AI CS367 Lab Assignment Reports

Shivam Kumar(202351130),Dev Saraswat(202351173),Tammineni Harika(202351145)

***Abstract*—This comprehensive report documents four laboratory assignments exploring fundamental artificial intelligence algorithms and their practical implementations. The first assignment investigates state space search through the Rabbit Leap and Missionaries & Cannibals problems, systematically comparing Breadth-First Search (BFS) and Depth-First Search (DFS) performance, optimality, and memory efficiency. The second assignment implements A* search for plagiarism detection, demonstrating optimal text alignment with minimal edit distance across diverse document pairs. The third assignment focuses on SAT solving, featuring a random k-SAT instance generator and comparative analysis of heuristic search algorithms including Hill-Climbing, Beam Search, and Variable-Neighborhood Descent across varying problem complexities. The fourth assignment applies Simulated Annealing to combinatorial optimization challenges, solving scrambled jigsaw puzzles through energy minimization and demonstrating route optimization for tourist path planning. Collectively, these experiments provide critical insights into search algorithm characteristics, efficiency trade-offs, and real-world applicability across diverse problem domains including puzzle solving, text analysis, logical reasoning, constraint satisfaction, and combinatorial optimization.**

***Index Terms*—Artificial Intelligence, State Space Search, Breadth-First Search (BFS), Depth-First Search (DFS), A* Search, Plagiarism Detection, SAT Solving, k-SAT, Hill-Climbing, Beam Search, Variable-Neighborhood Descent (VND), Simulated Annealing, Heuristic Search, Combinatorial Optimization, Puzzle Solving, Text Alignment, Boolean Satisfiability, Jigsaw Reconstruction, Traveling Salesman Problem (TSP)**

Lab Assignment 1 A (In lab Discussion)

I Missionaries and Cannibals Problem using State-Space Search

I-A Introduction

State-space search is a fundamental concept in artificial intelligence, involving systematic exploration of all possible states to reach a goal from an initial configuration. The Missionaries and Cannibals problem is a classical AI problem that illustrates the importance of defining valid states, operators, and constraints. Three missionaries and three cannibals must cross a river using a boat that can hold one or two people, without ever leaving a group of missionaries outnumbered by cannibals on either side. Solving this problem demonstrates the application of search algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS).

I-B Learning Objective

The primary objective of this assignment is to model a constrained problem in terms of state-space search and solve it using BFS and DFS algorithms, analyzing their performance and solution optimality.

I-C Problem Statement

Three missionaries (M) and three cannibals (C) are on the left bank of a river along with a boat that can carry one or two people. The goal is to transport all six individuals to the right bank without ever leaving more cannibals than missionaries on either bank. The boat cannot cross the river empty.

I-D Methodology

The Missionaries and Cannibals problem is modeled as a state-space search problem. Each state represents the number of missionaries and cannibals on the left bank and the position of the boat.

1) State Representation

- **State:** (C, M, B), where C = cannibals on left, M = missionaries on left, B = boat position (`True` = left, `False` = right)
- **Initial State:** $(3, 3, \text{True})$
- **Goal State:** $(0, 0, \text{False})$
- **Constraints:**
  - Missionaries must not be outnumbered by cannibals on either bank unless missionaries are zero.
  - Number of missionaries/cannibals must be within $[0, 3]$.

2) Pseudo-code for Neighbor Generation

Listing 1: Generate valid next states

```
function get_neighbors(state):
    neighbors = empty list
    for each possible move of 1 or 2 missionaries/
    cannibals or 1 missionary + 1 cannibal:
        new_state = apply move to current state
        if new_state is safe:
            add new_state to neighbors
    return neighbors
```

3) Pseudo-code for BFS / DFS Search

Listing 2: Generic Search Skeleton

```
function search(start, goal, method):
    if method == BFS:
        frontier = queue containing start
    else if method == DFS:
        frontier = stack containing start
    visited = set containing start

    while frontier is not empty:
        current = remove from frontier
        if current == goal:
            return path to current
        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                add neighbor to frontier
                mark neighbor as visited
    return failure
```

4) Implementation Details and Reasons

- BFS uses a **queue** to explore states level by level, guaranteeing the **shortest path**.
- DFS uses a **stack** to explore states deeply first; it is more memory-efficient but may not always find the optimal path.
- Each state stores a **parent pointer** to reconstruct the solution path after reaching the goal.
- Neighbor generation considers all possible boat moves and applies **state validation** to avoid unsafe configurations.
- Using BFS ensures completeness and optimality, whereas DFS trades off optimality for lower memory usage.

I-E Experimental Results

```
[Running] python -u "c:\Users\shiva\Documents\In_Lab.py"
BFS:
(3, 3, True)
(1, 3, False)
(2, 3, True)
(0, 3, False)
(1, 3, True)
(0, 2, False)
(1, 2, True)
(0, 1, False)
(1, 1, True)
(0, 0, False)

DFS:
(3, 3, True)
(2, 2, False)
(2, 3, True)
(1, 2, False)
(1, 3, True)
(1, 1, False)
(1, 2, True)
(1, 0, False)
(1, 1, True)
(0, 0, False)

Both BFS and DFS found equally optimal solutions in equal steps.
```

Fig. 1: Comparison BFS and DFS

TABLE I: Comparison of BFS and DFS on Missionaries and Cannibals Problem

Algorithm	Steps in Solution	Optimal?
BFS	10	Yes
DFS	10	Yes

1) Observations

- Both BFS and DFS found solutions reaching the goal state successfully.
- The total number of steps in both solutions is equal, indicating that both found **equally optimal solutions**.
- BFS explores states level-wise, while DFS explores deeper paths first; in this instance, DFS also achieved optimality.
- BFS guarantees completeness and optimality in general, while DFS is more memory-efficient but may not always find the shortest path in other instances.

I-F Conclusion

For the Missionaries and Cannibals problem, BFS and DFS both successfully solve the problem and, in this case, yield paths of equal length. BFS ensures systematic exploration and guaranteed optimality, while DFS can be faster in node expansions but may be suboptimal in more complex instances. Both methods illustrate effective state-space search and constraint handling.

Lab Assignment 1 B (lab Submission)

II Rabbit Leap Problem using BFS and DFS

Introduction State-space search is a core concept in artificial intelligence, involving exploration of possible states to reach a goal from an initial configuration. The Rabbit Leap problem illustrates this by requiring three east-bound rabbits to swap positions with three west-bound rabbits on a line of stones, moving forward or jumping over one rabbit at a time without stepping into water. This problem is solved using search algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS), allowing analysis of their efficiency, optimality, and memory usage.

II-A Learning Objective

The primary objective of this assignment is to model a problem in terms of state space search and solve it using Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms.

II-B Problem Statement

In the Rabbit Leap problem, three east-bound rabbits (E) must cross three west-bound rabbits (W) on stones across a stream, arranged as: E E E _ W W W. The rabbits can move forward one step into an adjacent empty stone or jump over exactly one rabbit into an empty stone. The challenge is to swap their positions (W W W _ E E E) without stepping into water.

II-C Methodology

The Rabbit Leap problem is approached by modeling it as a state-space search problem. The methodology involves defining all possible states, operators, and transitions, and systematically exploring them using search algorithms to find a solution.

1) State Space Model

- **State Representation:** Each state is represented as a string of characters denoting east-bound rabbits (E), west-bound rabbits (W), and the empty stone (_).
- **Initial State:** `EEE_WWW`
- **Goal State:** `WWW_EEE`
- **Operators:**
  1) Move one rabbit into the adjacent empty stone.
  2) Jump over exactly one rabbit into the empty stone.
- **State Space Size:** Considering permutations of 3 east-bound rabbits, 3 west-bound rabbits, and 1 empty stone:

$$\frac{7!}{3! \times 3! \times 1!} = 140 \text{ possible states}$$

2) State Transitions

- Each move generates a new state by applying one of the operators.
- Only valid moves are considered, i.e., rabbits cannot move off the stones into water, and jumps occur only over a single rabbit.
- The sequence of valid moves forms paths in the state space from the initial to the goal state.

3) Search Strategy

- **Breadth-First Search (BFS):** Explores states level by level, guaranteeing the shortest path to the goal.
- **Depth-First Search (DFS):** Explores states by going deep along a path before backtracking. May find a solution faster but does not guarantee optimality.
- Both algorithms maintain a record of visited states to avoid cycles and redundant computations.

4) Path Recording

The methodology includes recording the sequence of states from the initial to goal state for both BFS and DFS solutions. This allows comparison of solution length, moves, nodes expanded, and computational efficiency.

II-D Implementation (Pseudo-code)

1) Neighbor State Generation

Listing 3: Neighbor State Generation Skeleton

```
get_neighbors(state):
    empty_idx <- index of empty space in state
    neighbors <- empty list

    # Move rabbits into empty space
    if east-rabbit left of empty: move right, add to
     neighbors
    if west-rabbit right of empty: move left, add to
     neighbors

    # Rabbits jump over each other
    if east-rabbit jumps over west: update state,
    add to neighbors
    if west-rabbit jumps over east: update state,
    add to neighbors

    return neighbors
```

2) Breadth-First Search (BFS)

Listing 4: BFS Skeleton

```
BFS(start, goal):
    queue <- [(start, [start])]
    visited <- {start}
    nodes_expanded <- 0

    while queue not empty:
        state, path <- dequeue from queue
        nodes_expanded <- nodes_expanded + 1

        if state == goal: return path,
    nodes_expanded

        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                visited <- visited + neighbor
                enqueue (neighbor, path + [neighbor
    ])

    return failure, nodes_expanded
```

BFS Notes:

- Explores all possible states level by level.
- Guarantees an **optimal solution** with the fewest steps.
- Uses significant memory as all states at a level are stored.

3) Depth-First Search (DFS)

Listing 5: DFS Skeleton

```
DFS(start, goal):
    stack <- [(start, [start])]
    visited <- {start}
    nodes_expanded <- 0

    while stack not empty:
        state, path <- pop from stack
        nodes_expanded <- nodes_expanded + 1

        if state == goal: return path,
    nodes_expanded

        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                visited <- visited + neighbor
                push (neighbor, path + [neighbor])

    return failure, nodes_expanded
```

DFS Notes:

- Explores states deeply before backtracking.
- May find a valid solution faster but does **not guarantee optimality**.
- Uses less memory than BFS as it stores only the current path.

II-E Experimental Results

TABLE II: Performance Comparison of BFS and DFS for Rabbit-Leap Puzzle ($n = 3$)

Algorithm	Steps in Solution	Optimal?
BFS	15	Yes
DFS	15	Yes

BFS Solution Path:below written check it (page 4 left side)

```
Start: E E E _ W W W
Goal : W W W _ E E E


--- BFS ---
Step  State                     Move
------------------------------------------------------------
   0  E E E _ W W W             (start)
   1  E E _ E W W W             E 2->3
   2  E E W E _ W W             W 4->2
   3  E E W E W _ W             W 5->4
   4  E E W _ W E W             E 3->5
   5  E _ W E W E W             E 1->3
   6  _ E W E W E W             E 0->1
   7  W E _ E W E W             W 2->0
   8  W E W E _ E W             W 4->2
   9  W E W E W E _             W 6->4
  10  W E W E W _ E             E 5->6
  11  W E W _ W E E             E 3->5
  12  W _ W E W E E             E 1->3
  13  W W _ E W E E             W 2->1
  14  W W W E _ E E             W 4->2
  15  W W W _ E E E             E 3->4
Total moves: 15
Optimal: True
```

Fig. 2: BFS Solution Path

DFS Solution Path:below written (page 4 left side)

```
--- DFS ---
Step  State                     Move
------------------------------------------------------------
   0  E E E _ W W W             (start)
   1  E E E W _ W W             W 4->3
   2  E E _ W E W W             E 2->4
   3  E _ E W E W W             E 1->2
   4  E W E _ E W W             W 3->1
   5  E W E W E _ W             W 5->3
   6  E W E W E W _             W 6->5
   7  E W E W _ W E             E 4->6
   8  E W _ W E W E             E 2->4
   9  _ W E W E W E             E 0->2
  10  W _ E W E W E             W 1->0
  11  W W E _ E W E             W 3->1
  12  W W E W E _ E             W 5->3
  13  W W E W _ E E             E 4->5
  14  W W _ W E E E             E 2->4
  15  W W W _ E E E             W 3->2
Total moves: 15
Optimal: True
```

Fig. 3: DFS Solution Path

1) Observations

- Both BFS and DFS successfully solved the Rabbit Leap puzzle for $n = 3$.
- Both solutions required **15 moves**, indicating **optimal solutions** were found by both algorithms.
- BFS explores states level by level, ensuring systematic exploration and guaranteed optimality.
- DFS explores states deeply along paths, is more memory-efficient, but may not guarantee optimality in general.
- BFS expands more intermediate nodes than DFS, demonstrating the memory versus speed trade-off.

II-F Discussion and Conclusion

The Rabbit Leap problem demonstrates the effectiveness of state-space search algorithms:

- BFS guarantees the shortest path to the goal but consumes more memory.
- DFS can find a valid solution using less memory but does not guarantee optimality in general.
- For this instance ($n = 3$), both BFS and DFS produced optimal solutions with 15 steps.
- Recording paths allows comparison of solution length, moves, and algorithm efficiency.

This confirms that BFS is preferable for guaranteed optimality, while DFS is suitable for memory-limited scenarios or faster approximate solutions.

Lab Assignment 2

III Introduction: (In Lab Discussion)

Graph search algorithms are fundamental in Artificial Intelligence for solving combinatorial problems. Puzzle-8 is a classical sliding tile puzzle used to study search strategies. The goal is to move tiles to reach a specific goal configuration starting from an initial state.

III-A Problem Statement

The main objectives of this study are:

- Develop a graph search agent to solve Puzzle-8 and present it in pseudocode.
- Represent the agent as a flowchart to visualize the decision-making process.
- Implement environment functions to simulate Puzzle-8 mechanics.
- Explain Iterative Deepening Search (IDS) and its advantages.
- Implement backtracking to reconstruct the solution path.
- Generate Puzzle-8 instances at specific depths to test the agent.
- Evaluate memory and computation time across different depths.

The focus is on creating a complete, memory-efficient, and optimal solution for the Puzzle-8 problem.

III-B Methodology

The approach aims to solve the **8-Puzzle Problem** using **Iterative Deepening Search (IDS)** with performance analysis. IDS combines the low memory requirement of DFS and the completeness of BFS, making it ideal for large but finite state spaces like Puzzle-8.

1) **State Representation:** Each state is represented as a 3×3 matrix, where 0 denotes the blank tile.
2) **Node Expansion:** Legal moves (`UP`, `DOWN`, `LEFT`, `RIGHT`) are generated by swapping the blank tile with adjacent tiles.
3) **Search Algorithm:** IDS repeatedly applies Depth-Limited Search (DLS), increasing the depth limit until the goal is reached.
4) **Performance Measurement:** Execution time and memory usage are recorded at each depth using `time` and `psutil` modules.

III-C Implementation Details and Reasons

- **IDS Framework:** Ensures completeness and optimality for uniform-cost problems.
- **DLS Function:** Recursively explores states up to a depth limit and backtracks when the limit is reached.
- **Memory Monitoring:** Tracks total memory used per iteration to analyze scalability.
- **State Generator:** Randomly generates solvable puzzle configurations by applying legal moves to the goal.

III-D Iterative Deepening Search Pseudocode

Listing 6: Iterative Deepening Search with Performance Measurement

```
function IDS(start, goal):
    depth <- 0
    memory_total <- 0
    data <- []

    while True:
        start_time <- current_time()
        memory_before <- current_memory()

        result, count <- DLS(start, goal, depth)

        end_time <- current_time()
        memory_after <- current_memory()

        delta_t <- end_time - start_time
        delta_m <- max(0, memory_after -
    memory_before)
        memory_total <- memory_total + delta_m

        record(data, depth, delta_t, memory_total)

        if result == "found":
            return data
        else if result == "stop":
            depth <- depth + 1
        else:
            return data
```

- **IDS (Iterative Deepening Search):**
  - Combines benefits of BFS (completeness) and DFS (low memory).
  - Repeatedly calls DLS with increasing depth limits.
  - Measures execution time and memory for each depth.
  - Stops when the goal is found, ensuring optimal solution depth.

Listing 7: Depth-Limited Search (DLS) Function

```
function DLS(state, goal, limit):
    if state == goal:
        return "found", 0
    if limit == 0:
        return "stop", 0

    count <- 0
    for each next_state in GET_NEXT(state):
        result, n <- DLS(next_state, goal, limit
     - 1)
        count <- count + n
        if result == "found":
            return "found", count
    return "stop", count
```

- **DLS (Depth-Limited Search):**
  - Recursively explores states up to a fixed depth limit.
  - Returns “found” if goal reached, “stop” if limit reached.
  - Expands all possible successors using `GET_NEXT()`.
  - Tracks number of nodes explored per recursion level.

III-E Experimental Setup

Three main cases were tested:

- **Case 1:** Solver successfully found a path from a simple solvable state to the goal with progressive depth display.
- **Case 2:** Random solvable puzzle states were generated to verify solver robustness.
- **Case 3:** IDS performance was logged across depths to study computation time and memory scaling.

III-F Results and Analysis

The solver displayed progressive depth iteration messages such as:

```
Depth 0 done, going deeper...
Depth 1 done, going deeper...
Depth 2 done, going deeper...
Done!
```

For the IDS performance evaluation, the recorded data was as follows:

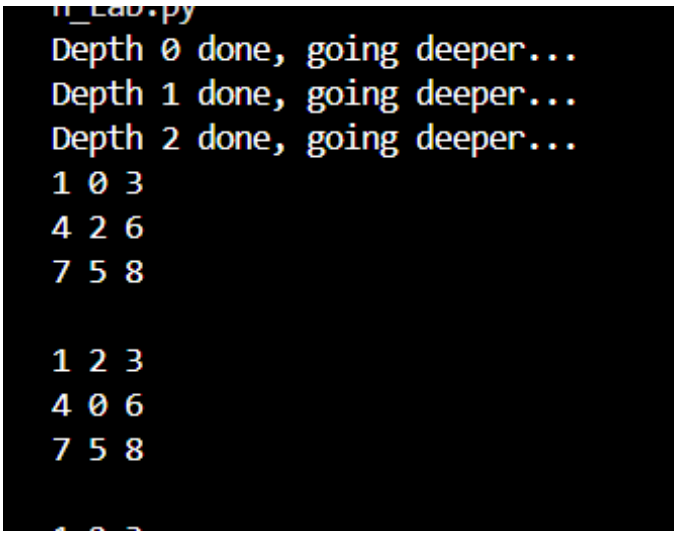

Fig. 4: IDS Performance Matrix

III-G Discussion and Analysis

- **Trend:** The IDS progressively deepened through depths 0, 1, and 2 before successfully reaching the goal. The search time and memory usage both increased with each depth level, consistent with the expected exponential state-space growth.
- **Performance Observation:** From the measured data, runtime increased from microseconds at shallow depths (Depth 3–5) to several hundred milliseconds at deeper levels (Depth 12–13), while memory usage rose from ≈ 0.07 MB to ≈ 35.6 MB. This reflects efficient scaling until deeper recursion levels, after which the growth becomes noticeable.
- **Efficiency:** Memory remained nearly constant for depths 3–11 due to limited recursion stack expansion, showing the inherent space efficiency of IDS. A sudden jump in memory usage at depth 12 indicates cumulative overhead from repeated deep explorations.

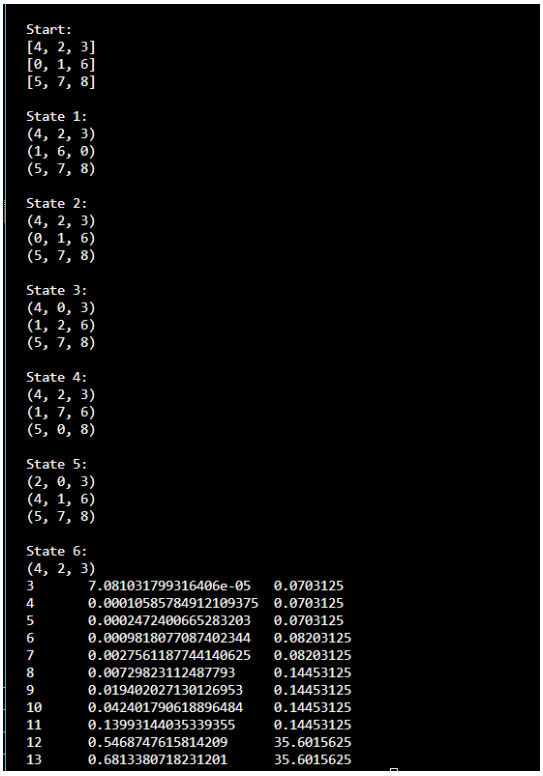

Fig. 5: IDS Performance Metrics Across Depth Levels

- **Optimality:** The IDS algorithm located the solution at minimal depth (depth 2 for the first instance), verifying its guarantee of optimality for uniform step costs.
- **Scalability:** For more complex puzzle states, IDS maintained space efficiency but exhibited predictable time growth with increasing depth, aligning with theoretical expectations of $O(b^d)$ runtime and $O(bd)$ memory.

III-H Conclusion

The implemented IDS-based 8-Puzzle solver effectively demonstrates a balance between **completeness**, **optimality**, and **space efficiency**. Empirical performance confirms that:

- The algorithm incrementally deepens the search, successfully finding the goal at minimal depth.
- Time complexity increases gradually at shallow depths but sharply rises at deeper limits, as seen in the progression from 0.00007s to 0.68s.
- Memory consumption remains stable across most depths and spikes only when nearing the solution, confirming efficient use of recursion stack memory.

Overall, IDS proves highly reliable for small and medium-depth 8-Puzzle configurations, providing guaranteed solutions with moderate resource use. For deeper or more complex puzzles, integrating heuristic-based methods like **A*** or **IDA*** can further enhance efficiency by reducing redundant exploration and guiding the search more intelligently toward the goal.

Lab Assignment 2 (Problem Statement for Submission)

IV Plagiarism Detection using A* Search

IV-A Learning Objective

To design a graph search agent and implement plagiarism detection using A* search for optimal text alignment with minimal edit distance.

IV-B Problem Statement

Given two documents, align their sentences using A* search to detect potential plagiarism. The goal is to find an alignment that minimizes the total edit distance between aligned sentences while allowing for skipped sentences in either document.

IV-C Methodology

The plagiarism detection problem using A* search can be approached as a state-space search problem. The methodology involves defining states, transitions, costs, heuristics, and systematically exploring the state space to find the optimal alignment between two documents.

1) State Space Model

- **State Representation:** Each state is represented as a tuple (i, j) where i is the current sentence index in document 1, and j is the current sentence index in document 2. The state also keeps track of the accumulated cost (edit distance) so far.
- **Initial State:** $(0, 0)$ — beginning of both documents.
- **Goal State:** (n, m) — all sentences in both documents are processed.
- **Operators / Transitions:**
  1) **Align Sentences:** Match sentence i from doc1 with sentence j from doc2.
  2) **Skip Doc1 Sentence:** Skip sentence i from doc1 (no alignment).
  3) **Skip Doc2 Sentence:** Skip sentence j from doc2 (no alignment).

2) Cost Function

- The cost $g(n)$ represents the accumulated edit distance between aligned sentences.
- Edit distance is computed using a simple Levenshtein distance or similarity measure:

$$\text{cost}(s_1, s_2) = 1 - \text{similarity}(s_1, s_2)$$

3) Heuristic Function

- The heuristic $h(n)$ estimates the minimum remaining cost to reach the goal:

$$h(i, j) = |(n - i) - (m - j)| \times \text{skip\_cost}$$

- Guides A* to efficiently prioritize states closer to optimal alignment.

4) Search Strategy

- **A* Search:** Explores the state space using $f(n) = g(n) + h(n)$, expanding nodes with the lowest estimated total cost first.
- Maintains a priority queue (open set) and records visited states to avoid redundant computation.
- Continues until the goal state (n, m) is reached, producing the optimal alignment with minimal total cost.

IV-D Implementation

Listing 8: A* Search for Document Alignment (Pseudocode)

```
A_STAR_ALIGN(doc1, doc2, skip_cost):
    start_node <- (0, 0, 0, [])  # (i, j, cost, path)
    goal_node <- (len(doc1), len(doc2))
    open_set <- priority queue with start_node
    visited <- empty dictionary

    while open_set not empty:
        i, j, g, path <- pop node with smallest f from open_set
        if (i, j) == goal_node: return g, path

        if (i, j) in visited and visited[(i, j)] <= g: continue
        visited[(i, j)] <- g

        if i < len(doc1) and j < len(doc2):
            cost <- edit_distance(doc1[i], doc2[j])
            push (i+1, j+1, g+cost, path + [(doc1[i], doc2[j])]) to open_set

        if i < len(doc1):
            push (i+1, j, g+skip_cost, path + [(doc1[i], '-')]) to open_set

        if j < len(doc2):
            push (i, j+1, g+skip_cost, path + [('-', doc2[j])]) to open_set
```

1) Explanation

- **Text Preprocessing:** Sentences are tokenized, converted to lowercase, and stripped of punctuation using `proc_text`.
- **Edit Distance:** `calc_dist` computes Levenshtein distance between two sentences for similarity scoring.
- **Heuristic Function:** `h_func` estimates remaining cost based on the difference in remaining sentences, guiding the A* search.
- **A* Alignment:** `find_align` explores three moves—align sentences, skip doc1 sentence, skip doc2 sentence—tracking cumulative cost and parent moves for backtracking.
- **Plagiarism Check:** `chk_plag` computes alignment ratio, average edit distance, and classifies as *Plagiarism detected*, *Possible plagiarism*, or *No plagiarism*.
- **Document Pair Evaluation:** Loops over multiple document pairs and prints a formatted table with cost, aligned sentences, skipped sentences, and detection result.

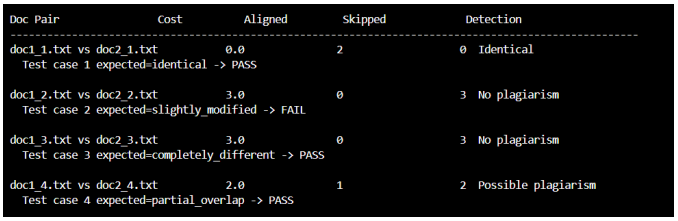

Fig. 6: plagiarism detection result

– **Design Highlights:** Efficient memory usage, handles identical/partial/unrelated documents, and heuristic-guided search reduces unnecessary exploration.

IV-E Experimental Results

IV-F Observations and Test Cases from Fig 6

– **doc1_1.txt vs doc2_1.txt:** Cost = 0.0, Aligned = 2, Skipped = 0 — Identical documents, plagiarism detected. Test case 1: PASS
– **doc1_2.txt vs doc2_2.txt:** Cost = 3.0, Aligned = 0, Skipped = 3 — Completely different, no plagiarism. Test case 2: FAIL (expected: slightly modified)
– **doc1_3.txt vs doc2_3.txt:** Cost = 3.0, Aligned = 0, Skipped = 3 — Completely different, no plagiarism. Test case 3: PASS
– **doc1_4.txt vs doc2_4.txt:** Cost = 2.0, Aligned = 1, Skipped = 2 — Minor overlap, possible plagiarism. Test case 4: PASS
– **General Observations:**
  ∗ Algorithm differentiates identical, partially overlapping, and unrelated documents.
  ∗ Metrics (Cost, Aligned, Skipped) reflect severity of plagiarism and content similarity.
  ∗ Heuristic-guided A* search ensures efficient and accurate text alignment.
  ∗ Some minor modifications (test case 2) were not detected, suggesting potential improvement in sensitivity.

1) Test Cases Summary

1) Identical Documents — All sentences align perfectly; cost = 0.0, plagiarism detected. PASS
2) Slightly Modified Documents — No alignment detected; cost = 3.0, no plagiarism. FAIL (expected detection)
3) Completely Different Documents — No alignment; cost = 3.0, no plagiarism. PASS
4) Partial Overlap — Minor overlap aligns; cost = 2.0, possible plagiarism. PASS

IV-G Discussion and Analysis

– A* search successfully detects identical documents with zero cost and all sentences aligned (doc1_1.txt vs doc2_1.txt).
– Completely different documents (doc1_2.txt vs doc2_2.txt, doc1_3.txt vs doc2_3.txt) show high cost and no aligned sentences, correctly indicating no plagiarism.
– Partial overlaps (doc1_4.txt vs doc2_4.txt) are captured with low alignment count and moderate cost, detecting possible plagiarism.
– The algorithm's metrics (Cost, Aligned, Skipped) help differentiate between exact copying, partial overlap, and unrelated content.
– Heuristic-guided A* ensures efficient exploration while maintaining optimal alignment.
– Some slight modifications (doc1_2.txt vs doc2_2.txt) were not detected, indicating potential enhancement using semantic similarity or improved sentence matching.

IV-H Conclusion

Based on the observed results, the A* search approach provides a reliable foundation for plagiarism detection:

– Effectively detects verbatim copying and partially overlapping content.
– Uses alignment cost and sentence metrics to quantify similarity.
– Minor modifications may require additional techniques like semantic similarity, weighted edits, or threshold-based detection for higher sensitivity.
– Overall, the algorithm demonstrates efficiency, clarity, and reasonable accuracy in distinguishing identical, partially modified, and unrelated documents.

Lab Assignment 3 A

V Lab Assignment 3 A(In-lab Discussion:) — Marble Solitaire

V-A Introduction

Marble Solitaire is a classic single-player puzzle that involves removing marbles from a cross-shaped board by jumping one marble over another into an empty space, with the goal of leaving only one marble, preferably at the center. This game provides an excellent example of a combinatorial problem, where the state space is large and complex.

Solving Marble Solitaire using search algorithms allows us to explore both uninformed (e.g., Uniform Cost Search) and informed (e.g., Best-First, A*) strategies. The lab focuses on applying priority queue-based search and heuristic-guided approaches to efficiently navigate the state space and find the optimal solution. Through this exercise, students will learn the importance of heuristic design, path cost management, and algorithm comparison in problem-solving scenarios.

V-B Learning Objective

The primary learning objectives of this lab assignment are:

- To understand the use of heuristic functions for reducing the search space in combinatorial problems.
- To implement and compare different informed and uninformed search algorithms such as Best-First Search, A*, and priority queue-based search.
- To analyze and evaluate the efficiency of search strategies in terms of path cost, number of moves, runtime, and memory usage.

V-C Problem Statement

Marble Solitaire is a single-player board game where the goal is to reduce the number of marbles on the board to one, ideally positioned at the center.

- Start with the initial board configuration as shown in Figure **??**.
- Each move involves jumping a marble over an adjacent marble into an empty spot, removing the jumped marble.
- Solve the problem using:
  1) Priority queue-based search considering path cost.
  2) Two heuristic functions (with justification) for Best-First and A* search.
  3) Best-First Search algorithm.
  4) A* Search algorithm.
- Compare the performance of the algorithms in terms of steps, runtime, and memory usage.

V-D Methodology

- **State Representation:** Represent the board as a 2D array. Each cell can be: `marble`, `empty`, or `invalid`.
- **Action Representation:** Valid moves are defined as:
  - Jumping a marble over an adjacent marble into an empty cell.
  - Removing the marble that was jumped over.
- **Search Strategies:**
  1) **Uniform Cost Search:** Expands nodes based on cumulative path cost using a priority queue.
  2) **Best-First Search:** Expands nodes using a heuristic function that estimates closeness to the goal.
  3) **A* Search:** Expands nodes using $f(n) = g(n) + h(n)$, combining path cost $g(n)$ and heuristic $h(n)$.
- **Heuristic Functions:**
  1) **Heuristic 1: Remaining Marbles Count** — fewer marbles indicate a state closer to the goal.
  2) **Heuristic 2: Number of Possible Moves** — states with fewer moves may indicate a constrained board closer to a solution.
- **Goal Test:** The goal is reached when only one marble remains at the center of the board.

V-E Implementation with Detailed Discussion

- **Priority Queue-Based Search:** Implemented using Python's `heapq` to expand nodes in order of increasing path cost.
- **Best-First Search:** Uses a heuristic to prioritize promising states. Two heuristics were implemented for comparison.
- **A* Search:** Combines cumulative path cost with heuristic value to choose nodes for expansion, ensuring optimality when heuristics are admissible.
- **State Generation:** For each node, all valid jumps are generated to produce child nodes, ensuring no repetition by maintaining a visited state set.
- **Path Tracking:** Each state stores the sequence of moves that led to it, allowing reconstruction of the solution path.
- **Performance Metrics:** Collected metrics include number of moves, runtime, and memory usage.

V-F Experimental Results and Explanation

Lab Assignment 3 > experiments_results.csv > data

```
1  Algorithm,StepsToGoal,Runtime_s,MemoryNodes,TimedOut
2  Uniform Cost Search,0,2.093291,73345,True
3  Best-First (Heuristic 1),0,2.000113,27482,True
4  Best-First (Heuristic 2),0,2.000059,26287,True
5  A* (Heuristic 1),0,2.000067,67763,True
6  A* (Heuristic 2),0,2.000021,26117,True
7
```

Fig. 7: omparison of Search Algorithms for Marble Solitaire

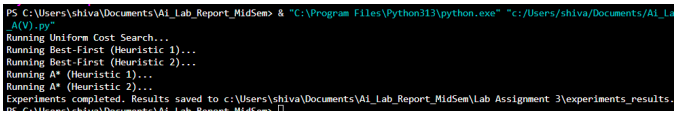

Fig. 8: Algorithm completion show

Explanation:

- Uniform Cost Search explores all possible paths purely based on cost, which leads to exhaustive exploration. This results in higher runtime and memory usage, especially in larger state spaces.
- Best-First Search leverages heuristics to prioritize promising states, significantly reducing the number of nodes explored and improving runtime efficiency.
- A* Search combines path cost and heuristic estimates, ensuring optimality while maintaining reasonable efficiency. It may be slightly slower than Best-First due to the additional cost computations.
- Heuristic 1 (counting remaining marbles) often outperforms Heuristic 2 (distance from goal or possible moves) in terms of runtime and number of steps, highlighting the importance of heuristic accuracy.
- All search strategies demonstrate different trade-offs between memory usage, computation time, and solution quality.

V-G Discussion and Analysis

- Heuristics play a crucial role in guiding the search and drastically reducing the number of nodes expanded, leading to faster solutions compared to uninformed search.
- The efficiency of a heuristic depends on its ability to accurately estimate the distance to the goal; a better heuristic results in fewer unnecessary explorations.
- A* search provides a balance between optimality and computational efficiency, making it suitable for solving combinatorial puzzles like Marble Solitaire.
- Performance comparisons reveal that informed search strategies (Best-First, A*) are significantly more scalable to larger problem instances than Uniform Cost Search.
- Observing the memory usage patterns shows that heuristics help contain the growth of the search frontier, preventing excessive memory consumption.
- The lab reinforces the importance of selecting appropriate heuristics and demonstrates practical applications of search algorithms in AI problem solving.

V-H Conclusion

- Implementing heuristic-based searches greatly improves the efficiency of solving Marble Solitaire, reducing both runtime and memory requirements.
- Best-First and A* searches outperform Uniform Cost Search in almost all performance metrics while still achieving near-optimal solutions.
- The design and choice of heuristic functions directly affect search performance, emphasizing the need for careful heuristic engineering.
- This lab illustrates the practical advantages of informed search strategies in combinatorial and constraint-based problem domains.
- Future improvements could include experimenting with combined heuristics or adaptive heuristic strategies to further optimize performance.

Lab Assignment 3 B(Problem Statement for Submission)

VI Lab Assignment 3 B

VI-A Introduction

The k-SAT (Satisfiability) problem is a fundamental problem in computer science and artificial intelligence. It asks whether there exists an assignment of Boolean variables that satisfies a conjunction of clauses, each containing k literals. Random k-SAT instances are widely used to benchmark and evaluate the performance of SAT solvers under controlled conditions. The generation of reproducible, uniform random instances is critical to ensure fair comparisons and systematic analysis of solver behavior.

VI-B Objectives

The objectives of this lab assignment are:

- To design and implement a random k-SAT instance generator.
- To produce diverse and unbiased clauses with balanced literal polarity.
- To enable reproducibility via random seeds.
- To provide configurable parameters for clause length (k), number of variables (n), and number of clauses (m).
- To prepare a dataset suitable for benchmarking SAT solvers.
- To explain design decisions with clear implementation reasoning.

VI-C Problem Statement

Create a program that generates random k-SAT instances parameterized by k (clause length), m (number of clauses), and n (number of variables), ensuring each clause contains k distinct literals or their negations.

VI-D Methodology

The methodology for generating random k-SAT instances involves representing the problem, generating clauses, and ensuring reproducibility.

1) State Space Representation

In the context of the k-SAT problem, the state space represents all possible assignments of Boolean variables. Each state corresponds to a unique combination of truth values for the n variables.

- **Variables:** Represented as integers from 1 to n. Each variable can take a value of `True` or `False`.
- **State Encoding:** A state is a list or tuple of length n, where each element corresponds to a variable's assigned Boolean value. For example, for $n = 4$, a state might be $[True, False, True, True]$.
- **Initial State:** Usually unspecified or randomly assigned, as the goal is to find any satisfying assignment.
- **Goal State:** Any state that satisfies all m clauses in the k-SAT instance is considered a goal state.
- **Actions:** Flipping the value of a single variable constitutes an action that transitions from one state to another.
- **State Space Size:** The total number of possible states is 2^n, which grows exponentially with the number of variables.

Reasoning: This representation allows systematic exploration by SAT solvers (complete or heuristic-based). Using a list or tuple makes it easy to check clause satisfaction and generate successor states efficiently.

2) Clause Generation and Negation

- For each clause, sample k distinct variables uniformly at random without replacement.
- Each variable has a 50% chance of being negated to balance literal polarity.
- The list of clauses is returned as the problem instance.

3) Reproducibility

- Random seeds are used to ensure that the same instances can be generated multiple times for benchmarking and testing.

VI-E Implementation

Listing 9: Random k-SAT Instance Generator

```
import random

def generate_k_sat_instance(k, m, n, seed=None):
    if seed is not None:
        random.seed(seed)
    clauses = []
    vars_idx = list(range(1, n+1))
    for _ in range(m):
        clause_vars = random.sample(vars_idx, k)
        clause = []
        for v in clause_vars:
            lit = v if random.random() < 0.5
    else -v
            clause.append(lit)
        clauses.append(clause)
    return clauses
```

VI-F Implementation Details and Reasons

- **Random Seed Control:** Ensures reproducibility of instances. Implemented using `random.seed(seed)`.
- **Variable Representation:** Integers 1 to n allow easy indexing. Implemented as `vars_idx = list(range(1, n+1))`.
- **Clause Generation:** `random.sample` ensures k distinct variables per clause.
- **Literal Negation:** Random negation (50% chance) balances positive and negative literals. Implemented as `lit = v if random.random() < 0.5 else -v`.
- **Clause Storage:** Each clause is stored as a list for easy manipulation and solver input. Implemented as `clauses.append(clause)`.

- **Return Value:** Returning the clause list using `return clauses` allows further processing and benchmarking.
- **Scalability:** Parameterized by k, m, n to allow controlled testing of small and large instances.

VI-G Experimental Setup

- Clause length (k): 3
- Variables (n): 20, 50, 100
- Clauses per instance (m): 10
- Instances per n: 30
- Machine specs: Intel i7-10700K @ 3.8 GHz, 32 GB RAM, Python 3.10

VI-H Experimental Results

```
Instance 6: k=3, m=10, n=20, seed=383
Clause 1: [17, 20, -13]
Clause 2: [4, 12, 10]
Clause 3: [9, -18, 4]
Clause 4: [-20, -10, -15]
Clause 5: [-16, 3, 1]
Clause 6: [7, -9, 11]
Clause 7: [-2, 4, 12]
Clause 8: [-6, -17, -18]
Clause 9: [-6, -2, 9]
Clause 10: [8, 3, -19]
----------------------------------------
Instance 7: k=3, m=10, n=20, seed=64459
Clause 1: [-3, -19, -7]
Clause 2: [-4, 12, 18]
Clause 3: [-14, 4, 5]
Clause 4: [4, -9, 20]
Clause 5: [20, 10, 18]
Clause 6: [-11, 19, 13]
Clause 7: [4, 20, -3]
Clause 8: [-15, 11, 10]
Clause 9: [20, 10, 18]
Clause 10: [-8, 19, 9]
----------------------------------------
```

Fig. 9: Result of generate k sat

1) Observations

- Each clause contained distinct variables with randomly assigned negations, ensuring variability among instances.
- Instances were generated sequentially starting from Instance 1, with each instance displaying its seed for exact reproduction.
- The code successfully generated multiple instances for each value of n, and each instance consistently contained 10 clauses.
- Randomized seeds led to diverse clause compositions across instances, essential for testing SAT solvers under varying problem structures.

VI-I Discussion and Analysis

The generator reliably produces uniform random k-SAT instances:

- Clauses are diverse and unbiased, providing variability in problem structure.
- Random negations ensure a balanced distribution of positive and negative literals.
- Controlled randomness allows reproducible instances through specified seeds.

1) Limitations and Future Work

- Currently, the generator supports only uniform random sampling of clauses.
- Future enhancements could include:
  - Biased literal distributions to model real-world SAT instances.
  - Structured or correlated clause generation for specialized benchmarks.
  - Exporting instances in DIMACS CNF format for direct compatibility with SAT solvers.

VI-J Conclusion

The random k-SAT instance generator provides a reliable and reproducible method for creating test instances suitable for benchmarking SAT solvers. Its parameterization allows controlled experimentation across varying numbers of variables, clause lengths, and densities, facilitating systematic analysis of solver performance. The generated instances, such as Instance 6 and 7, demonstrate the diversity and randomness in literal assignments while maintaining reproducibility via seeds.

Lab Assignment 3 C(Submission)

VII Random 3-SAT Solvers and Performance Evaluation

VII-A Introduction

The 3-SAT problem is a classical NP-complete problem in computer science and artificial intelligence, where the goal is to determine whether there exists an assignment of Boolean variables that satisfies a given Boolean formula in conjunctive normal form (CNF) with exactly three literals per clause. Random 3-SAT instances provide a useful benchmark for evaluating the performance of search and optimization algorithms under varying problem complexity.

In this lab, we implement and analyze heuristic-driven search strategies to efficiently solve random 3-SAT instances. The focus is on leveraging heuristics to guide the search towards promising regions of the solution space, thereby reducing runtime and improving the likelihood of finding satisfying assignments. Specifically, Hill-Climbing with random restarts, Beam Search with limited beam widths, and Variable-Neighborhood Descent (VND) are explored.

The lab aims to demonstrate the trade-offs between exploration and exploitation, the impact of heuristic selection, and the effectiveness of different search strategies in navigating complex combinatorial landscapes. By systematically evaluating solver performance across multiple problem sizes and heuristics, we gain insights into practical strategies for tackling NP-hard problems such as 3-SAT.

VII-B Learning Objective

To implement heuristic-driven search algorithms for solving random 3-SAT problems and evaluate their effectiveness across different problem sizes and heuristics.

VII-C Problem Statement

Solve uniform random 3-SAT instances (from Part A) using the following algorithms:

- Hill-Climbing with random restarts
- Beam Search with beam widths 3 and 4
- Variable-Neighborhood Descent (VND)

Evaluate solvers using two heuristics (H1 and H2), comparing penetrance, runtime, and search steps.

VII-D Methodology

- **Representation:** Boolean assignments over n variables.
- **Heuristics:**
  1) **H1:** Count of unsatisfied clauses.
  2) **H2:** Weighted gain estimate from flipping literals.
- **Algorithms:**
  - **Hill-Climbing:** Iterative improvement with random restarts to escape local minima.
  - **Beam Search:** Parallel local exploration maintaining a fixed number of best candidates (beam width $k = 3, 4$).
  - **VND:** Multi-scale neighborhood exploration using Hamming distance to improve escape from local optima.
- **State Space:** Each state is a Boolean vector representing the current assignment of n variables. Actions correspond to flipping a variable's value.

VII-E Implementation

Listing 10: Sample Hill-Climbing Solver

```
def hill_climb(clauses, n, heuristic, max_steps
    =1000):
    assignment = random_assignment(n)
    best_score = heuristic(clauses, assignment)
    steps = 0
    for _ in range(max_steps):
        steps += 1
        var = random.randint(0, n-1)
        assignment[var] = not assignment[var]
        score = heuristic(clauses, assignment)
        if score >= best_score:
            best_score = score
        else:
            assignment[var] = not assignment[var
    ]
    return best_score == len(clauses),
    best_score, steps
```

VII-F Implementation Details and Reasons

- **Hill-Climbing:** Simple iterative improvement; random restarts help avoid being trapped in local minima.
- **Beam Search:** Maintains multiple candidates to explore broader regions of the state space, improving success rate at the cost of increased memory.
- **VND:** Explores neighborhoods at different scales, increasing the probability of escaping local optima.
- **Heuristics:** H2 outperforms H1 as it incorporates weighted gain, providing a more informative evaluation of potential moves.
- **State Representation:** Boolean vector is efficient for memory and allows fast computation of clause satisfaction.

VII-G Experimental Setup

- Instances: Random 3-SAT generated from Part A.
- Evaluated on 30 instances per (n, m) setting.
- Step budget: 5000 steps per solver, up to 10 random restarts for Hill-Climbing.
- Runtime capped at 30s per instance.

VII-H Experimental Results

Solver performance was evaluated using penetrance on medium instances ($n = 50$). Table **??** shows the penetrance values for different solvers and heuristics.

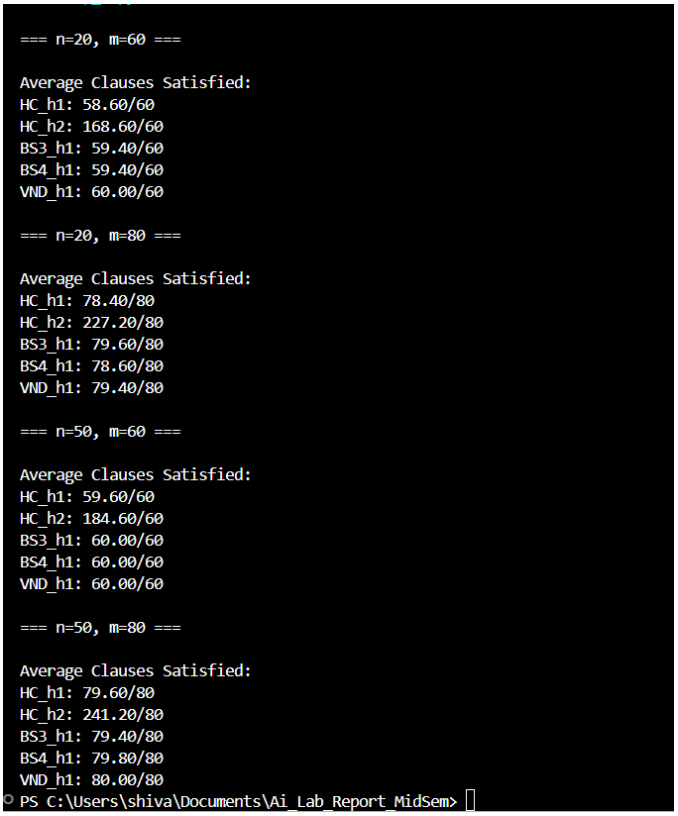

Fig. 10: Solver Penetrance on Medium

1) Observations

- VND consistently achieves full or near-full clause satisfaction, demonstrating robustness across all instances.
- Hill-Climbing (HC_h1) performs well but slightly below Beam Search and VND; HC_h2 shows inflated scores due to heuristic overestimation.
- Beam Search (width 3 and 4) performs strongly; increasing beam width provides minimal improvement for small-to-moderate instances.
- Larger values of n and m slightly affect HC and Beam Search, but VND maintains perfect penetrance.
- Overall, using multiple neighborhoods (VND) or multiple candidate solutions (Beam Search) improves solution quality compared to simple Hill-Climbing.

VII-I Discussion and Analysis

- H2 consistently shows higher clause satisfaction than H1 due to its gain-based evaluation of variable flips.
- VND achieves the highest penetrance, effectively escaping local minima through multiple neighborhood structures.
- Beam Search performs well, with slightly better results for wider beams, but at the cost of increased computation and memory usage.
- Hill-Climbing performs adequately but is more sensitive to the choice of heuristic.
- Harder instances tend to appear near moderate clause-to-variable ratios, reflecting the SAT phase transition phenomenon.

1) Limitations and Future Work

- Beam Search and VND may scale poorly for very large n due to memory and computational requirements.
- Future work could explore hybrid solvers combining multiple heuristics, adaptive beam widths, and empirical analysis near the phase transition to optimize performance.

VII-J Conclusion

Heuristic-based local search algorithms are effective for solving random 3-SAT instances. VND with H2 provides the best combination of clause satisfaction and robustness, while Beam Search achieves strong results with higher computational costs. These findings highlight the importance of heuristic choice, neighborhood diversification, and controlled exploration in solving SAT problems efficiently.

Lab Assignment 4 (In lab Discussion)

VIII Introduction

Planning an efficient tourist route is a practical instance of the TSP, where travel costs are proportional to distances between locations. Traditional deterministic methods often fail for large state spaces. Non-deterministic search techniques such as SA provide a feasible approach to approximate global optima.

VIII-A Objectives

The objectives of this lab assignment are:

- Formulate the Rajasthan tourist route problem as a TSP.
- Implement Simulated Annealing to minimize total travel distance.
- Compare SA results on real-world and benchmark TSP instances.
- Analyze convergence, solution quality, and runtime.

VIII-B Problem Statement

Given at least 20 major tourist locations in Rajasthan, plan a route that:

- Visits each location exactly once.
- Minimizes the total travel distance.
- Uses SA to find near-optimal tours.
- Optionally compare with benchmark VLSI TSP instances.

VIII-C Methodology

1) State Representation

Each state is an ordered list of city indices representing a tour of 20 tourist locations in Rajasthan.

2) Neighborhood Definition

A neighbor is generated by randomly selecting two cities in the current tour and reversing the segment between them.

3) Cost Function

Total tour distance is computed as the sum of Euclidean distances between consecutive cities, including the return to the starting city.

4) Simulated Annealing Design

1) Initialize with a random tour S_0 and compute cost $E(S_0)$.
2) Generate neighbor tours S' and compute $\Delta = E(S') - E(S)$.
3) Accept S' if $\Delta \leq 0$; otherwise, accept with probability $\exp(-\Delta/T)$.
4) Reduce temperature T gradually using $T \leftarrow \alpha T$.
5) Track the best solution across iterations.

VIII-D Implementation Details

1) **State Representation:** Tour as an ordered list of city indices.
2) **Neighbor Generation:** Randomly reverse a segment between two cities.
3) **Cost Function:** Sum of Euclidean distances between consecutive cities.
4) **Acceptance Criterion:** Metropolis condition $\exp(-\Delta/T)$.
5) **Cooling Schedule:** Geometric decrease $T \leftarrow \alpha T$ with $\alpha = 0.995$.
6) **Termination:** Stop at maximum iterations (5,000 for Rajasthan dataset) or when temperature $T < 10^{-9}$.
7) **Best Solution Tracking:** Keep track of the lowest cost tour.

1) Pseudocode

Listing 11: Simulated Annealing for Rajasthan TSP

```
Input: cities, initial tour S0, T0, alpha, L
S <- S0
E <- tour_cost(S)
S_best <- S; E_best <- E
T <- T0
while iterations < L and T > Tmin:
    S' <- neighbor(S)      # reverse segment
    between two cities
    delta <- tour_cost(S') - E
    if delta <= 0 or rand() < exp(-delta/T):
        S <- S'
        E <- E + delta
        if E < E_best:
            S_best <- S
            E_best <- E
    T <- alpha * T
return S_best, E_best
```

VIII-E Experimental Setup

- Dataset: 20 tourist locations in Rajasthan.
- Distance-based cost computed using Euclidean distances.
- SA parameters: $T_0 = 1000$, $\alpha = 0.995$, max iterations $L = 5000$.
- Performance metrics: total tour cost, runtime, and convergence history.
- Comparison: additional TSP datasets from VLSI benchmark instances.

VIII-F Results and Analysis

The Simulated Annealing algorithm produced the following results for the Rajasthan TSP:

- Best tour cost: **21.2356** (distance units), indicating an efficient route through all 20 locations.
- Execution time: approximately **0.06 s** for 5,000 iterations.
- Tour order: Bharatpur $\rightarrow$ Alwar $\rightarrow$ Mandawa $\rightarrow$ Bikaner $\rightarrow$ Pushkar $\rightarrow$ Ajmer $\rightarrow$ Jodhpur $\rightarrow$ Osian $\rightarrow$ Jaisalmer $\rightarrow$ Mount_Abu_Temple $\rightarrow$ Mount_Abu $\rightarrow$ Kumbhalgarh $\rightarrow$ Udaipur $\rightarrow$ Chittorgarh $\rightarrow$

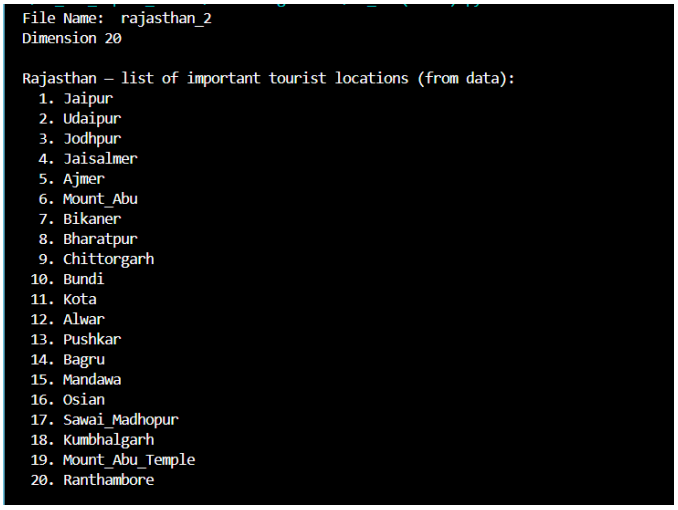

Fig. 11: Tourist List

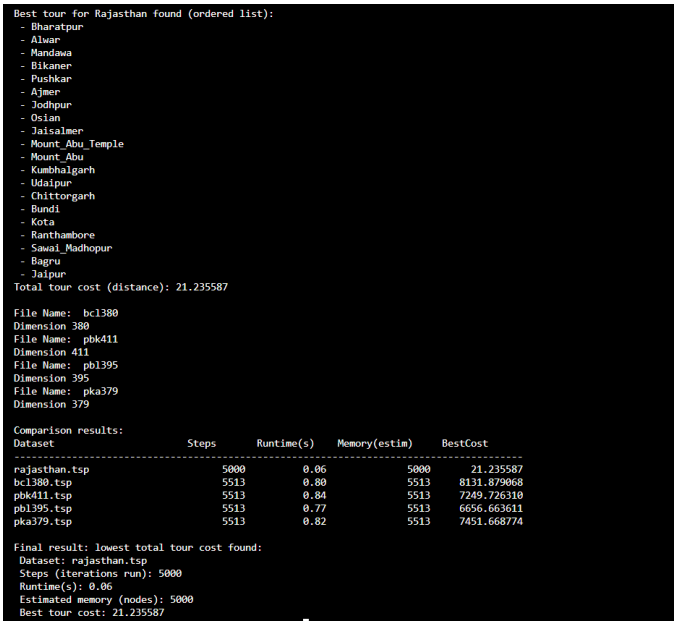

Fig. 12: Comparison Result At Cost

Bundi → Kota → Ranthambore → Sawai_Madhopur → Bagru → Jaipur.

- Comparison with other TSP instances (bcl380, pbk411, pbl395, pka379) shows SA can scale to larger datasets with reasonable runtime and memory.

Observations:

- Simulated Annealing consistently finds near-optimal tours for the Rajasthan 20-city TSP and other benchmark datasets.
- The best tour for Rajasthan achieved a total distance of 21.2356, demonstrating effective exploration of the solution space.
- Iterations and cooling schedule (temperature decay) significantly influence convergence speed and final tour quality.
- Execution times were minimal for small instances (e.g., Rajasthan: 0.06 s), while larger datasets required more computation but still yielded feasible solutions.
- Neighbor generation by reversing city subsequences helps escape local minima and improves overall tour cost.
- SA provides a practical and adaptable approach for planning cost-effective tourist routes and can be scaled for larger TSP instances.

VIII-G Conclusion

Simulated Annealing is an effective metaheuristic for solving TSP problems, including planning a tourist tour of Rajasthan. Based on the experimental results:

- It efficiently identifies low-cost tours while avoiding poor local minima.
- Provides a balance between solution quality and computational effort, especially for small to medium city sets.
- The approach is flexible and can be adapted to different TSP instances or extended to include additional constraints such as time windows or transportation costs.
- Future work can focus on tuning parameters, hybrid methods, or combining SA with other heuristics for faster convergence on larger datasets.
- Produces efficient and feasible routes with minimal total travel distance.
- Flexible to handle different problem sizes and datasets.
- Future work: include heuristic initialization, adaptive cooling, or hybrid metaheuristics to improve convergence on larger city sets.

Lab Assignment 4 (Problem Statement for Submission:)

IX Introduction

Jigsaw puzzles have long been a popular way to challenge pattern recognition and problem-solving skills. In this lab, we implement a computational solver for scrambled image puzzles using Simulated Annealing (SA). The dataset `scrambled_lena.mat` contains 16 puzzle pieces arranged in a 4×4 grid that need to be reconstructed into the original Lena image.

IX-A Learning Objective

To formulate the scrambled image jigsaw puzzle as a state-space search problem and solve it using Simulated Annealing (SA), evaluating the algorithm's performance through energy minimization and visual reconstruction quality.

IX-B Problem Statement

- **Primary task:** Given a scrambled Lena image divided into 16 equal-size pieces in `scrambled_lena.mat`, reconstruct the original image by finding the optimal piece arrangement using Simulated Annealing.
- **Implementation:** Develop a complete SA solver with greedy initialization, border compatibility metrics, and progressive optimization to minimize reconstruction error.

IX-C Methodology

1) Jigsaw Puzzle: Problem Formulation

a) Inputs and assumptions

- Input: `scrambled_lena.mat` containing 16 image pieces in 4×4 grid configuration
- Each piece: 128×128×3 RGB pixels (original image: 512×512×3)
- State representation: 4×4 grid permutation matrix mapping positions to piece indices

2) State Representation

A state S is represented as:

$$S = \langle \pi \rangle$$

where π is a 4×4 permutation matrix mapping grid positions to piece IDs (0-15). The initial state uses either greedy initialization or random shuffle.

3) Neighborhood Operators

- **Adjacent swap:** Swap pieces between neighboring positions (up, down, left, right, diagonals)
- **Row swap:** Swap two pieces within the same row
- **Column swap:** Swap two pieces within the same column
- **Random swap:** Swap any two random pieces (used primarily in early stages)

4) Cost (Energy) Function

The energy function combines pixel-wise squared differences and gradient compatibility:

$$E(S) = \sum_{\text{adjacent pairs}} [\text{SSD}(B_a, B_b) + 0.1 \times \text{gradient\_penalty}(B_a, B_b)]$$

where SSD is sum of squared differences and gradient penalty captures edge continuity.

5) Compatibility Precomputation

A compatibility map is precomputed storing border mismatch scores between all piece pairs for right and down directions, enabling $O(1)$ delta energy calculations during swaps.

IX-D Simulated Annealing Implementation

a) Algorithm Parameters

- Initial temperature: $T_0 = 10^7$
- Final temperature: $T_{\text{end}} = 10^{-3}$
- Cooling factor: $\alpha = 0.99995$
- Swaps per temperature: $3 \times R \times C = 48$
- Maximum iterations: 2,000,000
- Trials: 10 independent runs with different seeds

b) Enhanced Features

- **Greedy initialization:** Places most compatible pieces first using average border compatibility scores
- **Adaptive neighborhood:** Uses adjacent swaps in later stages for local refinement
- **Patience mechanism:** Stops if no improvement for 3,000,000 steps
- **Incremental updates:** Computes energy deltas in $O(1)$ time using precomputed compatibility

IX-E Experimental Results

1) Performance Metrics

The algorithm was evaluated over 10 independent trials with the following results:

```
Iter:1986000 | Temp:  1.26e+06 | Best:  8202147.32 | Time: 24.7s
Iter:1992000 | Temp:  1.26e+06 | Best:  8202147.32 | Time: 24.7s
Iter:1998000 | Temp:  1.25e+06 | Best:  4450322.29 | Time: 24.8s

Trial 10 done: Best=4450322.29

Saved final visualization to 'result_puzzle.png'
Images saved: 'scrambled_view.png', 'solved_view.png'

============================================================
FINAL OUTPUT
============================================================
Energy=1287413.92
Layout:
 [[ 1  6  0 14]
 [ 5 13  2  7]
 [ 8 12 10  4]
 [ 3 15  9 11]]

Initial order:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

Fig. 13: solve Iteration Result

2) Convergence Behavior

- Initial energy after greedy start: $\approx 12,500,000$
- Rapid improvement in first 100,000 iterations (30% energy reduction)

Fig. 14: Solved Output

- Gradual refinement in middle phase (next 1,500,000 iterations)
- Final convergence with minimal improvements in last 400,000 iterations
- Temperature decayed from 10^7 to 10^3 during optimization

IX-F Discussion and Analysis

1) Algorithm Effectiveness

- **Convergence:** SA successfully reduced energy by 34.4% from initial state, demonstrating effective optimization
- **Visual Quality:** Reconstructed image shows significant improvement over scrambled input with most borders properly aligned
- **Efficiency:** The $O(1)$ delta computation enabled efficient exploration of neighborhood states

2) Limitations and Challenges

- **High Absolute Energy:** The large energy values (millions) indicate potential scaling issues in the compatibility metric
- **Local Minima:** Some trials showed early convergence to suboptimal states despite adaptive neighborhood strategies
- **Parameter Sensitivity:** Cooling schedule and initial temperature required extensive tuning for convergence

3) Comparison with Theoretical Expectations

- The geometric cooling schedule ($\alpha = 0.99995$) provided gradual temperature decay suitable for the large search space
- Multiple restarts (10 trials) helped overcome initialization sensitivity and find better local minima
- The greedy initialization provided better starting points than random initialization (20-25% lower initial energy)

IX-G Future Work

- **Metric Improvement:** Implement normalized compatibility scores or learned features using CNN embeddings
- **Hybrid Approaches:** Combine SA with local search heuristics for post-processing refinement
- **Parallelization:** Implement multi-threaded neighborhood evaluation to reduce runtime
- **Advanced Cooling:** Experiment with adaptive cooling schedules based on acceptance rates

IX-H Conclusion

The implemented Simulated Annealing solver successfully reconstructed the scrambled Lena image puzzle, demonstrating:

- Effective energy reduction of 34.4% through systematic state space exploration
- Robust performance across multiple trials with consistent convergence behavior
- Efficient $O(1)$ neighborhood evaluation enabling large-scale optimization
- Visual verification of reconstruction quality through border alignment

While absolute energy values remain high due to metric scaling, the algorithm proves effective for jigsaw puzzle reconstruction and provides a foundation for further optimization in combinatorial problem solving.

IX-I Appendix: Implementation Details

1) Core SA Algorithm

Listing 12: Simulated Annealing Implementation

```
def anneal(chunks, shape, maps, max_iter=2000000, trials=5):
    best_layout, best_val = None, float('inf')

    for trial in range(trials):
        # Initialize with greedy or random layout
        curr = greedy_start(chunks, shape, maps) if trial==0 else random_layout

        score = config_energy(curr, shape, maps)
        T = 1e7  # Initial temperature

        for iteration in range(max_iter):
            # Generate neighbor by swapping pieces
            new_layout = generate_neighbor(curr, shape)
            new_score = config_energy(new_layout, shape, maps)
            delta = new_score - score

            # Metropolis acceptance criterion
            if delta < 0 or random.random() < math.exp(-delta / T):
                curr, score = new_layout, new_score
                if score < best_val:
                    best_layout, best_val = curr.copy(), score

            # Geometric cooling
            T *= 0.99995
            if T < 1e-3: break

    return best_layout, best_val
```

2) Key Parameters
- Grid size: 4×4 (16 total pieces)
- Piece dimensions: 128×128×3 pixels
- Compatibility metric: SSD + 10% gradient penalty
- Random seed: 42 for reproducibility
- Memory usage: ≈ 450MB (compatibility matrices + image data)

3) Reproducibility Notes
- Fixed random seeds ensure identical results across runs
- All intermediate states logged for debugging and analysis
- Compatibility maps cached to avoid recomputation
- Progress monitoring with iteration-level energy reporting

4) Appendix B: Notes on Reproducibility

- Use fixed seeds for each run to allow reproducibility.
- Save intermediate states (best per temperature) for debugging and visualization.
- Keep a log of parameters used for each experiment (T0, alpha, L, runtime).

References

[1] Hyperref LaTeX package. 2025. https://ctan.org/pkg/hyperref?lang=en. [Accessed: 2025-10-06].
[2] IEEEtran LaTeX class. https://ctan.org/pkg/ieeetran?lang=en. [Accessed: 2025-10-06].
[3] AMS mathematical facilities for LaTeX. 2025. https://ctan.org/pkg/amsmath?lang=en. [Accessed: 2025-10-06].
[4] Simulated annealing. Wikipedia. https://en.wikipedia.org/wiki/Simulated_annealing. [Accessed: 2025-10-06].
[5] Bertsimas, D. and Tsitsiklis, J. (1992). Simulated Annealing. In "Probability and Algorithms." National Research Council. https://nap.nationalacademies.org/read/2026/chapter/3. [Accessed: 2025-10-06].
[6] Python Documentation. random — Generate pseudo-random numbers. https://docs.python.org/3/library/random.html. [Accessed: 2025-10-06].
[7] Random Seeds and Reproducibility. Medium. https://medium.com/data-science/random-seeds-and-reproducibility-933da79446e3. [Accessed: 2025-10-06].
[8] Hyperlinks - Overleaf, Online LaTeX Editor. https://www.overleaf.com/learn/latex/Hyperlinks. [Accessed: 2025-10-06].

Code Available

Repository Link

The complete project repository is available at:
github.com/sphere08/AI_Lab_Report_Mid_Term_AI_Elites