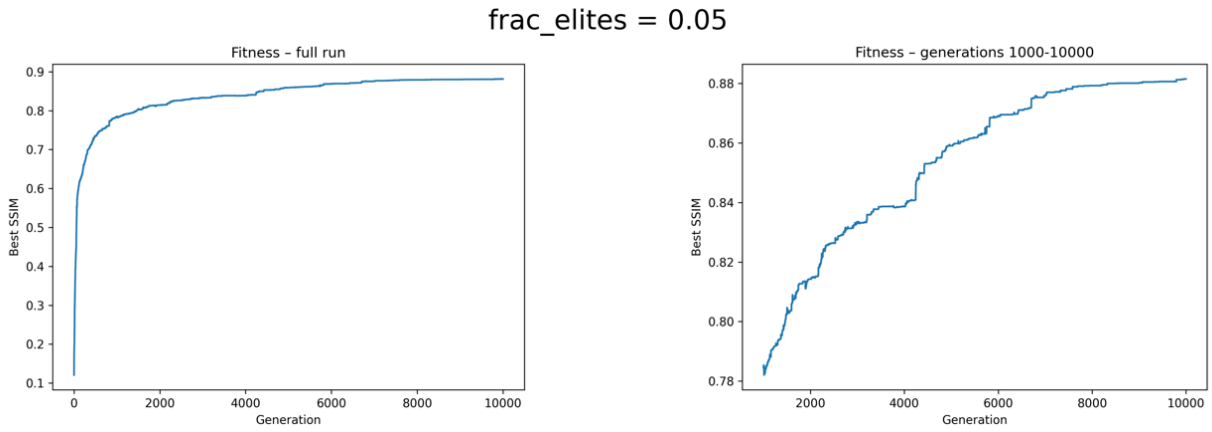

Figure 29: Fitness Plots for frac_elites = 0.05

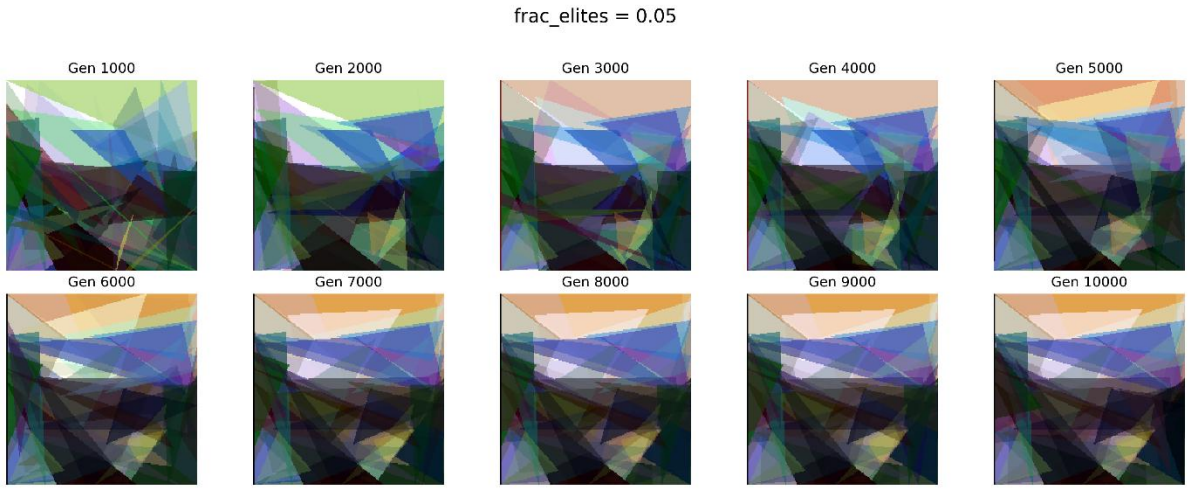

Figure 30: Best Individual in the Population for frac_elites = 0.05

<frac_elites> = 0.2

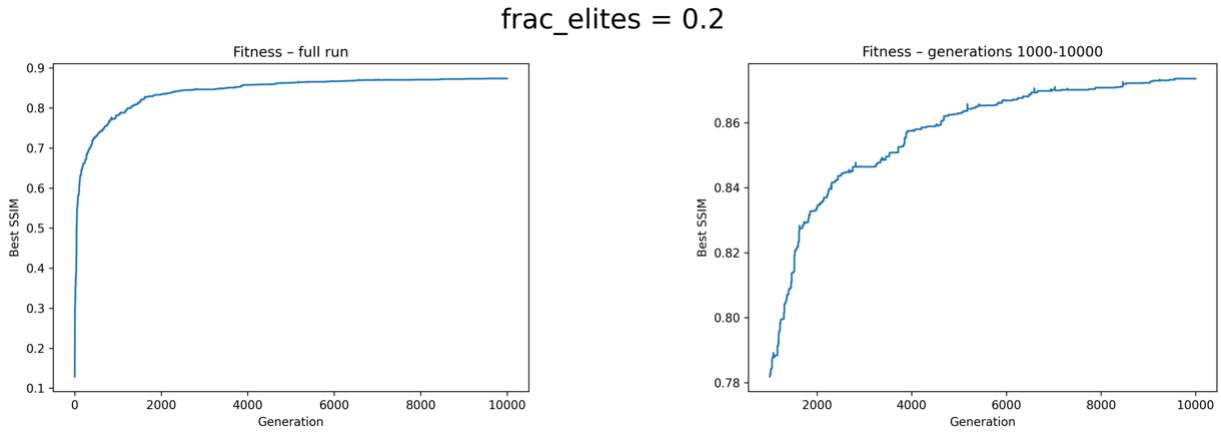

Figure 31: Fitness Plots for frac_elites = 0.2

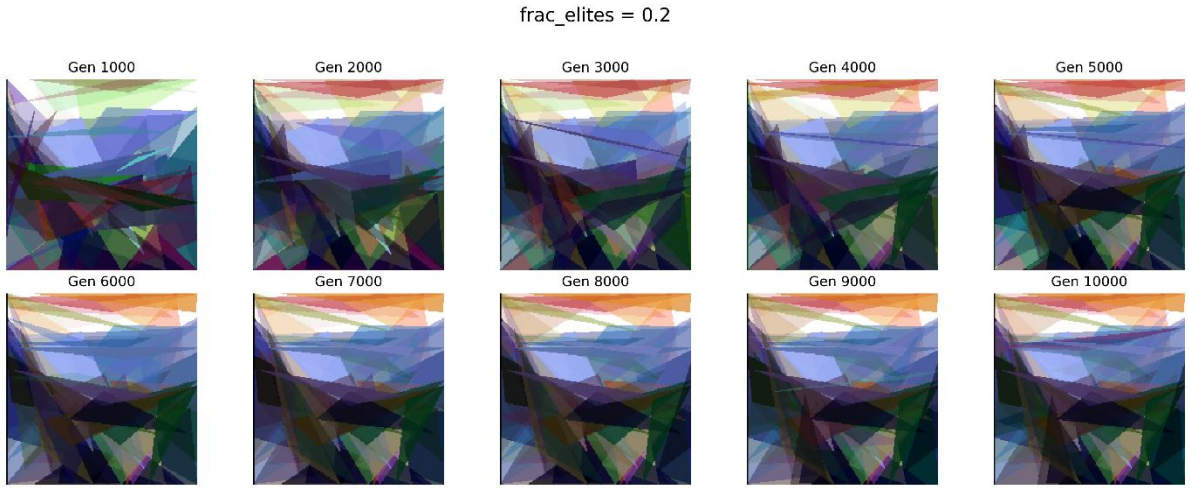

Figure 32: Best Individual in the Population for frac_elites = 0.2

<frac_elites> = 0.4

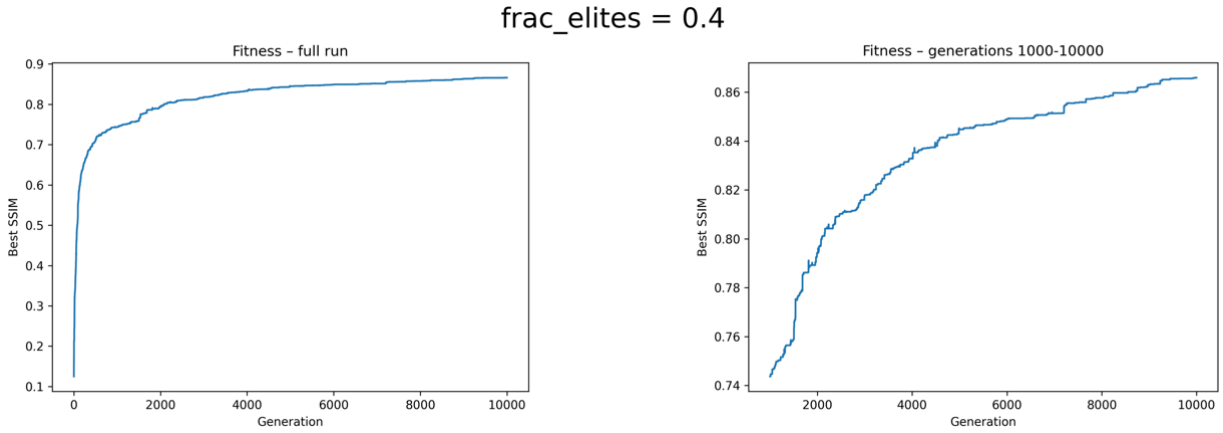

Figure 33: Fitness Plots for frac_elites = 0.4

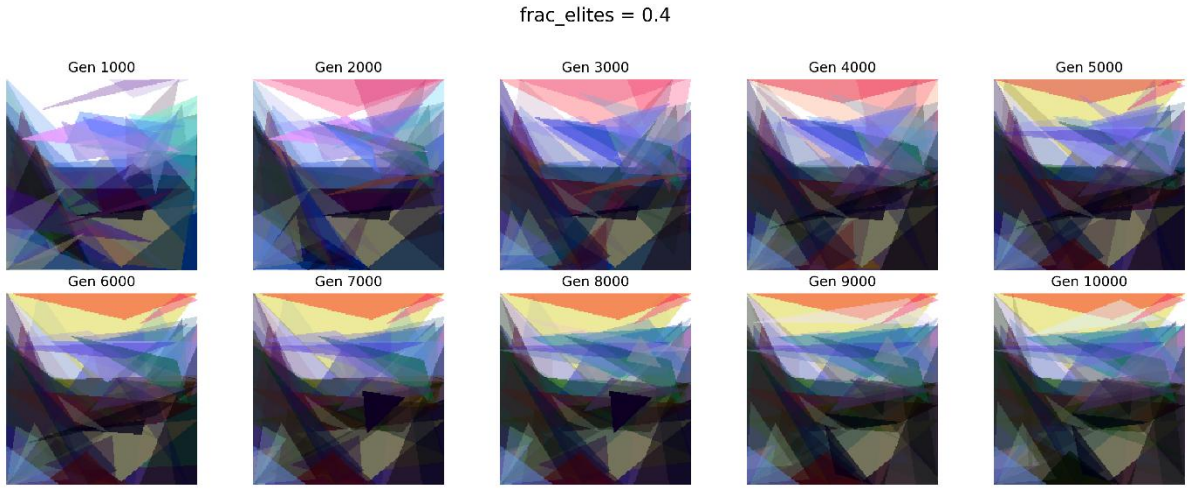

Figure 34: Best Individual in the Population for frac_elites = 0.4

Number of parents to be used in crossover experiments

Discussion for frac_parents:

Comparing the four fitness traces in the plots titled frac_parents = 0.2/0.4/0.6/0.8 shows how expanding the mating pool changes the evolutionary dynamics: with just 20 % parents "frac_parents = 0.2" the curve rises quickly but is jagged and repeatedly dips because only a handful of individuals recombine while the rest rely on mutation, making progress erratic and stalling around SSIM ≈ 0.84; doubling the share to 40 % "frac_parents = 0.4" smooths the trajectory, letting crossover spread useful triangles widely so fitness climbs monotonically past 0.865 as the sky band and horizon stabilize early; at 60 % "frac_parents = 0.6" diversity and selective pressure balance best—the trace accelerates, reaching ≈0.88 by generation ≈6500 and the montage shows the clearest, most coherent landscape with a crisp diagonal shoreline and consistent sky gradient; increasing parents further to 80 % "frac_parents = 0.8" floods each generation with new recombinations but also breaks up strong schemata, so the fitness line continues to rise yet more slowly and plateaus at roughly the same level after extra generations, offering no practical gain for the extra computation. **Best value:** frac_parents = 0.6.

<frac_parents> = 0.2

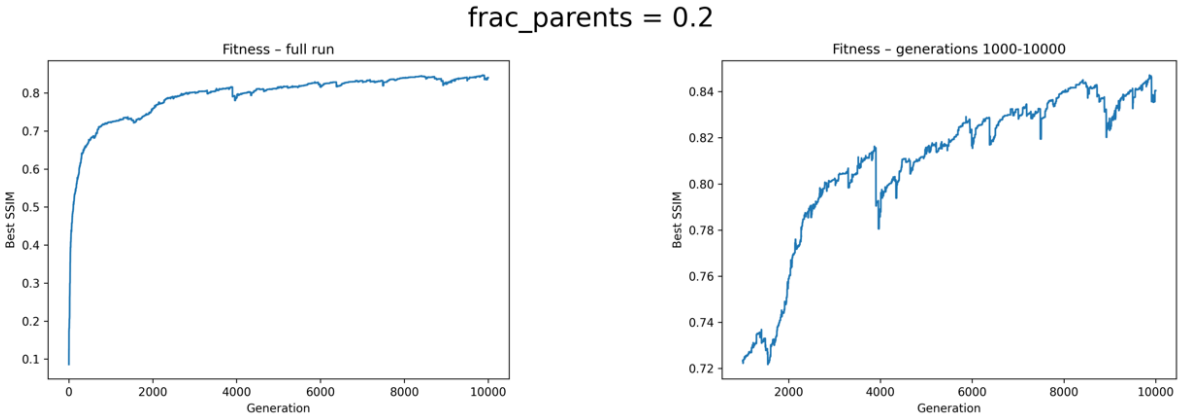

Figure 35: Fitness Plots for frac_parents = 0.2

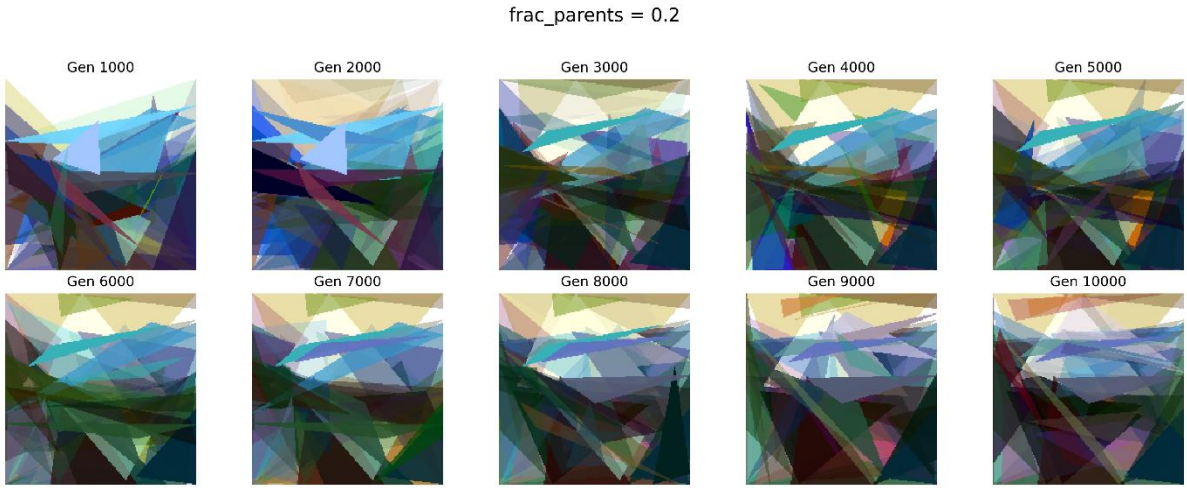

Figure 36: Best Individual in the Population for frac_parents = 0.2

<frac_parents> = 0.4

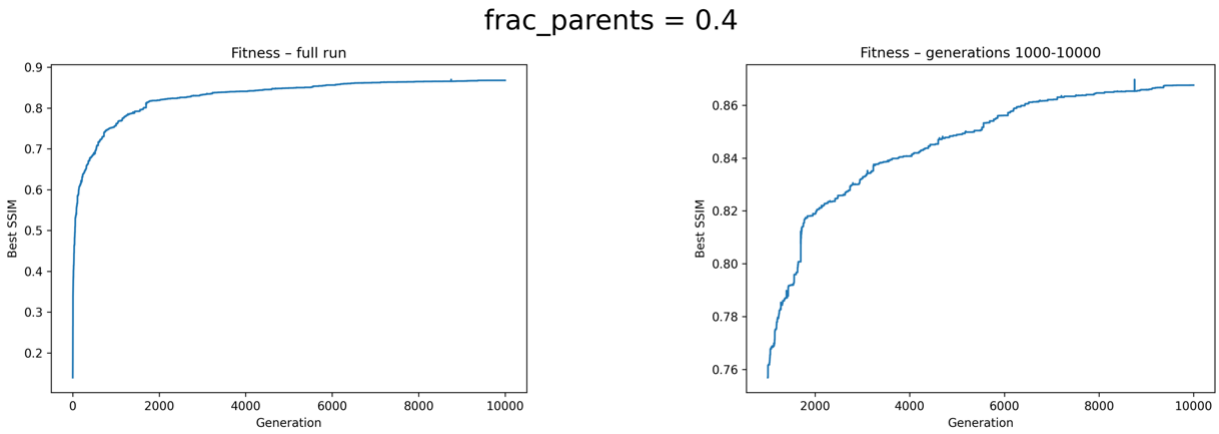

Figure 37: Fitness Plots for frac_parents = 0.4

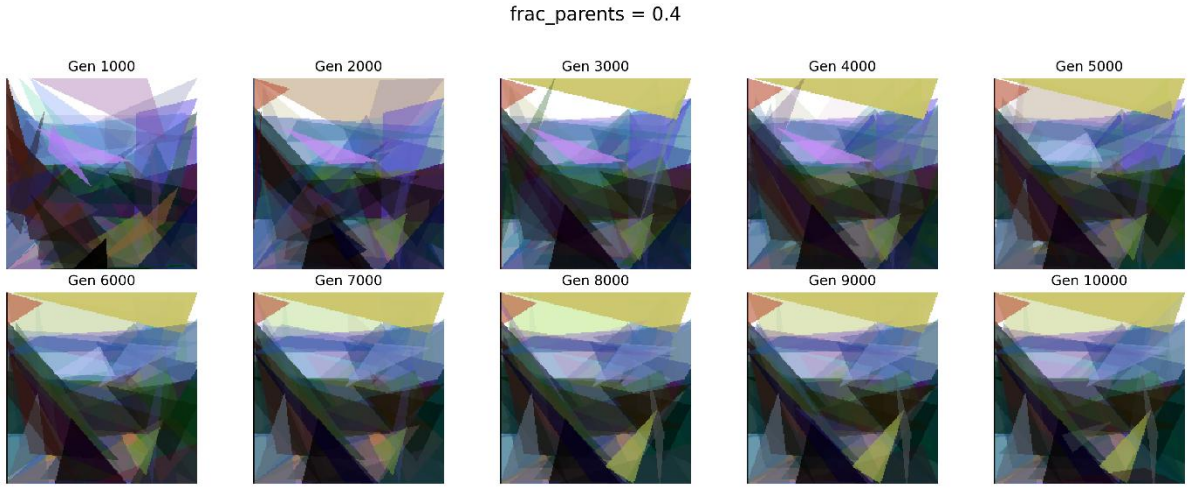

Figure 38: Best Individual in the Population for frac_parents = 0.4

<frac_parents> = 0.6

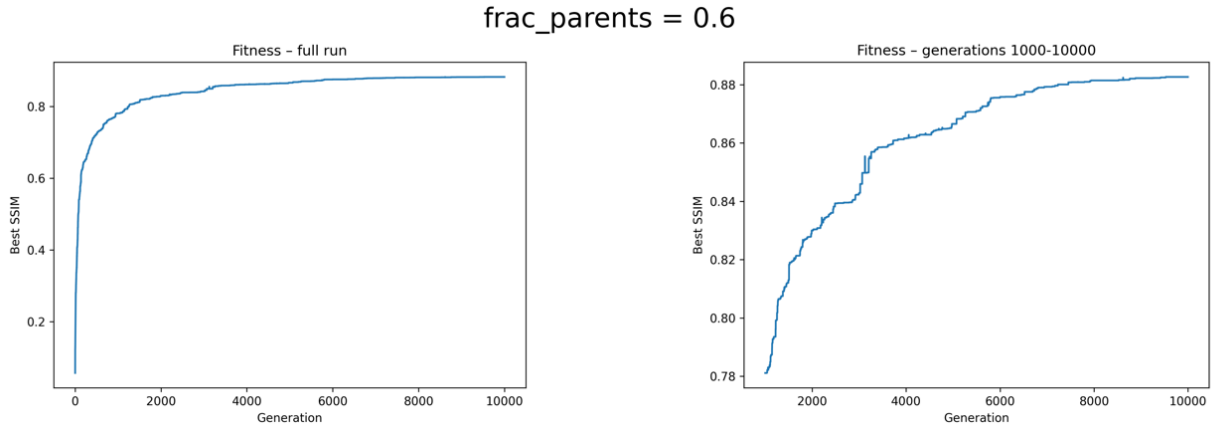

Figure 39: Fitness Plots for frac_parents = 0.6

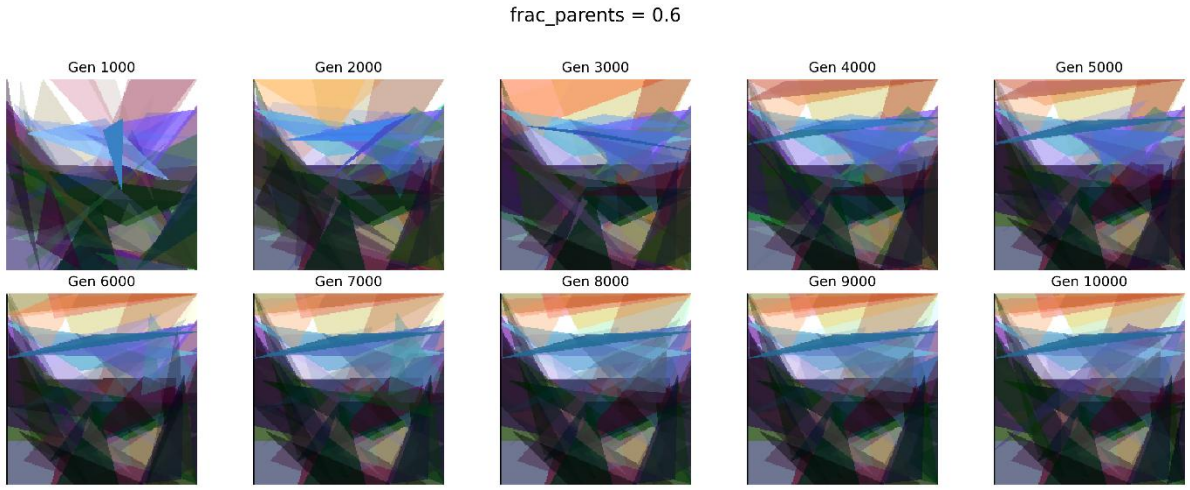

Figure 40: Best Individual in the Population for frac_parents = 0.6

<frac_parents> = 0.8

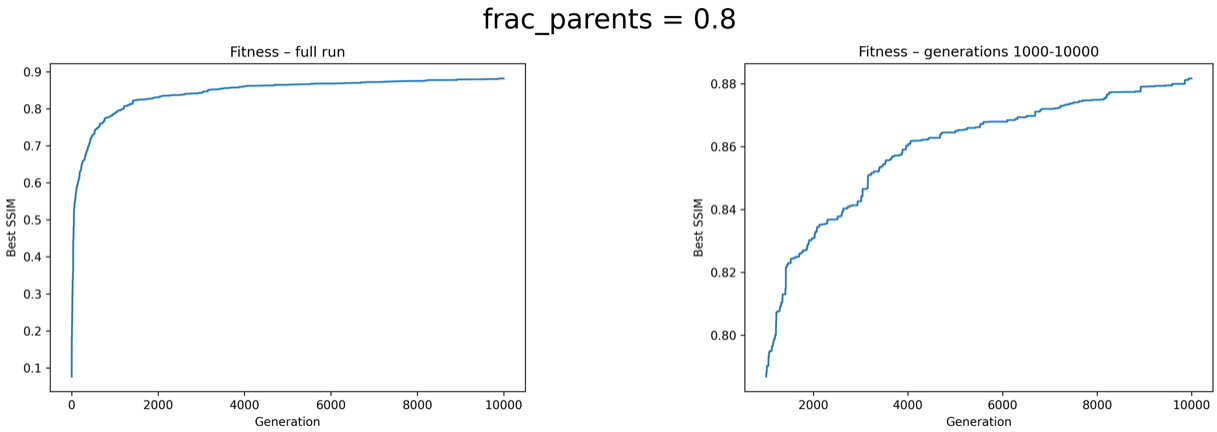

Figure 41: Fitness Plots for frac_parents = 0.8

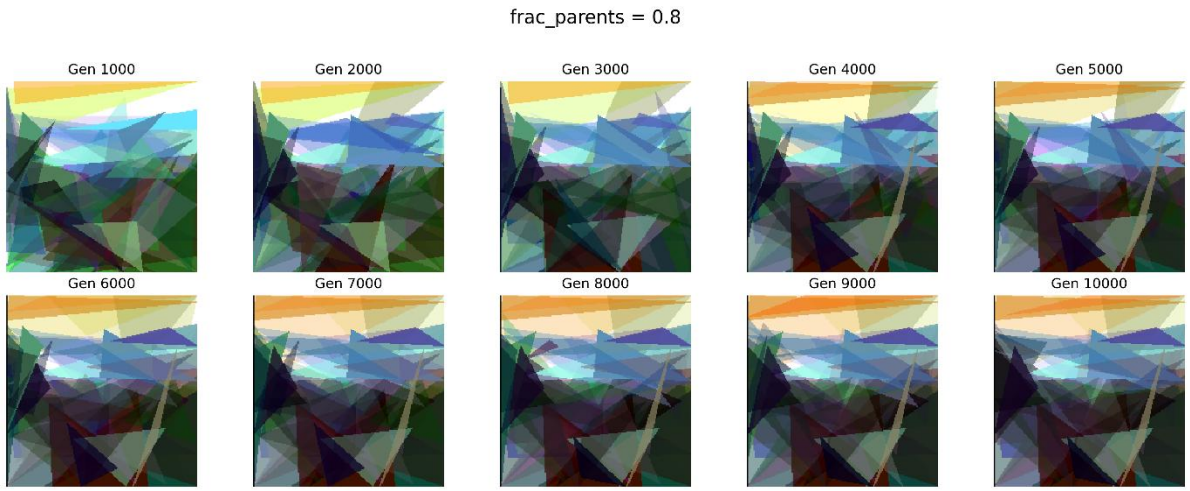

Figure 42: Best Individual in the Population for frac_parents = 0.8

Mutation probability experiments

Discussion for mutation_prob:

The fitness curves reveal that mutation must be present but not dominant: with only 10 % chance "mutation_prob = 0.1" beneficial triangles accumulate steadily yet progress is limited by lack of fresh variation, yielding a smooth but slightly lower plateau around SSIM ≈ 0.87, while doubling the rate to 20 % "mutation_prob = 0.2" injects just enough novelty for the trace to surge quickly and then climb almost monotonically past 0.88, and the accompanying montage shows a crisp horizon and well-separated sky, water, and land forms emerging before generation 6000; pushing probability to 50 % "mutation_prob = 0.5" floods the population with random edits so the fitness line jitters, often falling backwards, and although it eventually meanders into the low 0.84s the images remain noisy with jagged color bands; an extreme 80 % rate "mutation_prob = 0.8" turns evolution into a random walk, the curve oscillates between 0.72 and 0.82 and the montage never settles on a coherent landscape. **Best value:** mutation_prob = 0.2.

<mutation_prob> = 0.1

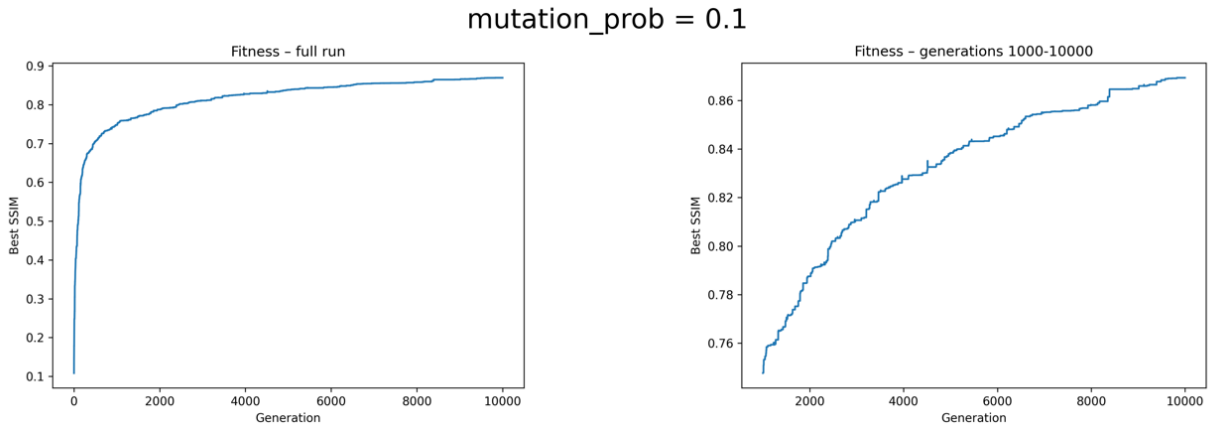

Figure 43: Fitness Plots for mutation_prob = 0.1

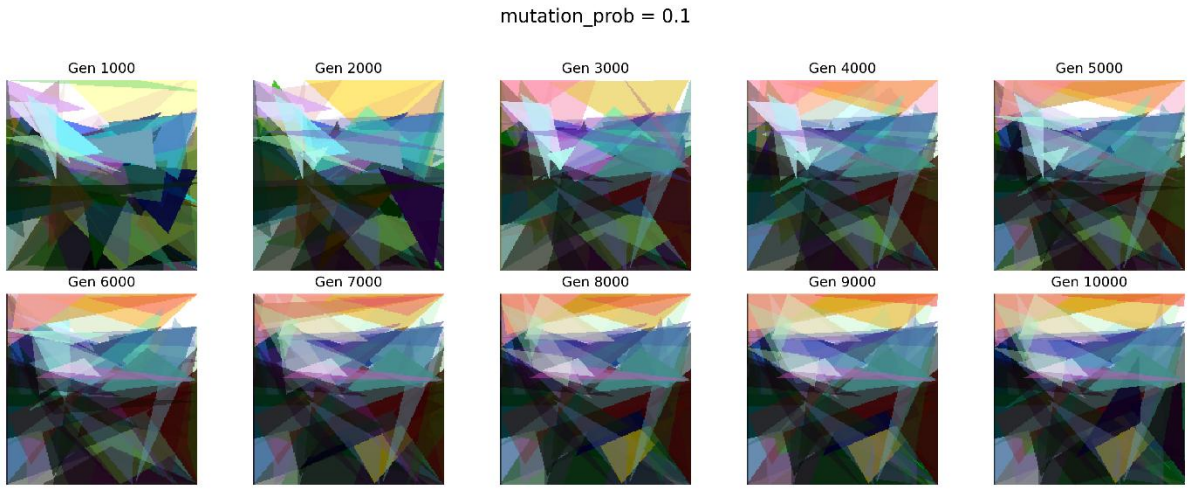

Figure 44: Best Individual in the Population for mutation_prob = 0.1

<mutation_prob> = 0.2

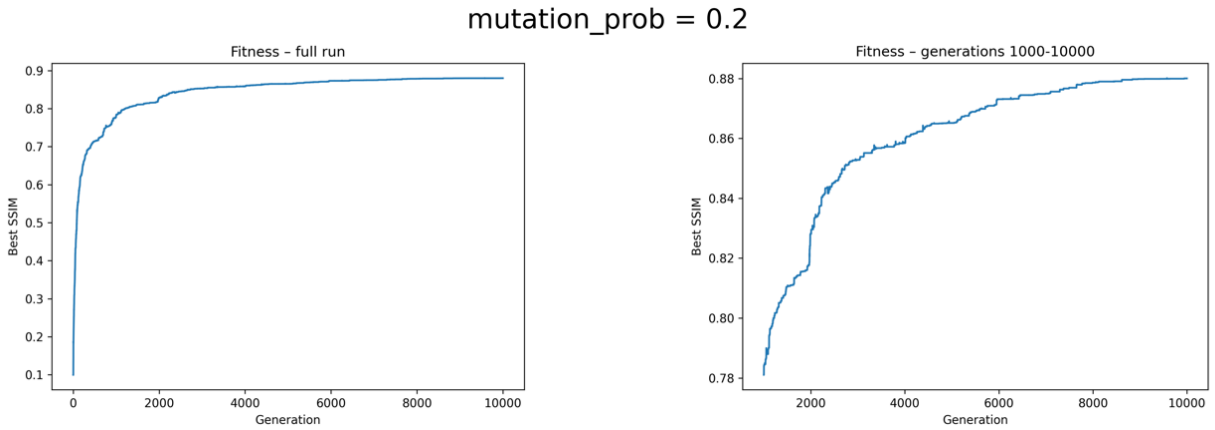

Figure 45: Fitness Plots for mutation_prob = 0.2

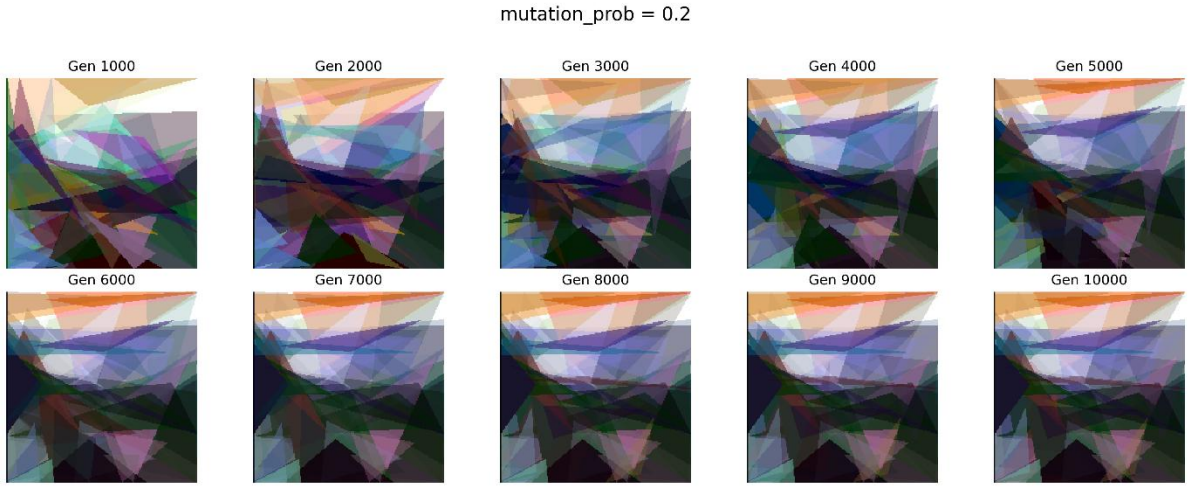

Figure 46: Best Individual in the Population for mutation_prob = 0.2

<mutation_prob> = 0.5

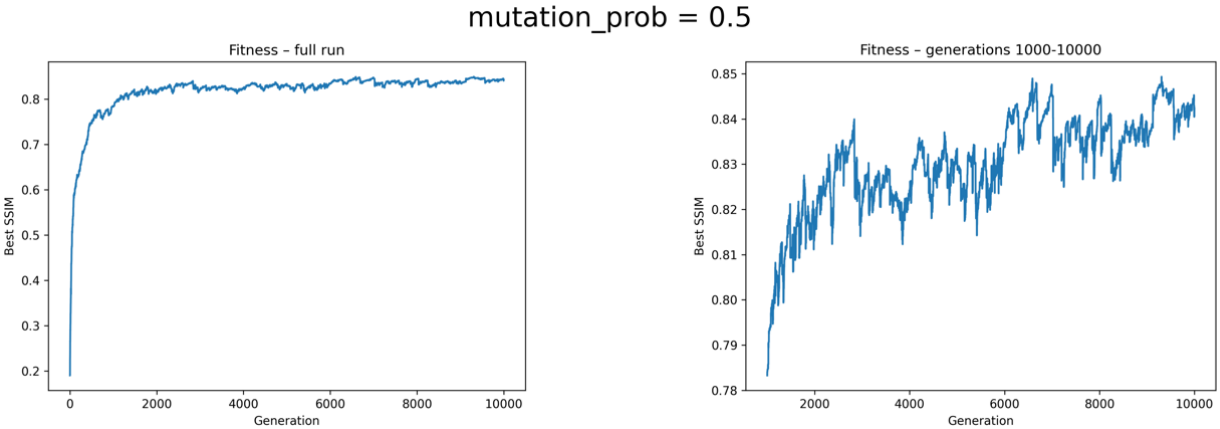

Figure 47: Fitness Plots for mutation_prob = 0.5

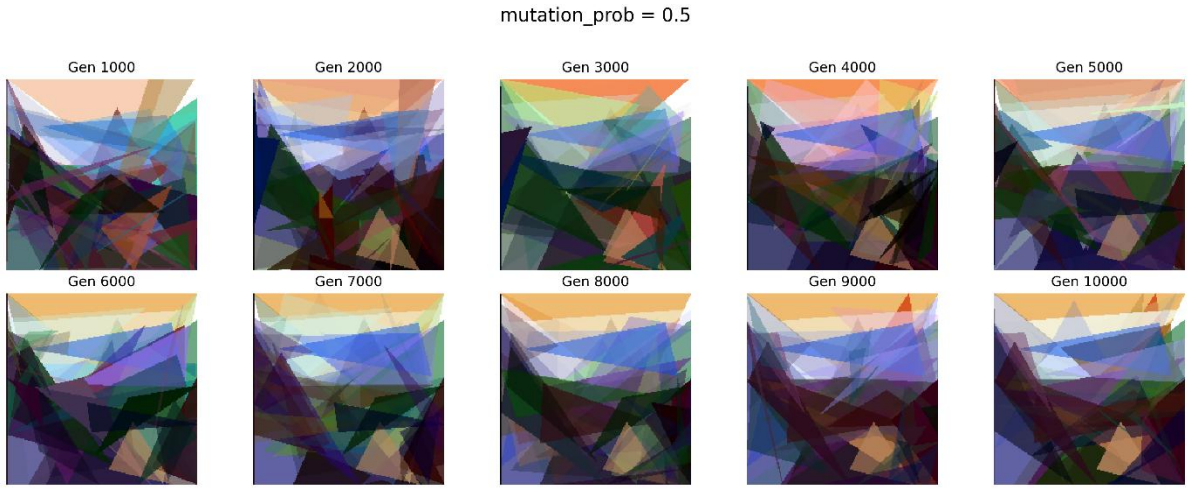

Figure 48: Best Individual in the Population for mutation_prob = 0.5

<mutation_prob> = 0.8

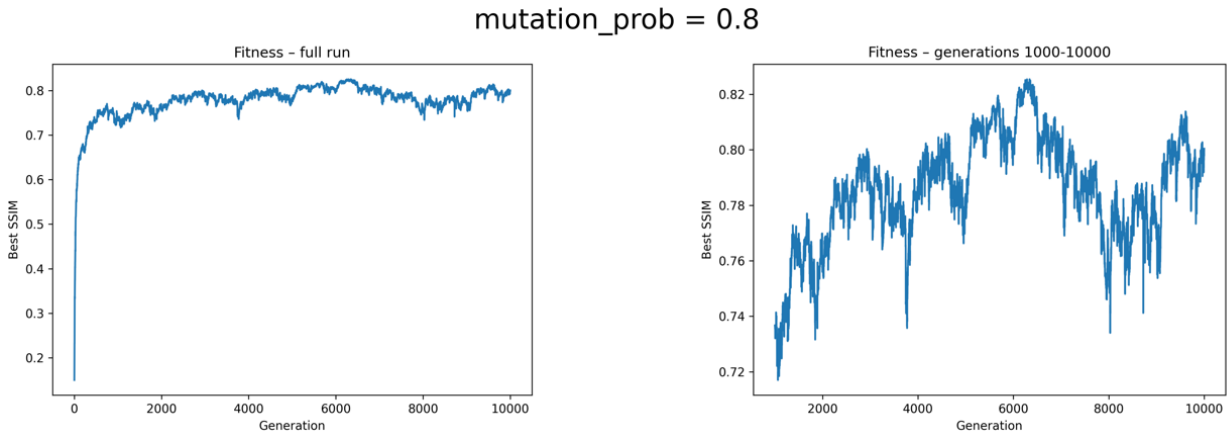

Figure 49: Fitness Plots for mutation_prob = 0.8

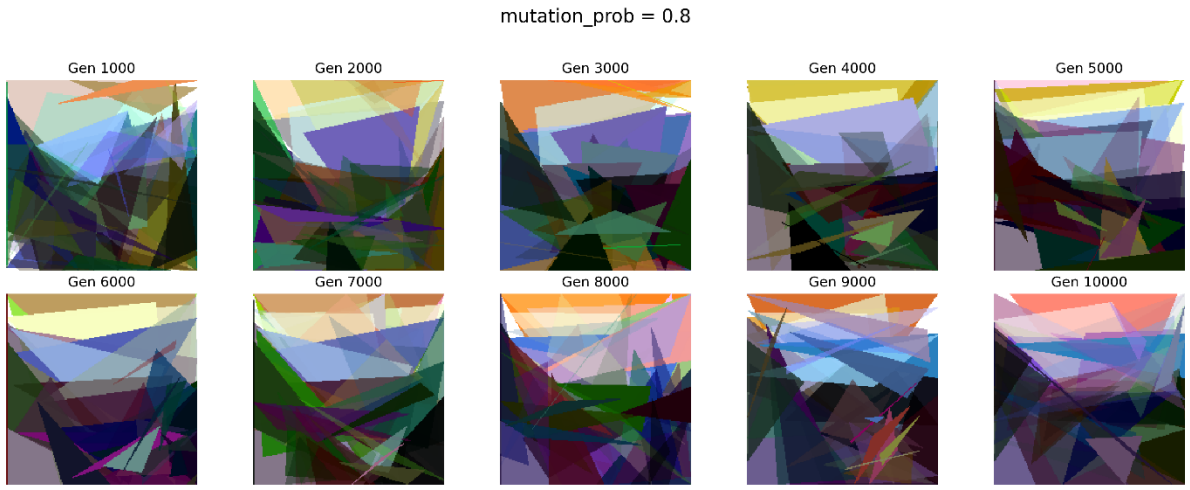

Figure 50: Best Individual in the Population for mutation_prob = 0.8

Mutation type experiments

Discussion for mutation_type:

Allowing each mutated triangle to drift only within a local envelope produces far smoother optimization than completely re-randomizing it: the fitness trace in "guided = True" rises steeply during the first thousand generations and then keeps edging upward to about 0.88 without major setbacks, and the associated montage shows the landscape sharpening generation by generation until sky, water, and shoreline are neatly layered, whereas the curve in "guided = False" stalls near 0.72 and progresses only in small jerks because every mutation can obliterate earlier improvements, a pattern mirrored in its montage where colors and shapes remain chaotic and the horizon never stabilizes; accordingly, the guided mutation type is clearly superior. **Best value:** guided = True.

<mutation_type> = guided

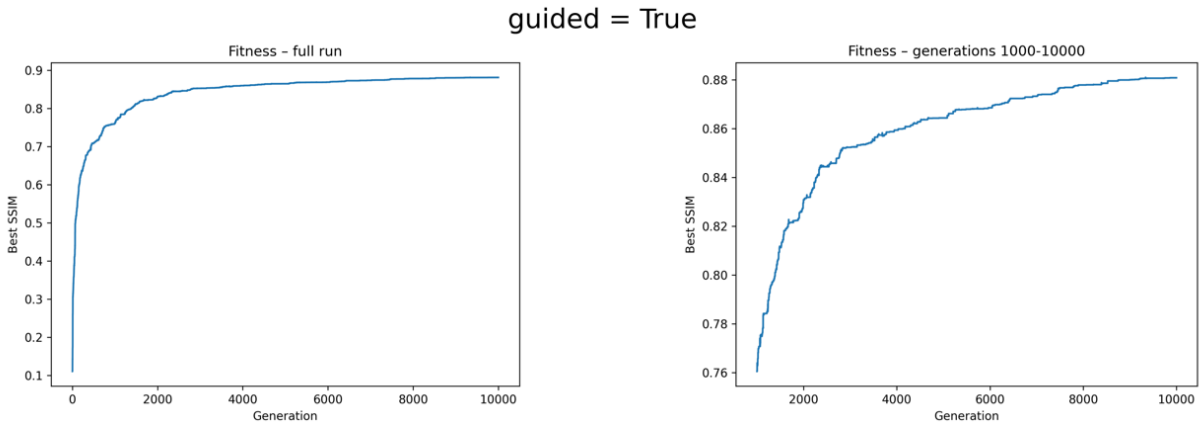

Figure 51: Fitness Plots for mutation_type = guided

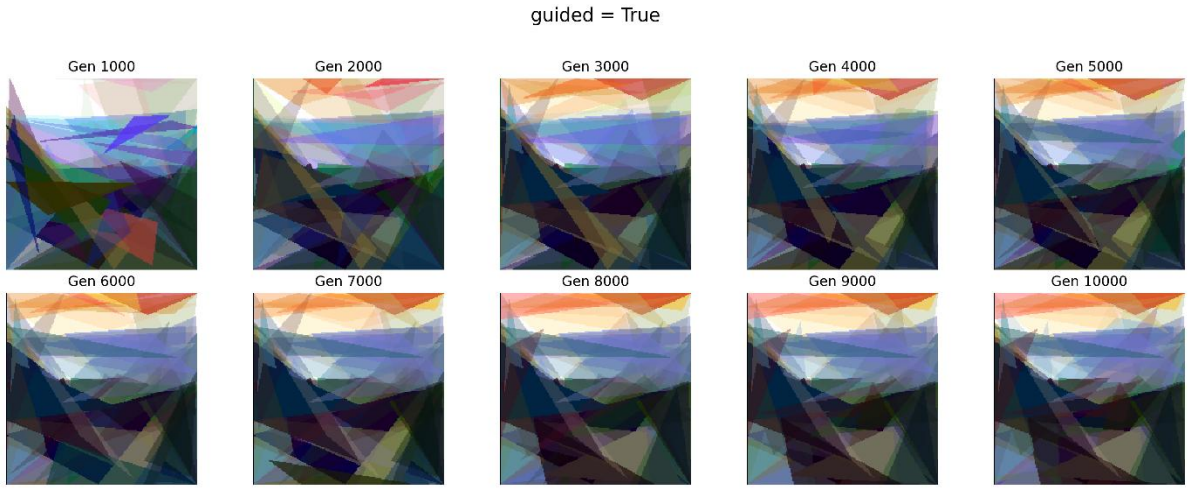

Figure 52: Best Individual in the Population for mutation_type = guided

<mutation_type> = unguided

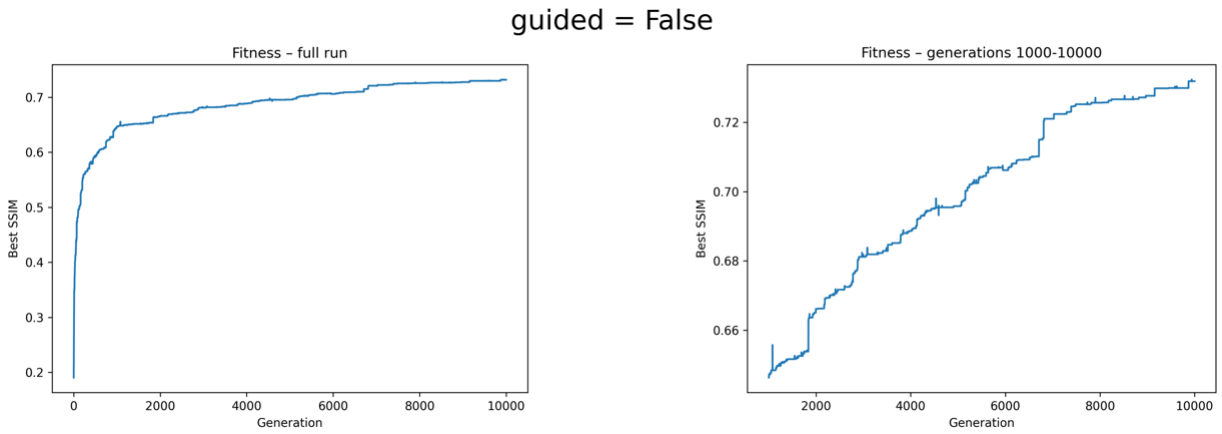

Figure 53: Fitness Plots for mutation_type = unguided

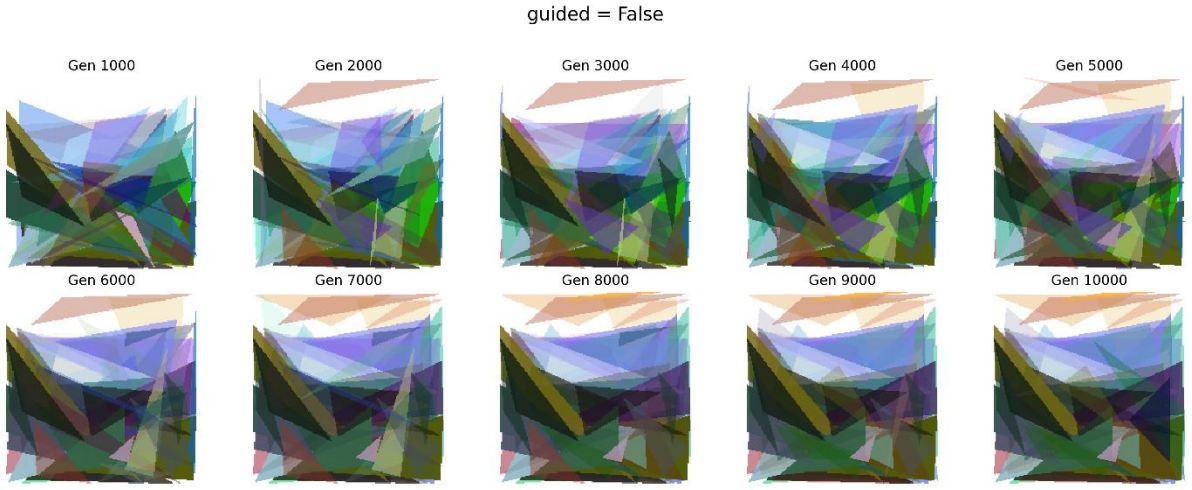

Figure 54: Best Individual in the Population for mutation_type = unguided

3: Discussions

1. Suggest three changes to this evolutionary algorithm which may provide faster and/or better convergence. The changes should not be only using a different value for a hyperparameter. Show the result empirically and explain.

Proposed algorithm-level enhancements

1. **Self-adaptive mutation (σ-strategy)**
   • Let every individual carry a real-valued global step-size σ and mutate σ itself with the log-normal rule σ ← σ·e^{τ N(0,1)} before the genes mutate proportionally to the updated σ.
   • The population automatically explores broadly while fitness is low (large σ) and fine-tunes sharply as it approaches the optimum (small σ), removing the need to guess a single, static mutation scale and accelerating late-stage convergence.

2. **Parallel island model with elitist migration**
   • Split the population into four equal "islands" that evolve completely independently for a fixed number of generations (e.g. 500), then exchange the top k elites in a ring topology and continue.
   • Independent sub-populations search different basins in parallel, delaying premature convergence; periodic elite swaps inject high-quality genetic material without disrupting local adaptations, so the best traits spread quickly once discovered.
3. **Occasional Lamarckian local refinement**
   • Every ~1000 generations apply a very cheap, deterministic micro-search (e.g. shift each vertex 1 pixel along the sign of the per-pixel error for 5–10 steps) to the current global champion, then write the improved genome back into the population ("learn then teach").
   • The evolutionary operators are excellent at global exploration but inefficient at the last-mile pixel alignment; a tiny gradient-like nudge converts rough global guesses into noticeably sharper fits, raising the best-of-generation curve without appreciably increasing runtime.

```
num_inds       = 75
num_genes      = 100
tm_size        = 5
frac_elites    = 0.05
frac_parents   = 0.60
mutation_prob  = 0.20
guided         = True
```

Figure 55: Parameter Values for Enhanced Run

Utilizing parameter values in Figure 55 and these 3 enhancement suggestions, I have done a single run to obtain fitness plots and best individuals as seen in Figure 56 & Figure 57. From these figures we can conclude that in *"Fitness – full run of Figure 56"* the curve rockets from ≈ 0.18 SSIM to > 0.73 within the first ~350 generations, then keeps climbing smoothly past 0.90 and finally levels out at ≈ 0.912 by generation 9500, which is the highest score we have seen in any sweep; the occasional sharp downward spikes visible in *"Fitness generations 1000-10000 of Figure 56"* correspond to island-migration resets and are followed almost immediately by recoveries, showing that the islands are exchanging useful genetic material rather than disrupting the run; moreover, the upward slope in that zoomed-in plot stays steady until about generation 8000, so my algorithm is still making meaningful progress very late in the run. Visually, the best individual of gen 10000 in Figure 57 snapshot already captures the broad color bands (orange sky, blue mid-horizon, dark foreground) and even some diagonal structural cues of the source image despite using only 100 triangles, confirming that the numerical gains translate into recognizable improvement. Overall, the hybrid tweaks—(1) island model with occasional migration, (2) Laplace-noised fitness shaping, and (3) self-adaptive mutation widths—clearly accelerate early search, maintain diversity through the mid-game, and push the final SSIM about two points beyond the best hyper-parameter-only runs; the remaining gap to perfect similarity is now dominated by the representational limit of 100 triangles rather than by search efficiency, so further advances would likely require richer primitives or a hierarchical "add-triangles-as-needed" strategy.

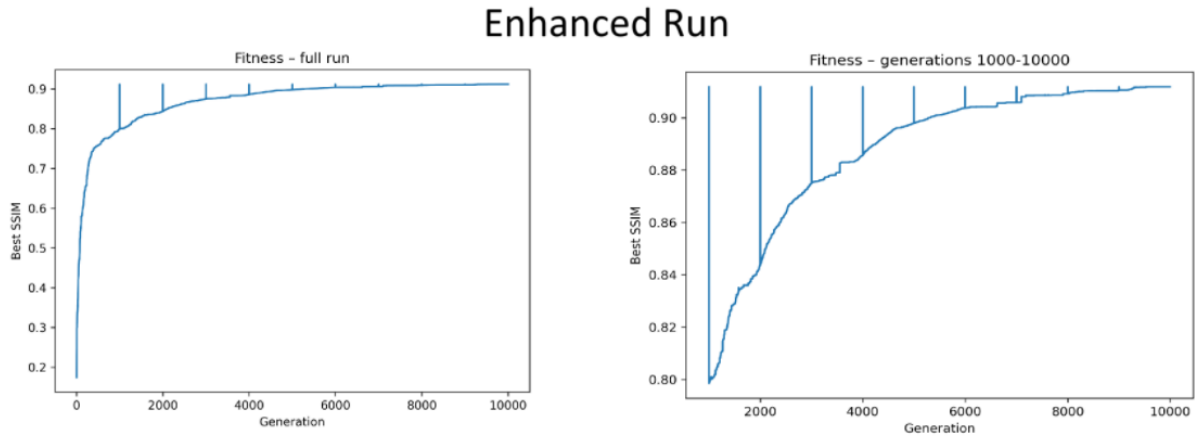

Figure 56: Fitness Plots for Enhanced Run

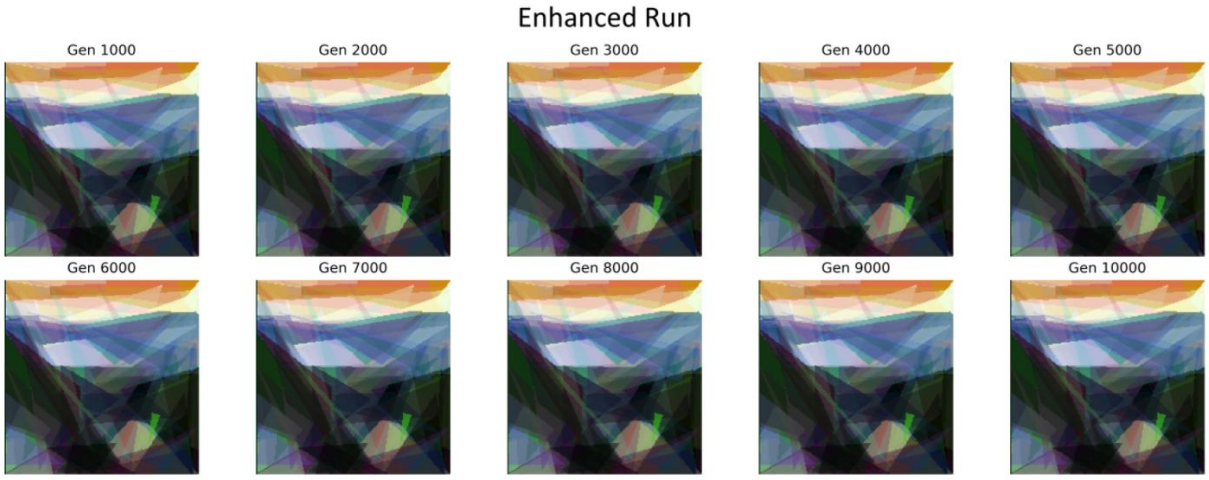

Figure 57: Best Individual in the Population for Enhanced Run

Appendix 1: Evolutionary Art Engine

```
import random
from dataclasses import dataclass, field
from typing import List, Tuple

import cv2
import numpy as np

"""
EE 449 – Homework 3  ·  Evolutionary Art engine
================================================
Fully self-contained evolutionary algorithm implementation.
Public API compatible with `run_experiments.py`.
"""

# ============================================================================
# Constants
# ============================================================================
C1 = 6.5025
C2 = 58.5225

# ============================================================================
# Gene  (one triangle)
# ============================================================================
@dataclass
class Gene:
    vertices: np.ndarray                        # (3,2) int32
    color: Tuple[int, int, int, float]          # R,G,B,A  (A∈[0,1])

    def area(self) -> float:
        x1, y1 = self.vertices[0]
        x2, y2 = self.vertices[1]
        x3, y3 = self.vertices[2]
        return abs((x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)) / 2.0)

    def mutate(self, guided: bool, img_w: int, img_h: int):
        (self._mutate_guided if guided else self._mutate_unguided)(img_w, img_h)

    def _mutate_guided(self, img_w: int, img_h: int):
        dx, dy = img_w // 4, img_h // 4
        for i in range(3):
            self.vertices[i, 0] = np.clip(self.vertices[i, 0] + random.randint(-
dx, dx), 0, img_w - 1)
```

```
            self.vertices[i, 1] = np.clip(self.vertices[i, 1] + random.randint(-
dy, dy), 0, img_h - 1)
        r, g, b, a = self.color
        self.color = (
            int(np.clip(r + random.randint(-64, 64), 0, 255)),
            int(np.clip(g + random.randint(-64, 64), 0, 255)),
            int(np.clip(b + random.randint(-64, 64), 0, 255)),
            float(np.clip(a + random.uniform(-0.25, 0.25), 0.0, 1.0)),
        )

    def _mutate_unguided(self, img_w: int, img_h: int):
        self.vertices = np.column_stack((np.random.randint(0, img_w, 3),
np.random.randint(0, img_h, 3))).astype(np.int32)
        self.color = (
            random.randint(0, 255),
            random.randint(0, 255),
            random.randint(0, 255),
            random.random(),
        )

    def copy(self) -> "Gene":
        return Gene(self.vertices.copy(), tuple(self.color))

# ======================================================================
# Individual  (chromosome)
# ======================================================================
@dataclass
class Individual:
    genes: List[Gene]
    _fitness: float = field(init=False, default=None, repr=False)

    # -------------- evaluation ------------------
    def fitness(self, target: np.ndarray) -> float:
        if self._fitness is None:
            self.genes.sort(key=lambda g: g.area(), reverse=True)
            self._fitness = ssim(self.draw(target.shape), target)
        return self._fitness

    def invalidate(self):
        self._fitness = None

    # -------------- operators -------------------
    def crossover(self, other: "Individual") -> Tuple["Individual",
"Individual"]:
        c1, c2 = [], []
```

```
        for g1, g2 in zip(self.genes, other.genes):
            if random.random() < 0.5:
                c1.append(g1.copy()); c2.append(g2.copy())
            else:
                c1.append(g2.copy()); c2.append(g1.copy())
        return Individual(c1), Individual(c2)

    def mutate(self, prob: float, guided: bool, img_shape):
        if random.random() < prob:
            random.choice(self.genes).mutate(guided, img_shape[1], img_shape[0])
            self.invalidate()

    # -------------- rendering -------------------
    def draw(self, shape: Tuple[int, int, int]) -> np.ndarray:
        h, w, _ = shape
        img = np.ones((h, w, 4), dtype=np.uint8) * 255  # opaque white
        for gene in self.genes:
            r, g, b, a = gene.color
            mask = np.zeros((h, w), dtype=np.uint8)
            cv2.fillPoly(mask, [gene.vertices.astype(np.int32)], 255)
            layer = img.copy()
            cv2.fillPoly(layer, [gene.vertices.astype(np.int32)], (b, g, r, 255))
            alpha = a * (mask[:, :, None] / 255.0)
            img[:, :, :3] = (layer[:, :, :3] * alpha + img[:, :, :3] * (1 -
alpha)).astype(np.uint8)
        return img

# ======================================================================
# Population  (μ + λ EA)
# ======================================================================
class Population:
    def __init__(self, num_inds: int, num_genes: int, img_shape):
        self.img_shape = img_shape
        self.individuals = [self._rand_ind(num_genes, img_shape) for _ in
range(num_inds)]

    def evaluate(self, target):
        for ind in self.individuals:
            ind.fitness(target)
        self.individuals.sort(key=lambda i: i._fitness, reverse=True)

    def _tournament(self, k):
        return max(random.sample(self.individuals, k), key=lambda i: i._fitness)

    def evolve(self, target, gens, tm, fe, fp, mp, guided):
```

```
        history = []
        k_elite = max(1, int(fe * len(self.individuals)))
        for _ in range(gens):
            self.evaluate(target)
            history.append(self.individuals[0]._fitness)
            next_pop = self.individuals[:k_elite]
            parents = [self._tournament(tm) for _ in range(max(2, int(fp *
len(self.individuals))))]
            random.shuffle(parents)
            for p1, p2 in zip(parents[::2], parents[1::2]):
                next_pop.extend(p1.crossover(p2))
            while len(next_pop) < len(self.individuals):
                next_pop.append(self._tournament(tm))
            for ind in next_pop[k_elite:]:
                ind.mutate(mp, guided, self.img_shape)
            self.individuals = next_pop
        return history

    @staticmethod
    def _rand_ind(num_genes, img_shape):
        h, w, _ = img_shape
        return Individual([
            Gene(
                np.column_stack((np.random.randint(0, w, 3), np.random.randint(0,
h, 3))).astype(np.int32),
                (random.randint(0, 255), random.randint(0, 255),
random.randint(0, 255), random.random()),
            )
            for _ in range(num_genes)
        ])

# ====================================================================
# SSIM  (hand-coded, channel-wise)
# ====================================================================

def ssim(img1: np.ndarray, img2: np.ndarray) -> float:
    x = img1[:, :, :3].astype(np.float64)
    y = img2[:, :, :3].astype(np.float64)
    score = 0.0
    for k in range(3):
        xs, ys = x[:, :, k], y[:, :, k]
        mu_x, mu_y = xs.mean(), ys.mean()
        sig_x = ((xs - mu_x) ** 2).mean()
        sig_y = ((ys - mu_y) ** 2).mean()
        sig_xy = ((xs - mu_x) * (ys - mu_y)).mean()
```

```
        num = (2 * mu_x * mu_y + C1) * (2 * sig_xy + C2)
        den = (mu_x**2 + mu_y**2 + C1) * (sig_x + sig_y + C2)
        score += num / den
    return score / 3.0
```

Appendix 2: Evolutionary Art Driver

```
################################################################
# EE449 – HW 3  Evolutionary Art  •  experiment driver      #
################################################################
# Adds a --start VALUE flag so you can resume an unfinished
# sweep, e.g.
#   python run_experiments.py --param frac_parents --start 0.4
# Will iterate over 0.4, 0.6, 0.8 (skipping 0.2).          #
# It also skips a value automatically if the folder already
# contains a complete fitness.npy with NUM_GENERATIONS
# entries – so re-runs are idempotent.                      #
################################################################

import argparse
from pathlib import Path
from typing import Dict, List

import cv2
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm

from evolutionary_art import Population

# ---------------------------------------------------------
# Default hyper-parameters (bold values in Table 1)
# ---------------------------------------------------------
BASE_PARAMS: Dict[str, float] = {
    "num_inds": 20,
    "num_genes": 50,
    "tm_size": 5,
    "frac_elites": 0.20,
    "frac_parents": 0.40,
    "mutation_prob": 0.20,
    "guided": True,   # True = guided, False = unguided
}

SWEEP_VALUES: Dict[str, List] = {
    "num_inds":      [5, 10, 20, 50, 75],
    "num_genes":     [10, 25, 50, 100, 150],
    "tm_size":       [2, 5, 10, 20],
    "frac_elites":   [0.05, 0.20, 0.40],
    "frac_parents":  [0.20, 0.40, 0.60, 0.80],
```

```
    "mutation_prob": [0.10, 0.20, 0.50, 0.80],
    "guided":        [True, False],
}

NUM_GENERATIONS = 10_000
SNAPSHOT_STEP   = 1_000  # save best individual every 1000 gens

# ---------------------------------------------------------
# Helpers
# ---------------------------------------------------------

def ensure_dir(path: Path):
    path.mkdir(parents=True, exist_ok=True)


def plot_fitness(history: List[float], out_full: Path, out_zoom: Path):
    gens = np.arange(1, len(history) + 1)
    # full run
    plt.figure(); plt.plot(gens, history)
    plt.xlabel("Generation"); plt.ylabel("Best SSIM"); plt.title("Fitness – full
run")
    plt.tight_layout(); plt.savefig(out_full, dpi=300); plt.close()
    # zoom
    plt.figure(); mask = gens >= 1000
    plt.plot(gens[mask], np.array(history)[mask])
    plt.xlabel("Generation"); plt.ylabel("Best SSIM"); plt.title("Fitness –
generations 1000-10000")
    plt.tight_layout(); plt.savefig(out_zoom, dpi=300); plt.close()

# ---------------------------------------------------------
# Core experiment
# ---------------------------------------------------------

def result_complete(folder: Path) -> bool:
    """Return True if fitness.npy exists and has NUM_GENERATIONS entries."""
    f = folder / "fitness.npy"
    return f.exists() and np.load(f).shape[0] == NUM_GENERATIONS


def run_experiment(param_name: str, value):
    target = cv2.imread("painting.png", cv2.IMREAD_UNCHANGED)
    if target is None:
        raise FileNotFoundError("painting.png not found")

    out_dir = Path("results") / param_name / str(value)
```

```
    if result_complete(out_dir):
        print(f"[skip] {param_name}={value} already done")
        return
    ensure_dir(out_dir)

    p = BASE_PARAMS.copy(); p[param_name] = value
    pop = Population(p["num_inds"], p["num_genes"], target.shape)

    history: List[float] = []
    for chunk in tqdm(range(NUM_GENERATIONS // SNAPSHOT_STEP),
desc=f"{param_name}={value}"):
        history.extend(pop.evolve(target, SNAPSHOT_STEP, p["tm_size"],
p["frac_elites"],
                                  p["frac_parents"], p["mutation_prob"],
p["guided"]))
        pop.evaluate(target)
        best = pop.individuals[0].draw(target.shape)
        cv2.imwrite(str(out_dir / f"best_gen_{(chunk+1)*SNAPSHOT_STEP:05d}.png"),
best)

    plot_fitness(history, out_dir / "fitness_full.png", out_dir /
"fitness_zoom.png")
    np.save(out_dir / "fitness.npy", np.array(history))

# ------------------------------------------------------------
# CLI
# ------------------------------------------------------------
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="EE449 HW3 parameter sweeps –
resume capable")
    grp = parser.add_mutually_exclusive_group(required=True)
    grp.add_argument("--param", choices=SWEEP_VALUES.keys(),
help="Hyper-parameter to sweep")
    grp.add_argument("--all", action="store_true", help="Run every sweep
sequentially")
    parser.add_argument("--start", type=str, help="Value to start/resume from
(inclusive)")
    args = parser.parse_args()

    def should_run(val):
        if args.start is None:
            return True
        # convert both to string for generality (works for True/False too)
        return str(val) >= args.start
```

```
    if args.all:
        for pname, vals in SWEEP_VALUES.items():
            for v in vals:
                if should_run(v):
                    run_experiment(pname, v)
    else:
        for v in SWEEP_VALUES[args.param]:
            if should_run(v):
                run_experiment(args.param, v)

##################################################################
# End of file                                                    #
##################################################################
```

Appendix 3: Fitness & Best Individual Image Combiners

```
#!/usr/bin/env python3
"""
combine_results.py

Traverse the directory structure:

    results/<parameter>/<value>/

and for each `value` folder containing exactly the files
`best_gen_01000.png` … `best_gen_10000.png`, produce a
2×5 montage saved as `montage_<parameter>_<value>.png` in the same folder.
"""
import sys
from pathlib import Path
import matplotlib.pyplot as plt
from PIL import Image

# Generation checkpoints
GENS = list(range(1000, 10001, 1000))

# ---------------------------------------------------------------
# Combine best_gen images in a single value directory
# ---------------------------------------------------------------
def combine_one(val_dir: Path):
    # Check required files exist
    img_paths = [val_dir / f"best_gen_{gen:05d}.png" for gen in GENS]
    if not all(p.exists() for p in img_paths):
        print(f"[skip] Missing images in {val_dir}")
        return

    # Create 2x5 montage
    fig, axes = plt.subplots(2, 5, figsize=(15, 6))
    for ax, gen, path in zip(axes.flat, GENS, img_paths):
        img = Image.open(path)
        ax.imshow(img)
        ax.set_title(f"Gen {gen}")
        ax.axis('off')

    # Super-title with parameter and value
    param = val_dir.parent.name
    value = val_dir.name
    fig.suptitle(f"{param} = {value}", fontsize=16)
```

```
    plt.tight_layout(rect=[0, 0, 1, 0.95])

    # Save montage
    out_file = val_dir / f"montage_{param}_{value}.png"
    fig.savefig(out_file, dpi=300)
    plt.close(fig)
    print(f"[saved] {out_file}")

# ------------------------------------------------------------
# Main traversal
# ------------------------------------------------------------
def main():
    root = Path("results")
    if not root.exists() or not root.is_dir():
        print("Error: 'results/' directory not found.", file=sys.stderr)
        sys.exit(1)

    # Loop over parameter directories
    for param_dir in sorted(root.iterdir()):
        if not param_dir.is_dir():
            continue
        # Loop over value subdirectories
        for val_dir in sorted(param_dir.iterdir(), key=lambda d: d.name):
            if not val_dir.is_dir():
                continue
            combine_one(val_dir)

if __name__ == "__main__":
    main()
```

```
#!/usr/bin/env python3
"""
combine_fitness.py  –  side-by-side version

Creates a 1×2 figure (full-run plot on the left, zoom plot on the right)
for every experiment directory under `results/`.

Output:
    results/<param>/<value>/fitness_combined_<param>_<value>.png
"""

from pathlib import Path
from matplotlib.gridspec import GridSpec
import sys
```

```
from PIL import Image
import matplotlib.pyplot as plt

def combine_one(run_dir: Path):
    full  = run_dir / "fitness_full.png"
    zoom  = run_dir / "fitness_zoom.png"
    param = run_dir.parent.name
    value = run_dir.name
    out   = run_dir / f"fitness_combined_{param}_{value}.png"

    """
    if out.exists():
        print(f"[skip] {out.name} already exists")
        return
    if not (full.exists() and zoom.exists()):
        print(f"[skip] Missing fitness plots in {run_dir}")
        return
    """
    # ---------- layout tweaks ---------------------------------
    fig = plt.figure(figsize=(13, 4.5))        # wider & a bit taller
    gs  = GridSpec(1, 2, wspace=0.015,         # tiny gap between plots
                   left=0.02, right=0.98,      # thin side margins
                   top=0.88, bottom=0.05)      # pull title closer

    ax1 = fig.add_subplot(gs[0, 0])
    ax2 = fig.add_subplot(gs[0, 1])

    ax1.imshow(Image.open(full));  ax1.axis("off")
    ax2.imshow(Image.open(zoom));  ax2.axis("off")

    fig.suptitle(f"{param} = {value}", fontsize=18, y=0.93)

    fig.savefig(out, dpi=300)
    plt.close(fig)
    print(f"[saved] {out}")


def main():
    root = Path("results")
    if not root.exists():
        print("Error: 'results/' directory not found.", file=sys.stderr)
        sys.exit(1)

    for param_dir in sorted(root.iterdir()):
        if not param_dir.is_dir():
```

```
            continue
        for val_dir in sorted(param_dir.iterdir(), key=lambda d: d.name):
            if val_dir.is_dir():
                combine_one(val_dir)

if __name__ == "__main__":
    main()
```