Week-4: Jigsaw Puzzle Solving using Simulated Annealing

Chetankumar Kamani (20251603005), Sakariya Devraj (20251603006), Divyesh Dodiya (20251603007),
M.Tech Sem-1, IIIT Vadodara

I. Problem Statement

The objective is to reconstruct a scrambled grayscale image (a 4×4 grid of tiles from the 512×512 **Lena** image) using the **Simulated Annealing (SA)** metaheuristic. The search state is represented as a permutation of tiles (and optionally their rotations). The cost function measures the sum of pixel-intensity mismatches between adjacent tile edges. The SA algorithm iteratively minimizes this cost to approximate the correct arrangement.

II. Python Code

```python
import re
import numpy as np
import matplotlib.pyplot as plt


# Loading an Image (Octave/MATLAB ASCII .mat containing a 2D uint8 array)
def loading_image_from_matlab_file(path): # returns 2D uint8 array
    with open(path, "rb") as f:
        raw = f.read().decode("latin1")
    m = re.search(r"# ndims:\s*(\d+)", raw)
    if not m:
        raise ValueError("Cannot find '# ndims:'")
    ndims = int(m.group(1))
    if ndims != 2:
        raise ValueError(f"Expected ndims=2, got {ndims}")
    after = raw.split("# ndims:", 1)[1].splitlines()
    sizes_line = after[1].strip()
    H, W = map(int, sizes_line.split()[:2])
    rest = "\n".join(after[2:])
    # Remove the two dimension numbers that Octave writes at the top of the data block
    for s in (H, W):
        rest = re.sub(r"^\s*" + re.escape(str(s)), "", rest.lstrip(), count=1)
    vals = np.fromstring(rest, sep=" ", dtype=np.int64)
    if vals.size != H * W:
        raise ValueError(f"Value count {vals.size} != H*W {H*W}")
    return vals.astype(np.uint8).reshape((H, W))

# Split image into GRID x GRID tiles (row-major)
def slice_tiles(img, grid):
    H, W = img.shape
    assert H % grid == 0 and W % grid == 0, "Image must be divisible by GRID"
    th, tw = H // grid, W // grid
    tiles = []
    for r in range(grid):
        for c in range(grid):
            tiles.append(img[r*th:(r+1)*th, c*tw:(c+1)*tw])
    return np.array(tiles), th, tw

# Precompute tile edges
def precompute_edges(tiles):
    top = np.array([t[0, :]  for t in tiles]) # top edges
    bot = np.array([t[-1, :] for t in tiles]) # bottom edges
    lef = np.array([t[:, 0]  for t in tiles]) # left edges
    rig = np.array([t[:, -1] for t in tiles]) # right edges
    return top, bot, lef, rig

# Board adjacency (right & down neighbors as position pairs)
def build_right_down_pairs(grid):
    right_pairs, down_pairs = [], []
    for r in range(grid): # rows
        for c in range(grid): # cols
            i = r*grid + c # position index
            if c+1 < grid: # right neighbor
                right_pairs.append((i, i+1)) # (left, right)
            if r+1 < grid: # down neighbor
                down_pairs.append((i, i+grid)) # (up, down)
    return np.array(right_pairs, np.int32), np.array(down_pairs, np.int32) # (a,b) pairs

# For fast  E  when swapping two positions: edges touching each position
def build_adjacency_for_delta(grid):
    rp, dp = build_right_down_pairs(grid) # right & down pairs
    adj = [[] for _ in range(grid*grid)] # adjacency list
    for a, b in rp:
        adj[a].append(('R', a, b))
        adj[b].append(('R', a, b))
    for a, b in dp:
        adj[a].append(('D', a, b))
        adj[b].append(('D', a, b))
    return rp, dp, adj

# Pairwise edge mismatch costs: i(left/up) vs j(right/down)
def pair_cost_mats(top, bot, lef, rig):
    N = top.shape[0]
    costR = np.zeros((N, N), dtype=np.float32)
    costD = np.zeros((N, N), dtype=np.float32)
    for i in range(N):
        ri = rig[i].astype(np.int32)
        bi = bot[i].astype(np.int32)
        for j in range(N):
```

```
            costR[i, j] = np.mean(np.abs(ri - lef[
                j].astype(np.int32)))
            costD[i, j] = np.mean(np.abs(bi - top[
                j].astype(np.int32)))
    return costR, costD

# Total board energy for a permutation
def board_cost(perm, costR, costD, right_pairs,
    down_pairs):
    s = 0.0
    for a, b in right_pairs:
        s += costR[perm[a], perm[b]]
    for a, b in down_pairs:
        s += costD[perm[a], perm[b]]
    return s

#  E  if we swap tiles at board positions i and j
def delta_for_swap(perm, i, j, costR, costD,
    adj_list):
    affected = set()
    for info in adj_list[i]:
        affected.add(info)
    for info in adj_list[j]:
        affected.add(info)

    before = 0.0
    after  = 0.0
    ti, tj = perm[i], perm[j]

    def tile_at(pos):
        if pos == i: return tj
        if pos == j: return ti
        return perm[pos]

    for kind, a, b in affected:
        ta_before, tb_before = perm[a], perm[b]
        if kind == 'R':
            before += costR[ta_before, tb_before]
            ta_after, tb_after = tile_at(a),
                tile_at(b)
            after  += costR[ta_after, tb_after]
        else:  # 'D'
            before += costD[ta_before, tb_before]
            ta_after, tb_after = tile_at(a),
                tile_at(b)
            after  += costD[ta_after, tb_after]

    return after - before

# Rebuild the full image from tiles according to a
     permutation
def compose_image(tiles, perm, th, tw, GRID):
    """Assemble full image from tiles by
        permutation."""
    H, W = th * GRID, tw * GRID
    out = np.zeros((H, W), dtype=tiles.dtype)
    for pos, tid in enumerate(perm):
        r, c = divmod(pos, GRID)
        out[r*th:(r+1)*th, c*tw:(c+1)*tw] = tiles[
            tid]
    return out

# ---------- Simulated Annealing (SA) ---------

def simulated_annealing(
    costR, costD, right_pairs, down_pairs,
        adj_list, N,
    iters=150_000,      # tuned for 4x4
    alpha=0.9993,       # slow cooling
    seed=0,
    init_perm=None,
    early_stop_zero=True,
    record_trace=False
):
    rng = np.random.default_rng(seed)

    # Initial state
    if init_perm is None:
        cur = np.arange(N); rng.shuffle(cur)
    else:
        cur = init_perm.copy()

    cur_cost = board_cost(cur, costR, costD,
        right_pairs, down_pairs)
    best, best_cost = cur.copy(), cur_cost

    # Adaptive initial temperature from local  E
        samples
    deltas = []
    for _ in range(300):
        i, j = rng.choice(N, size=2, replace=False
            )
        deltas.append(delta_for_swap(cur, i, j,
            costR, costD, adj_list))
    T = 3.0 * (np.std(deltas) + 1e-6)

    best_trace = []
    if record_trace:
        best_trace.append(float(best_cost))

    # SA loop
    for _ in range(iters):
        i, j = rng.integers(0, N, size=2)
        while j == i:
            j = rng.integers(0, N)

        d = delta_for_swap(cur, i, j, costR, costD
            , adj_list)

        # Metropolis acceptance
        if d <= 0 or rng.random() < np.exp(-d /
            max(T, 1e-12)):
            cur[i], cur[j] = cur[j], cur[i]
            cur_cost += d
            if cur_cost < best_cost:
                best, best_cost = cur.copy(),
                    cur_cost
                if early_stop_zero and best_cost
                    <= 1e-9:  # near-perfect
                    if record_trace:
                        best_trace.append(float(
                            best_cost))
                    break
        T *= alpha
        if record_trace:
            best_trace.append(float(best_cost))

    return (best, best_cost, best_trace) if
        record_trace else (best, best_cost, None)

# ==============================
# STEP-1: STATE-SPACE FORMULATION (model)
#   - State: permutation of tile indices (length N
    )
#   - Initial state: scrambled image (tiles from
    img)
#   - Actions: swap two positions
#   - Cost: sum of right/down edge mismatches
#   - Goal: minimize cost
# ================================

path = "scrambled_lena.mat"  # Input image path
img  = loading_image_from_matlab_file(path)  #
    Loaded as 2D uint8 array

GRID = 4                                   # 4x4
    puzzle
tiles, th, tw = slice_tiles(img, GRID)     # tiles
     from scrambled image
top, bot, lef, rig = precompute_edges(tiles)
```

```python
right_pairs, down_pairs = build_right_down_pairs(
    GRID)
_, _, adj_list = build_adjacency_for_delta(GRID)

costR, costD = pair_cost_mats(top, bot, lef, rig)

N = GRID * GRID
identity = np.arange(N)

# Optional diagnostics:
# print("Min edge mismatch (R):", float(costR.min
    ()))
# print("Min edge mismatch (D):", float(costD.min
    ()))

# ==================================
# STEP-2: SIMULATED ANNEALING SEARCH (solver)
#   - Multi-restart SA (identity & random starts)
#   - Keep global best permutation
#   - Reconstruct & save final image
# ===================================

restarts = 12
rng = np.random.default_rng(7)

global_best_cost  = np.inf
global_best_perm  = identity.copy()
global_best_trace = None

for r in range(restarts):
    init = identity.copy() if (r % 2 == 0) else
        rng.permutation(N)
    perm, best_cost, trace = simulated_annealing(
        costR, costD, right_pairs, down_pairs,
            adj_list, N,
        iters=200_000, alpha=0.9997, seed=1020 + r
            ,
        init_perm=init, early_stop_zero=True,
            record_trace=True
    )
    if best_cost < global_best_cost:
        global_best_cost  = best_cost
        global_best_perm  = perm.copy()
        global_best_trace = trace[:] if trace is
            not None else None
    if global_best_cost <= 1e-9:
        break

# Compose and save outputs
recon = compose_image(tiles, global_best_perm, th,
     tw, GRID=GRID)
print("Best cost:", float(global_best_cost))

plt.figure(); plt.imshow(img, cmap="gray"); plt.
    title("Input (Scrambled)"); plt.axis("off")
plt.tight_layout(); plt.savefig("scrambled.png",
    dpi=150)

plt.figure(); plt.imshow(recon, cmap="gray"); plt.
    title("Reconstructed (SA, 4x4)"); plt.axis("
    off")
plt.tight_layout(); plt.savefig("reconstructed.png
    ", dpi=150)

np.savetxt("best_perm.txt", np.array(
    global_best_perm, dtype=np.int32), fmt="%d")
print("Saved: reconstructed.png, scrambled.png,
    best_perm.txt")

# Optional: SA progress curve
if global_best_trace is not None and len(
    global_best_trace) > 0:
    plt.figure()
    plt.plot(global_best_trace)
    plt.xlabel("Iteration"); plt.ylabel("Best Cost
        ")
    plt.title("Simulated Annealing Progress (Best
        Restart)")
    plt.tight_layout()
    plt.savefig("sa_cost.png", dpi=150)
    print("Saved: sa_cost.png")
```

III. Input File Details

File: scrambled_lena.mat

- Format: Octave/MATLAB ASCII file containing a 2D uint8 grayscale matrix.
- Dimensions: 512×512
- Description: Scrambled Lena image divided into a 4×4 grid (each tile 128×128 pixels).

IV. Best Permutation (Recovered Tile Order)

The following permutation (0-indexed) represents the optimal arrangement of tiles found by the simulated annealing solver:

```
6 0 14 5 13 2 7 8 12 10 4 3 15 9 11 1
```

This sequence defines the mapping from the scrambled order to the reconstructed order. For example, tile 6 in the scrambled image occupies position 0 in the final image.

V. Execution Instructions

```
pip install numpy matplotlib
python jigsaw_puzzle.py
# Outputs generated:
#   scrambled.png
#   reconstructed.png
#   best_perm.txt
#   sa_cost.png
```

VI. Experimental Results

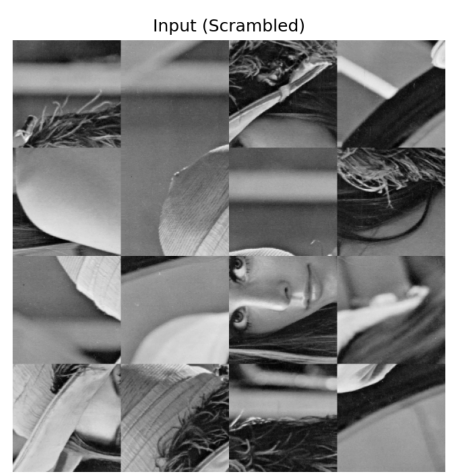

Fig. 1. Scrambled Lena image (input, 4x4 grid).

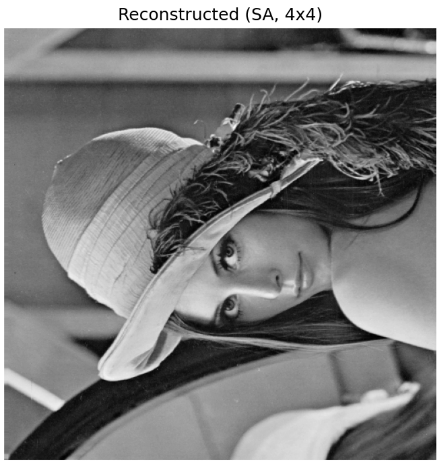

Fig. 2. Reconstructed image using Simulated Annealing.

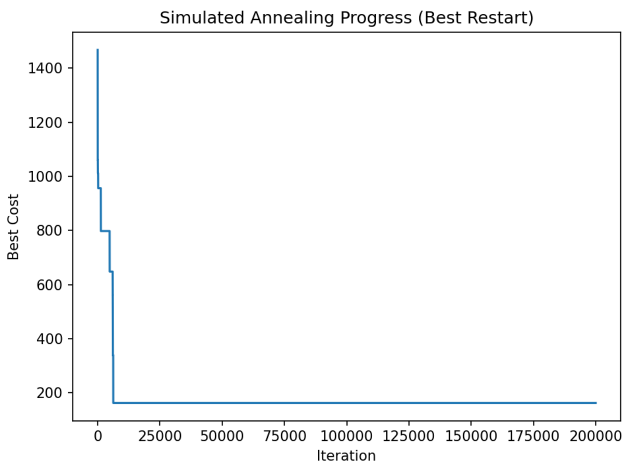

Fig. 3. Convergence curve: best cost vs. iteration.

VII. References

The implementation and design were guided by the following open-source and academic references:

1) MATLAB Central File Exchange: Jigsaw Puzzle Reconstruction
https://in.mathworks.com/matlabcentral/fileexchange/45547-jigsaw-puzzle
2) Nithyananda Bhat, "Project Report – Jigsaw Puzzle Reconstruction"
https://nithyanandabhat.weebly.com/uploads/4/5/6/1/45617813/project_report-jigsaw-puzzle.pdf
3) Goktug's GitHub Repository: Simulated Annealing for 8-Queens
https://github.com/Goktug/8queens-simulated-annealing-python
4) Visual Studio GitHub Copilot – Code Optimization Assistance
https://visualstudio.microsoft.com/github-copilot/
5) M. Noor Fawi, Python Simulated Annealing Gist
https://gist.github.com/MNoorFawi/4dcf29d69e1708cd60405bd2f0f55700
6) YouTube Tutorial: Simulated Annealing Explained (7JSttolQ0VY)
https://www.youtube.com/watch?v=7JSttolQ0VY&t=1s

VIII. Code Availability

- The complete source code is available at: GitHub Repository (CS659 – AI Laboratory).
- **GitHub Repository:**
https://github.com/ChetanKamani/CS659-LAB-TASK