IIIT Vadodara

M.Tech (CSE – AI Specialization), Semester–I

CS659 – Artificial Intelligence Laboratory

Lab Report for the First Four Problems

Course Code: CS659

Course Name: Artificial Intelligence Laboratory

Submitted by:

Chetankumar Kamani (20251603005)
Sakariya Devraj (20251603006)
Divyesh Dodiya (20251603007)

GitHub Repository:

https://github.com/ChetanKamani/CS659-AI-Lab

October 5, 2025

Week-1: Rabbit Leap — BFS and DFS State-Space Search

Chetankumar Kamani (20251603005), Sakariya Devraj (20251603006), and Divyesh Dodiya (20251603007)

I. Problem Statement

In the Rabbit Leap problem, three east-bound rabbits stand in a line blocked by three west-bound rabbits. They are crossing a stream using stones placed in a straight east–west direction. There is one empty stone between them. Each rabbit can only move forward (in its facing direction) by one or two stones at a time. A rabbit may jump over exactly one rabbit if needed, but not over two. The goal is to determine whether the rabbits can successfully cross each other without stepping into the water.

II. State Space Search

State representation: Each state is represented as a string of length 7 containing three east-bound rabbits ($>$), three west-bound rabbits ($<$), and one empty position (_).
Initial state: >>>_<<<
Goal state: <<<_>>>

Valid Moves

- $>$ can move one step right into the empty position.
- $<$ can move one step left into the empty position.
- $>$ can jump right over one $<$ into the empty position.
- $<$ can jump left over one $>$ into the empty position.

Search Space Size

- The total number of possible states is:

$$\text{Total possible states} = \frac{7!}{3!\,3!\,1!} = \frac{5040}{36} = 140$$

- Reachable states under valid moves $= 72$

III. Breadth-First Search (BFS) Solution

Breadth-First Search explores level by level, so the first solution found is optimal (i.e., it uses the minimum number of moves).

A. *Python Code (BFS)*

```python
from collections import deque

def next_states(state):
    s = list(state)
    i = s.index('_')
    # adjacent moves
    if i - 1 >= 0 and s[i - 1] == '>':
        t = s.copy(); t[i], t[i - 1] = t[i - 1], t[i]
        yield ('> moves 1 right', ''.join(t))
    if i + 1 < len(s) and s[i + 1] == '<':
        t = s.copy(); t[i], t[i + 1] = t[i + 1], t[i]
        yield ('< moves 1 left', ''.join(t))
    # jumps over exactly one opponent
    if i - 2 >= 0 and s[i - 2] == '>' and s[i - 1] == '<':
        t = s.copy(); t[i], t[i - 2] = t[i - 2], t[i]
        yield ('> jumps over <', ''.join(t))
    if i + 2 < len(s) and s[i + 2] == '<' and s[i + 1] == '>':
        t = s.copy(); t[i], t[i + 2] = t[i + 2], t[i]
        yield ('< jumps over >', ''.join(t))

def bfs(start=">>>_<<<", goal="<<<_>>>"):
    q = deque([start])
    parent = {start: None}
    action = {}
    seen = {start}
    while q:
        u = q.popleft()
        if u == goal:
            break
        for a, v in next_states(u):
            if v not in seen:
                seen.add(v)
                parent[v] = u
                action[v] = a
                q.append(v)
    if goal not in parent:
        return None
    # reconstruct path
    path, s = [], goal
    while s is not None:
        path.append(s)
        s = parent[s]
    path.reverse()
    steps = []
    for i in range(1, len(path)):
        steps.append((action[path[i]], path[i]))
    return steps

steps = bfs()
print("BFS found steps:", len(steps))
for move, state in steps:
    print(f"{move:>18} -> {state}")
```

B. *Execution Screenshot (BFS)*

IV. Depth-First Search (DFS) Solution

Depth-First Search explores one path fully before backtracking. It is not guaranteed to find the optimal solution, but it can still find a valid one.

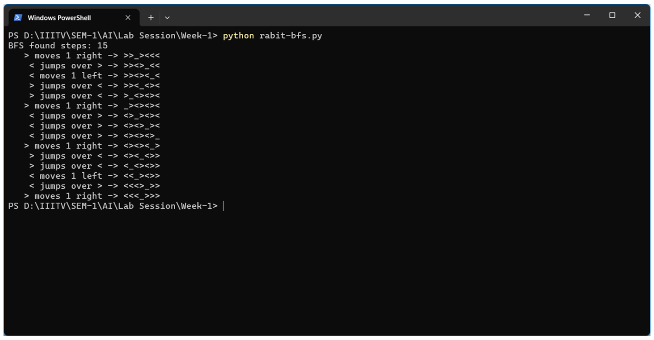


Fig. 1. BFS execution result.

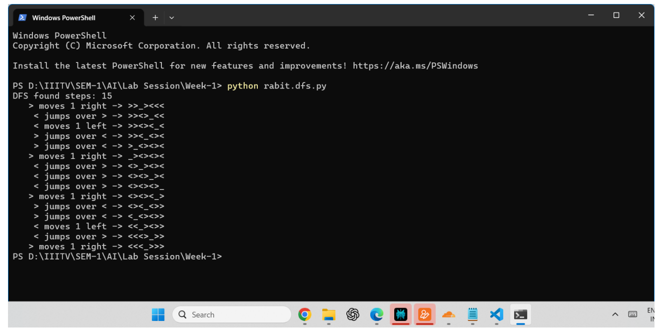


Fig. 2. DFS execution result.

A. Python Code (DFS)

```python
def next_states(state):
    s = list(state)
    i = s.index('_')
    # adjacent moves
    if i - 1 >= 0 and s[i - 1] == '>':
        t = s.copy(); t[i], t[i - 1] = t[i - 1], t[i]
        yield ('> moves 1 right', ''.join(t))
    if i + 1 < len(s) and s[i + 1] == '<':
        t = s.copy(); t[i], t[i + 1] = t[i + 1], t[i]
        yield ('< moves 1 left', ''.join(t))
    # jumps over exactly one opponent
    if i - 2 >= 0 and s[i - 2] == '>' and s[i - 1] == '<':
        t = s.copy(); t[i], t[i - 2] = t[i - 2], t[i]
        yield ('> jumps over <', ''.join(t))
    if i + 2 < len(s) and s[i + 2] == '<' and s[i + 1] == '>':
        t = s.copy(); t[i], t[i + 2] = t[i + 2], t[i]
        yield ('< jumps over >', ''.join(t))

def dfs(start=">>>_<<<", goal="<<<_>>>"):
    stack = [(start, [])]
    visited = {start}
    while stack:
        state, path = stack.pop()
        if state == goal:
            return path
        for move, nxt in reversed(list(next_states(state))):
            if nxt not in visited:
                visited.add(nxt)
                stack.append((nxt, path + [(move, nxt)]))
    return None

steps = dfs()
print("DFS found steps:", len(steps))
for move, state in steps:
    print(f"{move:>18} -> {state}")
```

B. Execution Screenshot (DFS)

V. Final Comparison

- **BFS:** Always finds the optimal (minimum number of moves) solution but required more memory.
- **DFS:** Required less memory but may not find the shortest path.

Aspect	BFS	DFS
Solution Type	Always optimal	Not guaranteed optimal
Time Complexity	$O(b^d)$	$O(b^m)$
Space Complexity	$O(b^d)$ (high)	$O(b \cdot m)$ (low)
Memory Usage	Large	Small

VI. Code Availability

The complete source code is available at: GitHub Repository (CS659 – AI Laboratory).

GitHub Repository:
https://github.com/ChetanKamani/CS659-AI-Lab

Week-2:Plagiarism Detection using A* Algorithm

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

I. Problem Statement

Given two text documents, the task is to align their sentences and detect plagiarism using the A* search algorithm. The alignment should minimize the edit distance (or maximize similarity) between corresponding sentences.

II. Python Code

```python
import sys, re
from heapq import heappush, heappop

# --- Removing Punctutation Marks ---

def normalize_the_text(s: str) -> str:
    s = s.lower()
    s = re.sub(r"[^\w\s\.!\?]", " ", s)
    return re.sub(r"\s+", " ", s).strip()

def tokenize(s: str):
    s = re.sub(r"\s*([\.!\?])\s*", r"\1 ", s)
    parts = re.split(r"[\.!\?]\s+", s)
    return [t.strip() for t in parts if t.strip()]

def words(s: str):
    return [t for t in re.split(r"\W+", s) if t]

def compute_edit_distance(a: str, b: str) -> int:
    A = words(a)
    B = words(b)
    m = len(A)
    n = len(B)
    if m == 0:
        return n
    if n == 0:
        return m
    dp = list(range(n + 1))
    #lavenshtein distance calculation
    for i in range(1, m + 1):
        prev = dp[0]
        dp[0] =  i
        for j in range(1, n + 1):
            tmp = dp[j]
            if A[i-1] == B[j-1]:
                cost = 0
            else:
                cost = 1
            # The following two lines were
                incorrectly indented.
            # They must be inside the inner loop
                to work correctly.
            dp[j] = min(dp[j] + 1, dp[j-1] + 1,
                prev + cost)
            prev = tmp
    return dp[-1]



def ned(text_a: str, text_b: str) -> float:
    num_words_a = len(words(text_a))
    num_words_b = len(words(text_b))
    max_length = max(num_words_a, num_words_b)
    if max_length == 0:
        return 0.0
    raw_distance = compute_edit_distance(text_a,
        text_b)
    return raw_distance / max_length


# --- A* function
def a_star_function(SA, SB, skip_penalty=3.0):
    m, n = len(SA), len(SB)
    def h(i, j):
        return abs((m - i) - (n - j)) *
            skip_penalty

    openq = []
    heappush(openq, (h(0,0), 0.0, 0, 0))
    best = {(0,0): 0.0}
    prev = {}

    while openq:
        f, g, i, j = heappop(openq)
        if (i, j) == (m, n):
            steps = []
            cur = (i, j)
            while cur != (0,0):
                p, op = prev[cur]
                steps.append(op)
                cur = p
            return g, list(reversed(steps))

        # SKIP_A
        if i < m:
            ng, s = g + skip_penalty, (i+1, j)
            if ng < best.get(s, 1e18):
                best[s] = ng; prev[s] = ((i,j), ("
                    SKIP_A", i, -1, skip_penalty))
                heappush(openq, (ng + h(*s), ng, *
                    s))
        # SKIP_B
        if j < n:
            ng, s = g + skip_penalty, (i, j+1)
            if ng < best.get(s, 1e18):
                best[s] = ng; prev[s] = ((i,j), ("
                    SKIP_B", -1, j, skip_penalty))
                heappush(openq, (ng + h(*s), ng, *
                    s))
        # ALIGN
        if i < m and j < n:
            c = compute_edit_distance(SA[i], SB[j
                ])
            ng, s = g + c, (i+1, j+1)
            if ng < best.get(s, 1e18):
                best[s] = ng; prev[s] = ((i,j), ("
                    ALIGN", i, j, float(c)))
                heappush(openq, (ng + h(*s), ng, *
                    s))

    return float("inf"), []

def compare_document(fileA, fileB, t_value,
    sp_value):

    SA = tokenize(normalize_the_text(fileA))
    SB = tokenize(normalize_the_text(fileB))
```

```
    aligned=[]
    total, steps = a_star_function(SA, SB,
        sp_value)

    for s in steps:
        if s[0]=="ALIGN":
            aligned.append(s)

    plag = []

    print("------------Result---------------")
    print("number of sentence in the file A:", len
        (SA))
    print("number of sentence in the file B:", len
        (SB))
    print("total_path_cost:", total)

    print("\nALIGNMENT")
    for op, i, j, c in steps:
        if op == "ALIGN":
            ne = round(ned(SA[i], SB[j]), 3)
            flag = (ne <= t_value)
            if flag: plag.append(1)
            status = "PLAGIARIZED" if flag else "
                ORIGINAL"
            print(f"[ALIGN] A[{i}]<->B[{j}] cost={
                int(c)} NED={ne} Status: {status}"
                )
            print("    A:", SA[i])
            print("    B:", SB[j])
        elif op == "SKIP_A":
            print(f"[SKIP_A] A[{i}] {SA[i]}")
        else:
            print(f"[SKIP_B] B[{j}] {SB[j]}")

    pr = (sum(plag)/len(aligned)) if aligned else
        0.0
    plagiarism_percentage = round(pr * 100, 2)

    # The documents are "identical" at the
        percentage of aligned content that is not
        plagiarized.
    originality_percentage = round((1.0 - pr) *
        100, 2)

    print("\n------------------SUMMARY
        ------------------")
    print(f"Total number of plagiarized pairs
        found: {sum(plag)} out of {len(aligned)}
        aligned sentences.")
    print(f"The documents are {
        plagiarism_percentage}% plagiarized.")
    print(f"The documents are {
        originality_percentage}% original.")
    print(f"Based on the analysis, the two
        documents are {originality_percentage}%
        identical.")



#Input two files while executing the program at
    the command line.

if len(sys.argv) < 3:
    print("Expectating Files. You have not given
        files.")
    sys.exit(1)

#path of file 1
A_path = sys.argv[1]

#path of file 2
B_path = sys.argv[2]

#Getting the threshold value from the user.
t_value = float(input("Enter threshold value: "))

#Getting the skip-panelty value from the user
sp_value  = float(input("Enter skip-panelty value:
     ")) # Corrected typo in the prompt

#initially checking the length of the file. If the
     file length

#reading the content from the file A
with open(A_path, "r") as f:
    A = f.read()

#reading the content from the file B
with open(B_path, "r") as f:
    B = f.read()

compare_document(A, B, t_value, sp_value)
```

III. Input Files

A. Test Case 1: Identical Documents

T1docA.txt

```
I like Artificial Intelligence. I like IIIT
    Vadodara
I like probability & Statistics. I like IIIT
    Vadodara.
```

T1docB.txt

```
I like Artificial Intelligence.
I like IIIT Vadodara.
I like probability and statistics.
```

B. Test Case 2: Slightly Modified Documents

T2docA.txt

```
I like Artificial Intelligence.
I like IIIT Vadodara.
I like probability and statistics.
```

T2docB.txt

```
I like Artificial Intelligence.
I like IIIT Vadodara.
I like probability and statistics.
```

C. Test Case 3: Completely Different Documents

T3docA.txt

```
My name is Chetan Kamani.
I live in Jamnagar.
I work as a lecturer in Government Polytechnic.
```

T3docB.txt

```
Sorting algorithms arrange data in ascending or
    descending order.
QuickSort uses partitioning and recursion.
Heaps are used for priority queues.
```

D. Test Case 4: Partial Overlap

T4docA.txt

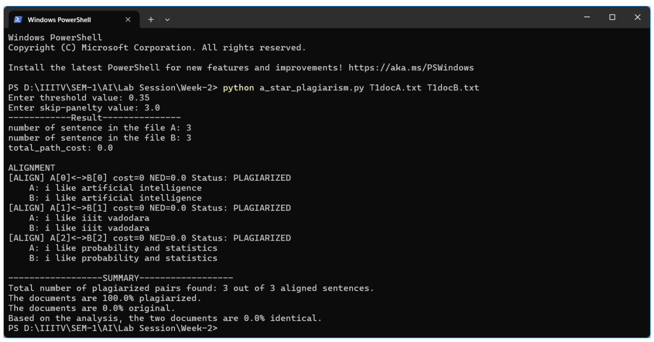
```
My name is Chetan Kamani.
I like IIIT Vadodara.
I enjoy probability and statistics.
Artificial Intelligence is my favorite subject.
I live in Jamnagar.
```

T4docB.txt

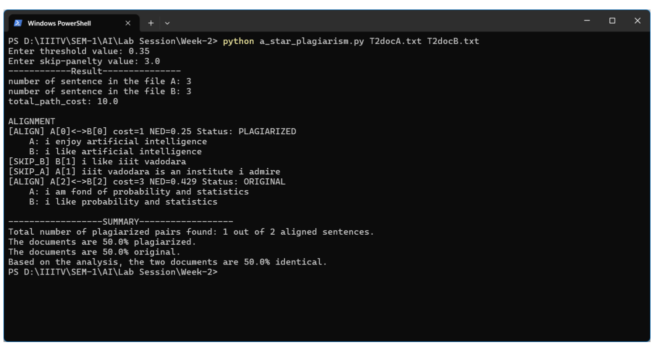
```
I like Artificial Intelligence.
I like IIIT Vadodara.
I study probability and statistics.
I live in Jamnagar.
```

IV. Screenshots of Results

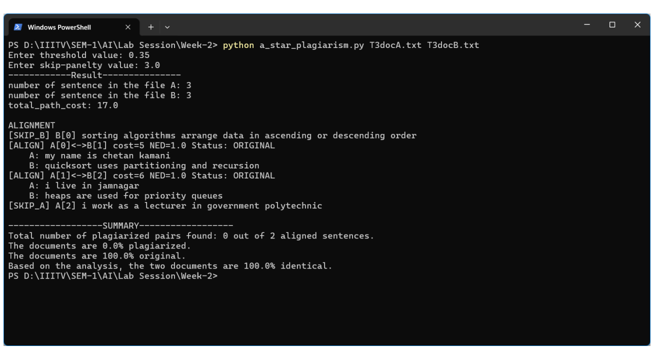
A. Test Case 1



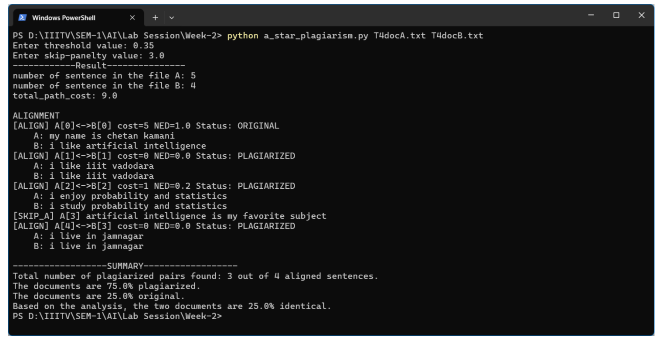
B. Test Case 2



C. Test Case 3



D. Test Case 4



V. Code Availability

The complete source code is available at: GitHub Repository (CS659 – AI Laboratory).

GitHub Repository:

https://github.com/ChetanKamani/CS659-AI-Lab

Week-3:Generating and Solving Uniform Random 3-SAT

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

I. Problem Statement

Write programs to generate uniform random k-SAT instances and, for k=3, solve a set of random 3-SAT problems for different (m, n) combinations. Compare **Hill-Climbing**, **Beam-Search** (beam widths 3 and 4), and **Variable-Neighborhood Descent** (three neighborhoods). Use two heuristic functions and compare them with respect to *penetrance* (fraction of runs solved).

A. Uniform Random k-SAT (Fixed Clause Length Model)

Given k, m, n, generate m clauses. Each clause picks k *distinct* variables from $\{x_1, \ldots, x_n\}$ uniformly at random; each literal is negated with probability $1/2$. For k=3 we obtain uniform random 3-SAT.

II. Python Code

A. A. Simple k-SAT Generator

```
import random

def generate_kSAT(k, m, n):
    clauses = []
    for _ in range(m):
        # pick k distinct variables
        vars_chosen = random.sample(range(1, n+1),
             k)
        clause = []
        for v in vars_chosen:
            if random.choice([True, False]):
                clause.append(f"x{v}")
            else:
                clause.append(f" x {v}")
        clauses.append(clause)
    return clauses

# Example usage
k = 3   # length of each clause
m = 5   # number of clauses
n = 4   # number of variables

formula = generate_kSAT(k, m, n)

print("Random k-SAT formula:")
for i, clause in enumerate(formula, 1):
    print(f"C{i}: ({'     '.join(clause)})")

%% Example function signature:
%%   generate_kSAT(k, m, n) -> list[list[str]]
```

B. B. 3-SAT (HC, Beam=3/4, VND) with Two Heuristics

```
import random

#3-SAT
def generate_3sat(m, n):
    """Generate random 3-SAT instance with m
        clauses, n variables"""
    clauses = []
    for _ in range(m):
        vars3 = random.sample(range(1, n+1), 3)
        clause = []
        for v in vars3:
            if random.choice([True, False]):
                clause.append(v)
            else:
                clause.append(-v)
        clauses.append(clause)
    return clauses

def evaluate(clauses, assignment):
    """Number of satisfied clauses"""
    count = 0
    for c in clauses:
        ok = False
        for lit in c:
            v = abs(lit)
            val = assignment[v]
            if (lit > 0 and val) or (lit < 0 and
                not val):
                ok = True
        if ok: count += 1
    return count

# two heuristics
def h1(clauses, A): return evaluate(clauses, A)
def h2(clauses, A): return 2*evaluate(clauses, A)
    - len(clauses)

# hill climbing
def hill_climbing(clauses, n, heuristic, steps
    =1000):
    A = [None] + [random.choice([False, True]) for
         _ in range(n)]
    for _ in range(steps):
        if evaluate(clauses, A) == len(clauses):
            return True

        best_score, best_var = heuristic(clauses,
            A), None
        for v in range(1, n+1):
            A[v] = not A[v]
            score = heuristic(clauses, A)
            A[v] = not A[v]
            if score > best_score:
                best_score, best_var = score, v
        if best_var is None:
            v = random.randint(1, n)
            A[v] = not A[v]
        else:
            A[best_var] = not A[best_var]
    return False
```

```
def beam_search(clauses, n, heuristic, beam_width
    =3, steps=200):
    beam = []
    for _ in range(beam_width):
        A = [None] + [random.choice([False, True])
             for _ in range(n)]
        beam.append(A)
    for _ in range(steps):
        new_beam = []
        for A in beam:
            if evaluate(clauses, A) == len(clauses
                ): return True
            for v in range(1, n+1):
                B = A.copy()
                B[v] = not B[v]
                new_beam.append(B)
        new_beam.sort(key=lambda x: heuristic(
            clauses, x), reverse=True)
        beam = new_beam[:beam_width]
    return False

def vnd(clauses, n, heuristic, steps=1000):
    A = [None] + [random.choice([False, True]) for
         _ in range(n)]
    for _ in range(steps):
        if evaluate(clauses, A) == len(clauses):
            return True
        improved = False
        for v in range(1, n+1):
            A[v] = not A[v]
            if heuristic(clauses, A) > heuristic(
                clauses, A):
                improved = True
            A[v] = not A[v]
        if not improved:
            v = random.randint(1, n)
            A[v] = not A[v]
    return False


n = 6   # variables
m = 15  # clauses
clauses = generate_3sat(m, n)
print("Generated 3-SAT clauses:", clauses)

for hname, hfun in [("h1", h1), ("h2", h2)]:
    print(f"\nUsing heuristic {hname}:")
    print(" Hill-Climbing:", hill_climbing(clauses
        , n, hfun))
    print(" Beam Search (width=3):", beam_search(
        clauses, n, hfun, beam_width=3))
    print(" Beam Search (width=4):", beam_search(
        clauses, n, hfun, beam_width=4))
    print(" VND:", vnd(clauses, n, hfun))

%%   - instance generator generate_3sat(m, n)
%%   - evaluate(), heuristics h1 and h2
%%   - hill_climbing(), beam_search(beam_width
    =3/4), vnd()
%%   - a small main that prints solved/unsolved
     for both heuristics
```

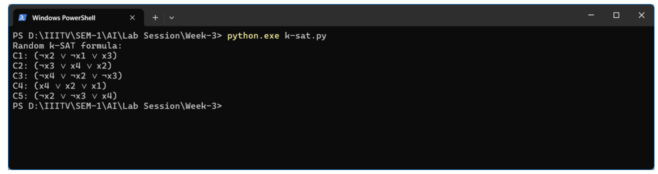
III. How to Run

- Generate k-SAT or run 3-SAT solvers from terminal:

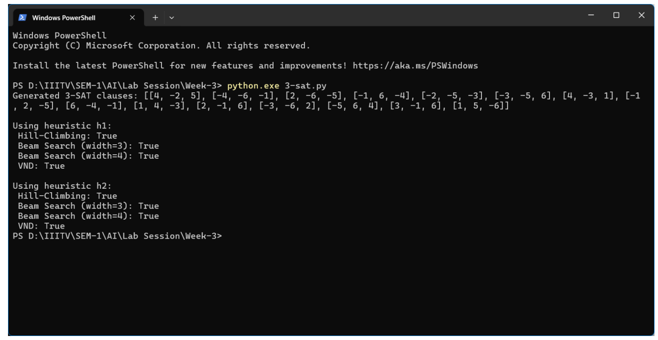
```
python k-sat.py
python 3-sat.py
```

IV. Screenshots of Results

A. k-sat.py



B. 3-sat.py



V. Code Availability

The complete source code is available at: GitHub Repository (CS659 – AI Laboratory).

GitHub Repository:
https://github.com/ChetanKamani/CS659-AI-Lab

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):
```

```python
            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)
```

```
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

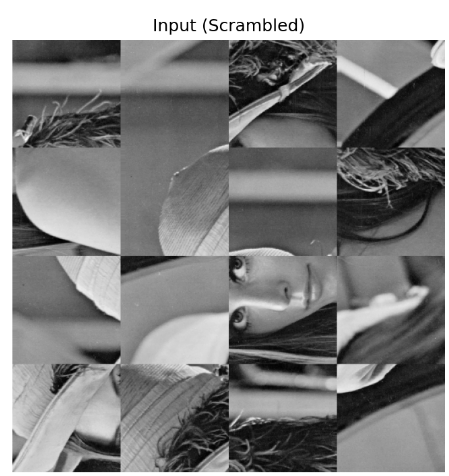


Fig. 1. Scrambled Lena image (input, 4x4 grid).

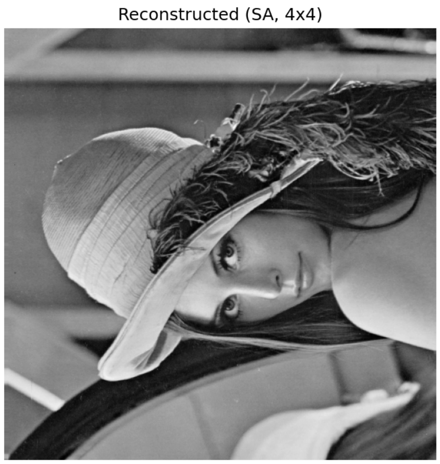


Fig. 2. Reconstructed image using Simulated Annealing.

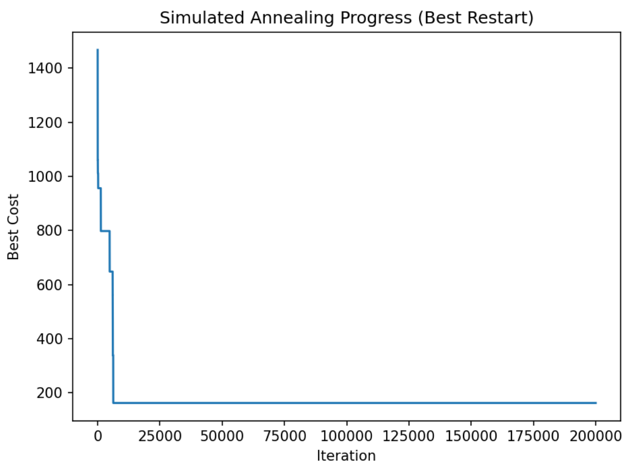


Fig. 3. Convergence curve: best cost vs. iteration.

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

sectionCode Availability The complete source code is available at: GitHub Repository (CS659 – AI Laboratory).

GitHub Repository:
https://github.com/ChetanKamani/CS659-AI-Lab