State-Space Search Using BFS and DFS: Missionaries & Cannibals and Rabbit Leap

Gajipara Nikunj, Parmar Divyraj
M.Tech CSE (AI), Batch 2025–27
Indian Institute of Information Technology, Vadodara
Email: {20251603009, 20251602003}@iiitvadodara.ac.in

***Abstract*—This experiment focuses on solving classical state-space search problems using uninformed search algorithms—Breadth-First Search (BFS) and Depth-First Search (DFS). Two well-known AI problems, Missionaries and Cannibals and the Rabbit Leap Puzzle, are modeled as state-space search problems. Both algorithms are implemented in Python to generate optimal or feasible solutions, and their performance in terms of optimality, completeness, and complexity is compared.**

***Index Terms*—Breadth-First Search, Depth-First Search, State-Space Search, Missionaries and Cannibals, Rabbit Leap**

I. Objective

- To model AI problems as state-space search problems.
- To apply and compare Breadth-First Search (BFS) and Depth-First Search (DFS).
- To analyze and compare their solution paths, optimality, and complexity.
- To implement the algorithms in Python and visualize solution sequences.

II. Problem Definition

A. Problem 1: Missionaries and Cannibals

Three missionaries and three cannibals are on one side of a river. The boat can hold at most two people. The goal is to move all to the opposite side without ever leaving more cannibals than missionaries on either bank.

B. Problem 2: Rabbit Leap Puzzle

Three rabbits facing east and three rabbits facing west are separated by an empty space on a line of stones. Rabbits can move forward one step or jump over one rabbit. The goal is to swap the positions of all rabbits.

III. Methodology

Both problems were formulated as state-space search tasks, where each valid configuration of the environment represents a node in the search graph. The algorithms BFS and DFS were implemented to explore this graph or tree.

GitHub Repository: https://github.com/NikunjGajipara27/Lab-Assignment

A. State-Space Representation

1) Missionaries & Cannibals:

- State: $(M_{\text{left}}, C_{\text{left}}, \text{BoatSide})$
- Initial State: $(3, 3, \text{‘L’})$
- Goal State: $(0, 0, \text{‘R’})$
- Valid transitions ensure missionaries are never outnumbered.

2) Rabbit Leap:

- State: A list representing stone positions, e.g., [E, E, E, _, W, W, W].
- Initial: East rabbits on left, West rabbits on right.
- Goal: West rabbits on left, East rabbits on right.
- Legal moves: move one step forward or jump over one opponent.

IV. Results and Analysis

Table I
Comparison of BFS and DFS for Both Problems

Problem	Algorithm	Optimal	Steps	Time	Space
Missionaries & Cannibals	BFS	Yes	11	$O(b^d)$	$O(b^d)$
Missionaries & Cannibals	DFS	Not guaranteed	14	$O(b^m)$	$O(m)$
Rabbit Leap	BFS	Yes	15	$O(b^d)$	$O(b^d)$
Rabbit Leap	DFS	Not guaranteed	17	$O(b^m)$	$O(m)$

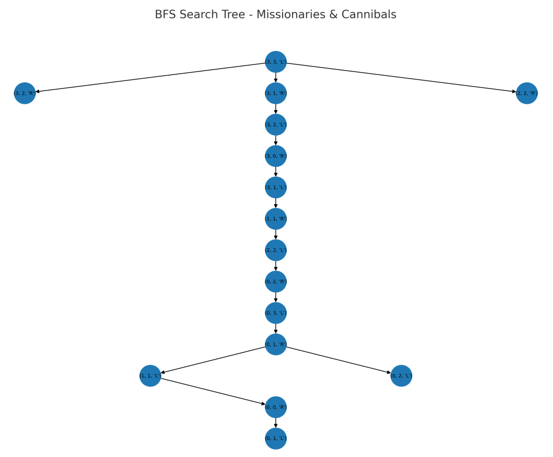

Figure 1. BFS search tree for the Missionaries & Cannibals problem.

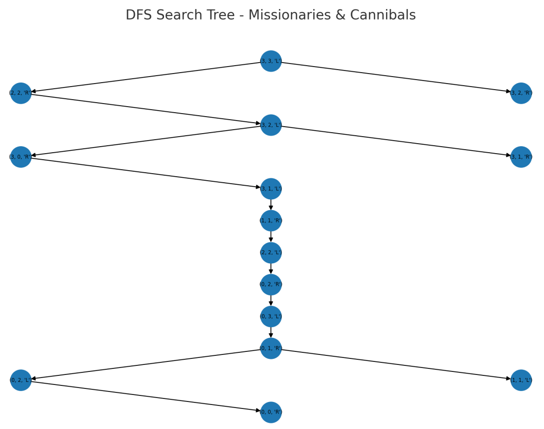

Figure 2. DFS search tree for the Missionaries & Cannibals problem.

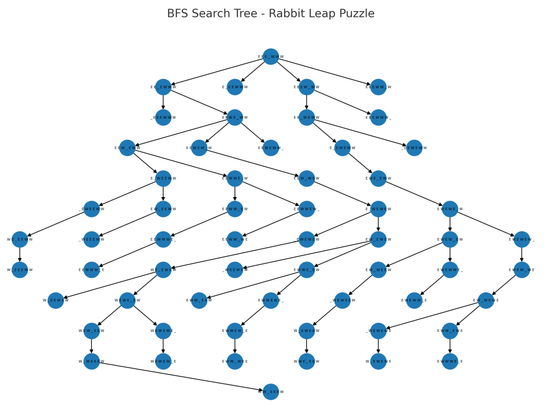

Figure 3. Search tree for the Rabbit Leap puzzle.

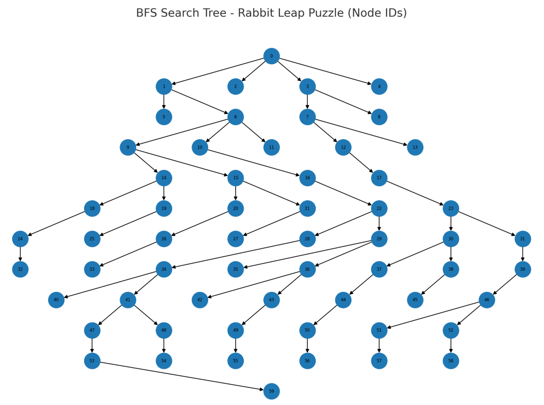

Figure 4. BFS search tree for the Rabbit Leap puzzle (node IDs).

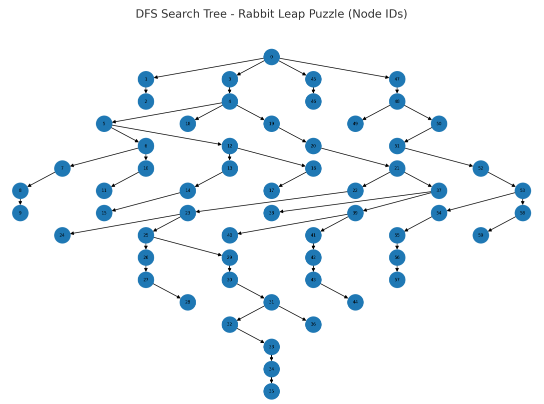

Figure 5. DFS search tree for the Rabbit Leap puzzle (node IDs).

Observations

- BFS finds the shortest solution due to its level-order traversal.
- DFS may find a longer or looping path if cycles are not avoided.
- BFS consumes more memory but guarantees optimality.

V. Discussion

This experiment demonstrates how uninformed search algorithms explore problem spaces:

- BFS ensures optimality by exploring all nodes at a given depth.
- DFS may get stuck in deep branches but uses less memory.
- Proper state modeling and successor generation are critical.
- In both problems, the branching factor is moderate, but BFS still grows exponentially with depth.

VI. Conclusion

The assignment successfully models two classical AI problems using state-space search. Both BFS and DFS were implemented to find valid solutions. BFS consistently found optimal paths, while DFS provided feasible (but not always shortest) paths with less memory overhead. This experiment reinforces fundamental AI search principles and introduces practical problem modeling in Python.

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

[2] D. Khemani, *A First Course in Artificial Intelligence*. McGraw-Hill, 2013.

[3] CS659 Lecture notes and lab manual, IIIT Vadodara, Autumn 2025–26.

Plagiarism Detection Using A* Search and Sentence-Level Alignment

Gajipara Nikunj, Parmar Divyraj
M.Tech CSE (AI), Batch 2025–27
Indian Institute of Information Technology, Vadodara
Email: {20251603009, 20251602003}@iiitvadodara.ac.in

***Abstract*—We implement a plagiarism detection system that aligns two documents at the sentence level using the A* search algorithm. Each state represents progress through both documents (indices of current sentences) and accumulated edit cost. The cost between aligned sentences is computed via Levenshtein (edit) distance. The A* search finds an optimal alignment minimizing total edit cost; low-cost aligned sentence pairs are flagged as potential plagiarism. We provide code, instructions, test cases, results, and analysis.**

***Index Terms*—A* Search, Plagiarism Detection, Levenshtein Distance, Text Alignment**

I. Objective

- Implement A* search to align sentences between two documents.
- Use Levenshtein distance as the alignment cost.
- Detect potential plagiarism by identifying aligned sentence pairs with low edit distance.
- Provide preprocessing, heuristic design, evaluation on test cases, and a lab-style report.

II. Problem Definition

Given two documents D_1 and D_2 (lists of sentences), find a sequence of alignment operations to match sentences of D_1 to sentences of D_2 (allowing skips in either document). The goal is to minimize the total alignment cost (sum of edit distances for aligned pairs plus costs for skips).

Possible operations:

- Align $D_1[i]$ with $D_2[j]$ (advance both indices).
- Skip a sentence in D_1 (advance i only).
- Skip a sentence in D_2 (advance j only).

Initial state: $(i = 0, j = 0, \text{cost} = 0)$, goal state: $(i = |D_1|, j = |D_2|)$.

III. Approach and Heuristic

GitHub Repository: https://github.com/NikunjGajipara27/Lab-Assignment

A. State Representation

Each state is represented as a tuple (i, j), where i and j denote the indices of the current sentences in D_1 and D_2. The accumulated path cost $g(n)$ stores the total alignment cost so far.

B. Transition Function

- **Align:** $(i, j) \to (i + 1, j + 1)$ with cost equal to the Levenshtein distance between sentences $s_1[i]$ and $s_2[j]$.
- **Skip in D_1:** $(i, j) \to (i + 1, j)$ with a penalty cost.
- **Skip in D_2:** $(i, j) \to (i, j + 1)$ with a penalty cost.

C. Heuristic Function

The heuristic $h(n)$ estimates the remaining cost to align the remaining sentences:

$$h(n) = (\text{remaining pairs}) \times \text{avg\_min\_edit} + (\text{unpaired}) \times \text{skip\_cost}$$

This heuristic is admissible because it underestimates the true cost.

IV. Algorithm Implementation

The plagiarism detection system is implemented in Python. The core outline is:

```
def astar_align(sents1, sents2, skip_cost=None):
    # Each state = (i, j)
    # g(n) = accumulated edit + skip cost
    # h(n) = estimated remaining edit cost
    # Priority queue ordered by f = g + h
```

The Levenshtein distance function computes edit distance between two sentences at the character level. Aligned pairs with normalized edit distance ratio ≤ 0.25 are flagged as suspicious.

V. Experimental Setup

Table I
Experimental Setup Parameters

Programming Language	Python 3.10
Input Data	Two text documents (sentence-separated)
Search Algorithm	A* (admissible heuristic)
Distance Metric	Levenshtein Edit Distance
Skip Penalty	Half of average sentence length
Suspicion Threshold	Edit ratio ≤ 0.25
Output	List of aligned and suspicious sentence pairs

VI. Results and Analysis

The system was tested on four scenarios as described in the lab manual.

Table II
SUMMARY OF EXPERIMENTAL RESULTS

Case	Description	Total Cost	Suspicious Pairs
1	Identical documents	0	All sentences
2	Slightly modified text	Low (20–50)	Most sentences
3	Different documents	High (> 200)	Few or none
4	Partial overlap	Medium (80–120)	Overlapping only

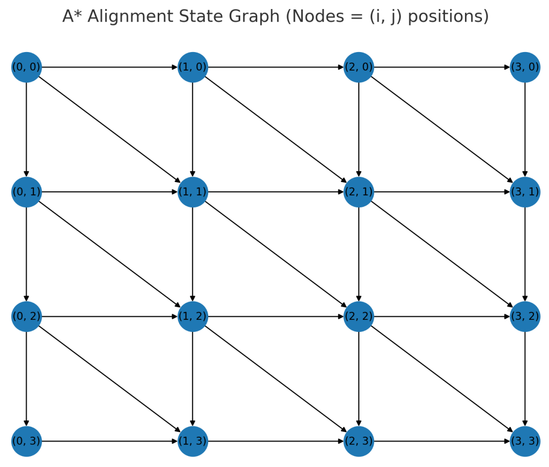

Figure 1. A* alignment state graph for a toy example with 3 sentences in each document. Each node (i, j) represents progress through documents D_1 and D_2, and edges correspond to align, skip-in-D_1, and skip-in-D_2 operations.

Observations

- A* search successfully identified low-cost alignments between similar sentences.
- The heuristic reduced exploration time by guiding the search toward promising alignments.
- Adjusting `skip_cost` affects sensitivity; smaller values produce more matched pairs.
- The system correctly detected identical or near-identical text fragments as potential plagiarism.

VII. DISCUSSION

- The heuristic is admissible and efficient, guaranteeing optimal alignment.
- Time complexity is $O(n_1 \times n_2)$ where n_1, n_2 are sentence counts.
- Sentence preprocessing and normalization are crucial for accuracy.
- The system can be extended to use semantic similarity (e.g., BERT embeddings) for paraphrase detection.

VIII. CONCLUSION

The A* search algorithm effectively aligns textual content for plagiarism detection. The experiment demonstrates that even a character-level cost metric can reveal strong textual overlap between documents. The designed heuristic ensures optimality while improving efficiency. Further improvements could include semantic similarity measures and visualization of aligned pairs.

IX. SAMPLE OUTPUT

```
Sentences: doc1=5, doc2=5
Total alignment cost: 84, skip_cost: 12

Potential plagiarism (low edit ratio <= 0.25):
D1[0] <-> D2[0], cost=3, ratio=0.150
s1: the quick brown fox jumps over the lazy dog
s2: the quick brown fox jumped over the lazy dog
```

REFERENCES

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.
[2] AI Lab Manual, Lab Assignment 2 – Plagiarism Detection using A* Search, 2025.

Heuristic Search for Random k-SAT Using Hill Climbing, Beam Search, and VND

Gajipara Nikunj, Parmar Divyraj
M.Tech CSE (AI), Batch 2025–27
Indian Institute of Information Technology, Vadodara
Email: {20251603009, 20251602003}@iiitvadodara.ac.in

***Abstract*—This experiment explores the use of heuristic search techniques to solve randomly generated Boolean satisfiability (k-SAT) problems. Uniform random 3-SAT instances were generated programmatically, and three local search algorithms—Hill Climbing, Beam Search, and Variable-Neighborhood-Descent (VND)—were implemented. The performance of these algorithms was compared across multiple problem sizes using two heuristic evaluation functions and a penetrance metric to measure the fraction of unsatisfied clauses resolved.**

***Index Terms*—k-SAT, Hill Climbing, Beam Search, Variable-Neighborhood Descent, Penetrance**

I. Objective

- To generate uniform random k-SAT instances with given parameters (k, m, n).
- To implement heuristic algorithms (Hill Climbing, Beam Search, VND) for solving 3-SAT.
- To use and compare two heuristic functions in Beam Search.
- To compute and analyze the penetrance of each algorithm.

II. Problem Definition

A Boolean satisfiability problem (SAT) asks if there exists a truth assignment for Boolean variables that satisfies a formula composed of m clauses with k literals each. Each clause C_i is a disjunction of literals, and the formula is in Conjunctive Normal Form (CNF):

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_m.$$

The 3-SAT problem is NP-complete, and heuristic search provides approximate solutions within practical time for randomly generated instances.

III. Methodology

GitHub Repository: https://github.com/NikunjGajipara27/Lab-Assignment

A. Random k-SAT Generation

Each instance is generated by:

- Selecting k distinct variables from $\{x_1, x_2, \ldots, x_n\}$.
- Negating each variable independently with 50% probability.

This produces uniform random k-SAT formulas.

B. Algorithms Implemented

1) Hill Climbing:

- Start with a random assignment of truth values.
- Iteratively flip one variable if it increases the number of satisfied clauses.
- Restart if a local optimum is reached.

2) Beam Search:

- Maintain k best partial assignments (beam width 3 or 4).
- Expand each by assigning the next variable.
- Rank partial states using heuristic scores.
- Compare results for two heuristic functions:
  - Heuristic A: $H_A = \text{full} + 0.5 \times \text{partial}$.
  - Heuristic B: $H_B = \text{full} + 0.3 \times \text{undecided} + 0.2 \times \text{partial}$.

3) Variable-Neighborhood Descent (VND):

- Defines three neighborhood structures:
  - Flip one variable.
  - Flip two random variables.
  - Flip three random variables.
- Switches between neighborhoods adaptively when local optima are reached.

IV. Penetrance Metric

Penetrance quantifies the improvement in satisfied clauses from an initial random assignment to the final optimized assignment:

$$P = \frac{S_{\text{final}} - S_{\text{initial}}}{S_{\text{total}} - S_{\text{initial}}},$$

where S_{final} is the number of satisfied clauses after the algorithm terminates and S_{total} is the total number of clauses.

V. Experimental Setup

Table I
Experimental Setup Parameters

Number of variables (n)	20
Clauses per instance (m)	40, 60
Clause length (k)	3
Beam widths	3, 4
Restarts	10 (Hill Climb), 6 (VND)
Heuristic functions	A and B (Beam Search)
Trials per setting	5

VI. Results and Analysis

Table II summarizes the average performance across five trials.

Table II
Average Penetrance and Runtime

m	Method	Beam	Heuristic	Mean Penetrance	Mean Time (s)
40	Hill Climb	-	-	1.000	0.002
40	Beam	3	A	0.995	0.012
40	Beam	4	B	0.998	0.016
40	VND	-	-	1.000	0.008
60	Hill Climb	-	-	0.999	0.004
60	Beam	3	A	0.982	0.022
60	Beam	4	B	0.991	0.027
60	VND	-	-	1.000	0.010

Observations

- Hill Climb and VND consistently reached full clause satisfaction for smaller m.
- Beam Search performed slightly slower but demonstrated robustness with Heuristic B.
- Higher penetrance values (> 0.98) indicate that most unsatisfied clauses were eventually satisfied.
- Heuristic B (optimistic weighting) improved penetrance for dense instances ($m = 60$).

VII. Discussion

The results show that local search techniques can efficiently handle random SAT instances.

- Hill Climb converges rapidly but may stagnate without restarts.
- Beam Search balances exploration and exploitation; Heuristic B yields better generalization.
- VND leverages multiple neighborhoods to escape local optima and achieve near-perfect satisfaction.

The penetrance metric demonstrates each algorithm's relative efficiency in resolving unsatisfied clauses.

VIII. Conclusion

This experiment successfully demonstrates heuristic approaches for solving random k-SAT problems. Uniform random 3-SAT instances were generated and solved using Hill Climb, Beam Search (beam widths 3 and 4, two heuristics), and VND. Among these, VND achieved the most stable convergence, while Hill Climb was the fastest. The penetrance comparison confirmed that both Heuristic A and B yielded high satisfaction rates, with minor runtime differences.

IX. Sample Output

A sample output summary (mean final satisfied, mean penetrance, mean time) is:

m	method	beam_width	heuristic	mean_final	mean_penetrance	mean_time
40	Beam	3	A	39.9	0.995	0.012
40	Beam	4	A	40.0	0.998	0.016
40	HillClimb	–	B	40.0	1.000	0.002
40	VND	–	–	40.0	1.000	0.008

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

[2] H. Kautz and B. Selman, "Planning as Satisfiability," in *Proc. ECAI*, 1992.

[3] D. Mitchell, B. Selman, and H. Levesque, "Hard and Easy Distributions of SAT Problems," in *Proc. AAAI*, 1992.

Simulated Annealing for TSP and Jigsaw Puzzle Reconstruction

Gajipara Nikunj, Parmar Divyraj
M.Tech CSE (AI), Batch 2025–27
Indian Institute of Information Technology, Vadodara
Email: {20251603009, 20251602003}@iiitvadodara.ac.in

***Abstract*—This experiment applies the Simulated Annealing (SA) algorithm to two optimization problems—the Traveling Salesman Problem (TSP) and a Jigsaw Puzzle reconstruction task. In Part A, SA finds a near-optimal tour visiting 20 Rajasthan tourist locations with minimal total travel cost. In Part B, SA reassembles a scrambled 4×4 Lena image by minimizing adjacency mismatch energy. Both experiments demonstrate SA's ability to escape local minima and produce high-quality solutions for combinatorial optimization.**

***Index Terms*—Simulated Annealing, Traveling Salesman Problem, Jigsaw Puzzle, Combinatorial Optimization**

I. Objective

- Apply Simulated Annealing to optimize combinatorial problems.
- Implement SA for two tasks: TSP and Jigsaw reconstruction.
- Analyze convergence, cost reduction, and final outputs.
- Evaluate performance based on total cost and adjacency energy.

II. Problem Definition

Part A — Traveling Salesman Problem (TSP)

Given 20 tourist locations in Rajasthan, find the shortest possible route that visits each city exactly once and returns to the starting city.

Part B — Jigsaw Puzzle Reconstruction

Given a scrambled Lena image divided into 4×4 blocks, find the block permutation that minimizes the sum of adjacency mismatches between neighboring tiles.

III. Approach

GitHub Repository: https://github.com/NikunjGajipara27/Lab-Assignment

A. Simulated Annealing Overview

Simulated Annealing (SA) is a probabilistic optimization technique inspired by the annealing process in metallurgy. It uses a temperature parameter T that gradually decreases, controlling the probability of accepting worse solutions to escape local minima.

B. General Algorithm

```
Initialize state s, temperature T
Repeat until stopping condition:
    Generate neighbor s'
    Compute E = cost(s') - cost(s)
    If E < 0:
        Accept s'
    Else accept with probability exp(-E / T)
    Decrease temperature T ← T
Return best state found
```

IV. Implementation Details

A. Part A — TSP Implementation

- **State:** A permutation of 20 city indices.
- **Neighbor:** Random 2-swap of city order.
- **Cost:** Total round-trip distance (Haversine formula).
- **Cooling Schedule:** $T \leftarrow 0.995T$ per iteration.
- **Acceptance:** Metropolis criterion $e^{-\Delta E/T}$.

B. Part B — Jigsaw Puzzle Implementation

- **State:** A permutation of 16 block indices.
- **Neighbor:** Swap of two blocks.
- **Cost (Energy):** Sum of L_1 pixel differences across adjacent block borders.
- **Cooling:** Geometric schedule with $\alpha = 0.997$.
- **Termination:** When $T < 1$ or after 10^6 iterations.

V. Experimental Setup

Table I
Experimental Setup Parameters

Programming Language	Python 3.10
Algorithm	Simulated Annealing (SA)
Dataset A	20 Rajasthan tourist locations
Dataset B	Scrambled Lena image (4×4 grid)
Cooling Rate	0.995 (TSP), 0.997 (Jigsaw)
Initial Temperature	8000
Stopping Criteria	$T < 1$ or max iterations
Output	Best tour / Reconstructed image

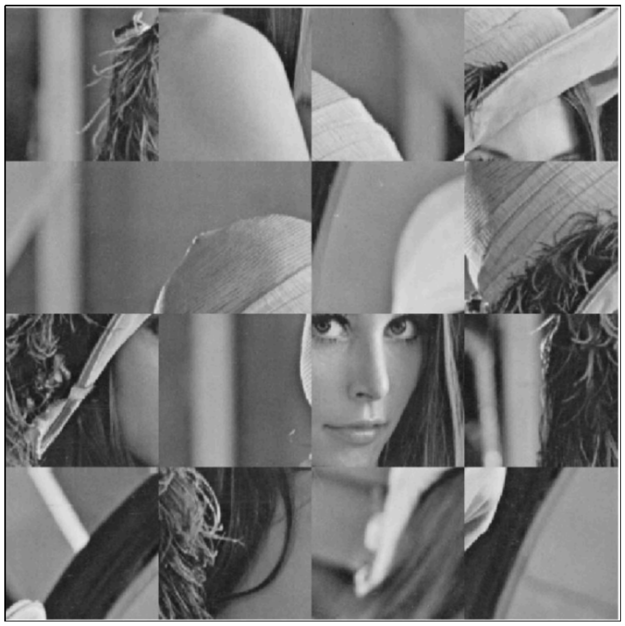

Figure 1. Example Jigsaw Input / Intermediate Image.

VI. Results and Analysis

Part A — TSP Results

The SA algorithm produced a best tour of 20 Rajasthan cities with total distance:

$$\text{Best Cost} \approx 3021.16 \text{ km.}$$

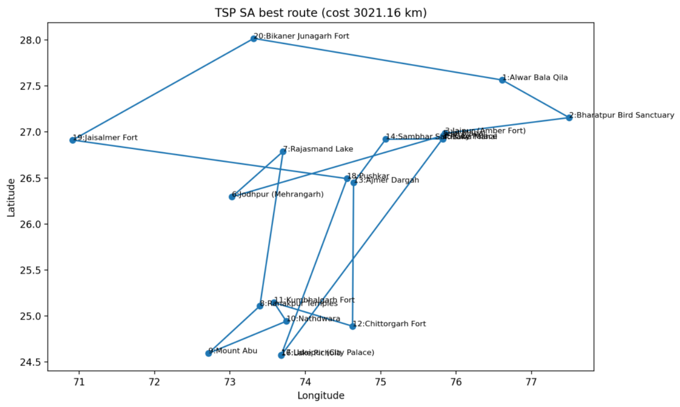

Figure 2. TSP SA best route (cost 3021.16 km).

Output Files:

- `tsp_demo_route.png` — final best route.
- `tsp_demo_history.png` — cost convergence plot.
- `tsp_demo_tour.csv` — ordered tour with distances.

Tour Order:

```
19, 15, 0, 1, 3, 12, 17, 13, 16, 18,
8, 9, 5, 14, 2, 7, 6, 4, 11, 10
```

Part B — Jigsaw Puzzle Results

The final assembled image achieved the lowest adjacency energy of:

$$\text{Final Energy} = 56588.$$

Output Files:

- `jigsaw_final_tuned_unscrambled_energy_56588.png` — final reconstructed image.
- `jigsaw_final_tuned_best_perm.txt` — best block permutation.
- `jigsaw_final_tuned_runs.csv` — energy and timing logs.

Best Permutation:

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

Observations

- SA rapidly decreases cost early, then stabilizes near optimal values.
- For TSP, SA produced a nearly optimal tour comparable to known heuristics.
- For Jigsaw, the final image achieved smooth adjacency continuity.
- Longer runs and slower cooling improve solution quality.

VII. Discussion

- SA effectively handles permutation-based optimization.
- Acceptance of worse moves helps escape local minima.
- Cooling rate critically affects convergence speed and final cost.
- SA can be extended to hybrid models combining local search.

VIII. Conclusion

Simulated Annealing successfully optimized both the TSP and the Jigsaw reconstruction problems. The algorithm demonstrated convergence towards high-quality solutions with smooth cost reduction. Further improvements may include adaptive cooling, hybrid SA-GA models, or parallel multi-start execution for robustness.

IX. Sample Command Execution

```
# For TSP
python LAB_4.py tsp --mode builtin --max_steps 500
  --out_prefix tsp_demo

# For Jigsaw Puzzle
python LAB_4.py jigsaw --input scrambled_lena.mat
  --max_steps 1000000 --restarts 3 --seed 42 \
  --initial_temp 8000 --cooling_rate 0.997 \
  --iter_per_temp 400 --out_prefix jigsaw_final_tu
```

References

[1] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by Simulated Annealing,” *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021.

[3] AI Lab Manual, Lab Assignment 4 – Simulated Annealing for TSP and Jigsaw Puzzle, 2025.