Artificial Intelligence (CS367) Lab Report

Group Name: $Synapse^3$
Arpit Jain (202311016), Saransh Naik (202311077), Param Sonawane (202311082)

***Abstract*—This work investigates a spectrum of artificial intelligence search and optimization techniques. Uninformed search algorithms, specifically Breadth-First Search (BFS) and Depth-First Search (DFS), are implemented to solve the Rabbit Leap problem, followed by a comparative analysis of solution optimality and computational complexity. The study progresses to informed search, where the A* algorithm is applied to develop a plagiarism detection system via text alignment, demonstrating the efficacy of heuristic functions. Local search methods, including Hill-Climbing, Beam Search, and Variable-Neighborhood Descent, are utilized to find solutions for the 3-SAT problem. Finally, the application of Simulated Annealing, a stochastic optimization technique, is explored for solving large-scale problems such as the Traveling Salesman Problem (TSP) and a jigsaw puzzle. This report highlights the practical application of these diverse paradigms, emphasizing the importance of problem modeling, heuristic design, and the trade-offs between solution quality and computational resources.**

INTRODUCTION

This report provides a practical exploration of core artificial intelligence (AI) concepts through hands-on experiments involving search algorithms, optimization techniques, and probabilistic models.

1) Missionaries and Cannibals
2) Rabbit Leap
3) Puzzle-8
4) Plagiarism Detection
5) Marble Solitaire
6) k-SAT Problem
7) 3-SAT Problem
8) Tour of Rajasthan
9) Jigsaw Puzzle
10) Raag Bhairav

ASSIGNMENT 1

I. MISSIONARIES AND CANNIBALS PROBLEM

A. Objective

The objective of this problem is to model the **Missionaries and Cannibals Problem** as a state space search problem and solve it using **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**.

The task is to determine a safe sequence of boat crossings that transfers all missionaries and cannibals across the river without violating the safety constraints.

B. Problem Statement

Three missionaries and three cannibals are positioned on the left bank of a river, along with a boat capable of carrying either one or two individuals at a time.

The objective is to transport all six individuals safely to the right bank of the river. The boat may be rowed by either a missionary or a cannibal, but the following restriction must always be observed:

At no point on either bank should the number of cannibals exceed the number of missionaries, otherwise the missionaries will be at risk.

The task is to determine a valid sequence of crossings that achieves the goal state while ensuring the safety of the missionaries throughout the process.

Constraints:

1) The boat can carry either one or two people.
2) If at any point on either side of the river, the cannibals outnumber the missionaries, the missionaries will be eaten, and the game is lost.
3) The goal is to transport everyone safely to the other side of the river.

C. Methodology

State Representation:

- The state of the problem is represented by a tuple **(M, C, B)**, where:
  - **M**: Number of missionaries on the left side (0, 1, 2, or 3).
  - **C**: Number of cannibals on the left side (0, 1, 2, or 3).
  - **B**: Position of the boat (0: Left bank, 1: Right bank).
- **Initial State:** (3, 3, 0)
- **Goal State:** (0, 0, 1)
- **Total Possible States:** Considering the possibilities:
  - Missionaries: 0, 1, 2, or 3 $\rightarrow$ 4 possibilities
  - Cannibals: 0, 1, 2, or 3 $\rightarrow$ 4 possibilities
  - Boat: 0 or 1 $\rightarrow$ 2 possibilities

$$\Rightarrow 4 \times 4 \times 2 = 32 \text{ possible states}$$

State Transition Model:

The problem is modeled as a series of valid moves that transfer missionaries and/or cannibals across the river using the boat. The permissible moves are:

- Transfer two missionaries.
- Transfer two cannibals.
- Transfer one missionary and one cannibal together.
- Transfer a single missionary.
- Transfer a single cannibal.

Validation Function:

A validation function is applied to verify the correctness of each state. It ensures that the safety condition is never violated, i.e., at no point on either riverbank the number of cannibals exceed the number of missionaries.

D. Search Strategies

Two classical search strategies are used to solve the Missionaries and Cannibals problem:

- **Breadth-First Search (BFS):** Explores states level by level, ensuring the first solution found is optimal with the fewest moves.

- **Depth-First Search (DFS):** Explores one path as deeply as possible before backtracking. DFS can find a solution faster, but it does not guarantee optimality.

Algorithm 1 Breadth-First Search (BFS)

Input: Start state $start\_state$, Goal state $goal\_state$
Output: Path to goal or "No solution found"
Initialize queue $\leftarrow$ **enqueue** $(start\_state, [])$
Initialize visited $\leftarrow \emptyset$
while queue not empty **do**
  $(current\_state, path) \leftarrow$ **dequeue** from queue
  if $current\_state = goal\_state$ **then**
    Return path
  end if
  if $current\_state \notin visited$ **then**
    $visited \leftarrow visited \cup \{current\_state\}$
    for each successor of $current\_state$ **do**
      if successor is valid $\wedge$ successor $\notin$ $visited$ **then**
        Enqueue $(successor, path + [successor])$
      end if
    end for
  end if
end while
Return "No solution found" =0

Algorithm 2 Depth-First Search (DFS)

Input: Start state $start\_state$, Goal state $goal\_state$
Output: Path to goal or "No solution found"
Initialize stack $\leftarrow$ **push** $(start\_state, [])$
Initialize visited $\leftarrow \emptyset$
while stack not empty **do**
  $(current\_state, path) \leftarrow$ **pop** from stack
  if $current\_state = goal\_state$ **then**
    Return path
  end if
  if $current\_state \notin visited$ **then**
    $visited \leftarrow visited \cup \{current\_state\}$
    for each successor of $current\_state$ **do**
      if successor is valid $\wedge$ successor $\notin$ $visited$ **then**
        Push $(successor, path + [successor])$ onto stack
      end if
    end for
  end if
end while
Return "No solution found" =0

E. Analysis

Breadth-First Search (BFS):

The sequence of moves generated by BFS represents the **shortest possible solution**, ensuring that all missionaries and cannibals are safely transported across the river while strictly adhering to all constraints. The output of the problem using BFS is as follows:

Output of this problem using BFS:

- `[(3, 3, 0), (3, 1, 1), (3, 2, 0), (3, 0, 1), (3, 1, 0), (1, 1, 1), (2, 2, 0), (0, 2, 1), (0, 3, 0), (0, 1, 1), (1, 1, 0), (0, 0, 1)]`

Depth-First Search (DFS):

DFS can also solve the problem, but it doesn't always find the shortest solution. The number of moves may vary based on how it explores different states. DFS goes deep into one path first and only backtracks when it hits a dead end, which can lead to less efficient or suboptimal solutions.

Output of this problem using DFS:

- `[(3, 3, 0), (2, 2, 1), (3, 2, 0), (3, 0, 1), (3, 1, 0), (1, 1, 1), (2, 2, 0), (0, 2, 1), (0, 3, 0), (0, 1, 1), (0, 2, 0), (0, 0, 1)]`

Complexity Type	BFS	DFS
Time Complexity	$O(b^d)$	$O(b^d)$
Space Complexity	$O(b^d)$	$O(bd)$

TABLE I: Time and Space Complexities of BFS and DFS

Complexity Analysis: b is the branching factor and d is the depth of the shallowest goal state.

F. Conclusion

The **Missionaries and Cannibals Problem** illustrates the application of state-space search methods such as **BFS** and **DFS** for solving constraint-based puzzles. While both methods are capable of finding a solution, BFS guarantees the shortest possible path by exploring all states level by level, at the cost of higher memory usage. In contrast, DFS may find a solution more quickly, but it does not ensure optimality. This comparison emphasizes the trade-offs between search strategies in terms of solution quality, time, and space efficiency when addressing problems with constraints.

II. Rabbit Leap Problem

A. Objective

Model the **Rabbit Leap Problem** as a state-space search problem and solve it using BFS and DFS. Determine a sequence of moves allowing all rabbits to safely cross to the opposite side.

B. Problem Statement

Three eastbound rabbits are in a line blocked by three westbound rabbits, with one empty stone in between. Rabbits can only move 1 or 2 steps forward, or jump over exactly one rabbit into the empty stone.

Constraints:

1) Rabbits can only jump into the empty stone.
2) Rabbits may move 1 or 2 steps forward.
3) Rabbits can jump over exactly one adjacent rabbit but not more.
4) The goal is to swap the positions of east-bound and west-bound rabbits safely.

C. Methodology

State Representation: The state is represented as a tuple of 7 positions (6 rabbits + 1 empty space):

- 'E' – east-bound rabbit
- 'W' – west-bound rabbit
- '_' – empty space

Initial State: EEE_WWW
Goal State: WWW_EEE

Total Possible States:

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

However, not all configurations are valid due to movement constraints, so the effective search space is smaller than 140 configurations.

State Transition Model:

The problem is modeled as a series of valid moves that transfer the east bound rabbits to the right and the west bound rabbits to the left. The permissible moves are:

- Move a single rabbit forward into the empty space.
- Move a rabbit two steps forward into empty space.
- Jump a rabbit over exactly one adjacent rabbit into the empty space.

Validation Function:

A validation function ensures moves follow the rules: rabbits move only into empty space, can jump over one adjacent rabbit, and do not exceed movement limits.

D. Search Strategies

Two classical search strategies are used to solve the Rabbit Leap problem:

- **Breadth-First Search (BFS):** Explores states level by level, ensuring the first solution found is optimal with the fewest moves.
- **Depth-First Search (DFS):** Explores one path as deeply as possible before backtracking. DFS can find a solution faster, but it does not guarantee optimality.

Algorithm 3 Generate Rabbit States

```
Initialize state ← 'EEE_WWW'
states ← [state]
for each state in states do
    for each valid rabbit move from state do
        new_state ← perform move on state
        if new_state ∉ states then
            Add new_state to states
        end if
    end for
end for
Return states =0
```

Algorithm 4 BFS Algorithm

```
Input: initial state, goal state
Initialize queue with the initial state
Initialize a visited set to track explored states
while queue is not empty do
    Current_state = dequeue(queue)
    if Current_state = goal_state then
        return the sequence of steps to reach the goal
    end if
    for each valid move from Current_state do
        New_state = apply_move(Current_state)
        if New_state not in visited then
            Mark New_state as visited
            Enqueue New_state
        end if
    end for
end while
return no solution =0
```

Algorithm 5 DFS Algorithm

```
Input: initial state, goal state
Initialize stack with the initial state
Initialize a visited set to track explored states
while stack is not empty do
    Current_state = pop(stack)
    if Current_state = goal_state then
        return the sequence of steps to reach the goal
    end if
    for each valid move from Current_state do
        New_state = apply_move(Current_state)
        if New_state not in visited then
            Mark New_state as visited
            Push New_state onto stack
        end if
    end for
end while
return no solution =0
```

E. Analysis

Complexity Type	BFS	DFS
Time Complexity	$O(b^d)$	$O(b^d)$
Space Complexity	$O(b^d)$	$O(bd)$
Nodes Explored	35	72

TABLE II: Time and Space Complexities of BFS and DFS

Complexity Analysis: b is the branching factor and d is the depth of the shallowest goal state.

F. Conclusion

The **Rabbit Leap Problem** demonstrates the use of state-space search techniques such as **BFS** and **DFS** for solving puzzles with movement constraints. While both algorithms can find a solution, **BFS** guarantees the shortest possible sequence of moves by exploring all states level by level, though it requires more memory. Specifically, BFS has a time complexity of $O(b^d)$ and a space complexity of $O(b^d)$, where b is the branching factor and d is the depth of the shallowest goal state. In contrast, **DFS** may reach a solution more quickly, but it does not ensure optimality, and it has a time complexity of $O(b^d)$ with a more memory-efficient space complexity of $O(bd)$. This comparison highlights the trade-offs between search strategies in terms of solution optimality, time, and space efficiency when addressing constraint-based problems.

ASSIGNMENT 2

III. PUZZLE-8

A. Objective

The objective is to design a **graph search agent** for the **Puzzle-8 problem** using **hash tables** to track visited states and **queues** for managing the frontier. The experiment also incorporates **Iterative Deepening Search (IDS)**, **backtracking** for solution path retrieval, and a **performance analysis** of **time and memory** requirements across varying depths.

B. Problem Statement

The **Puzzle-8** consists of arranging tiles 1–8 on a 3×3 grid with one blank space to reach a **goal state** from a given **start state**. The task includes:

- Develop a **graph search agent** with pseudocode and flowchart.
- Implement environment functions for **Puzzle-8 moves and states**.
- Explain and apply **Iterative Deepening Search (IDS)** for systematic search.
- Use a **backtracking function** to reconstruct the path to the goal.
- Generate puzzle instances at the depth d and record their **performance metrics** in terms of **memory and time**.

C. Methodology

State Representation:

- Each puzzle configuration is represented as a 3×3 grid or a tuple of 9 elements, where each element corresponds to a tile number, and one element represents the blank space.
- The blank space (typically represented by 0 or an underscore) allows movement of adjacent tiles into its position.

Environment Functions:

- Implement functions to generate possible moves, apply actions, and check **the validity of the goal state**.
- Verify the legality of the move based on the position of the blank tile.

Graph Search Agent:

- **Frontier Management:** The frontier contains states that are yet to be explored, ensuring systematic expansion.
- **Visited State Table:** A hash table (or set) stores previously visited states to prevent cycles and redundant exploration.
- **State Expansion:** For each state removed from the frontier, all valid successor states are then generated and are added to the frontier if they have not been visited.

Iterative Deepening Search (IDS):

- Combine DFS depth-limiting with repeated deepening to ensure optimal solution depth.
- Explore increasing depth limits until the goal is reached.
- Ensures optimality in terms of the number of moves required to reach the goal.
- Particularly useful for large state spaces, as it avoids the high memory cost of BFS while still guaranteeing that shallow solutions are found first.

Backtracking:

- Store **parent pointers** for each state during search.
- Once the goal state is found, trace the steps from the goal to begin reconstructing the solution path.

Performance Analysis:

- Generate Puzzle-8 instances at varying goal depths d.
- Measure **time and memory usage** for each instance.
- Record results in a table for comparison.

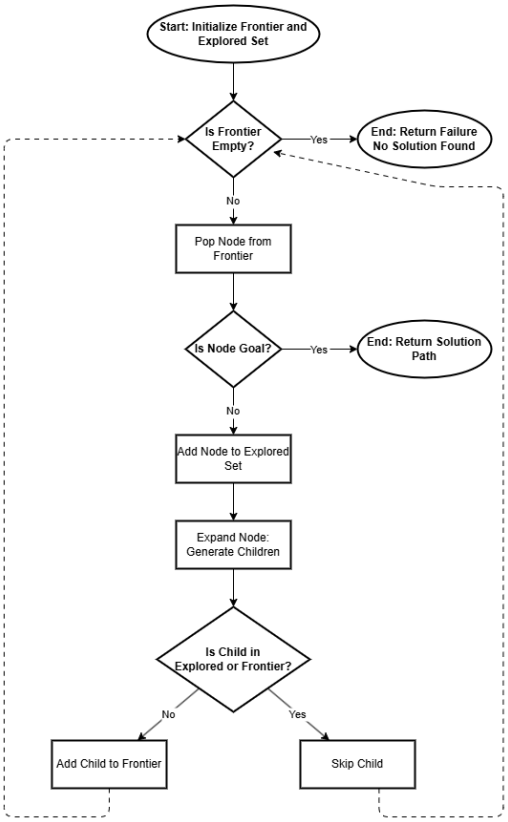

Fig. 1: Flow Chart of the Graph Search Agent.

D. Search Strategies

The **Puzzle-8 problem** is solved using a **graph search agent** with a **frontier** (queue for BFS or stack for DFS) and a **hash table** to track visited states. **Successor states** are generated by moving the blank space in valid directions. **Iterative Deepening Search (IDS)** ensures systematic exploration to the optimal depth. Once the goal is reached, **backtracking** reconstructs the path of the solution from the start to the end. These strategies allow efficient navigation of the large state space while preventing redundant exploration. The choice of search method impacts memory usage, execution time, and the ability to find the optimal solution efficiently.

Algorithm 6 Graph Search Agent (Uniform-Cost / BFS)

```
Input: initial state, goal state
Output: path from start to goal (if exists)
Initialize queue with initial state
Initialize visited set
Initialize parent mapping: came_from[start] ← None
while queue is not empty do
    node ← pop from queue
    if node = goal state then
        Backtrack using came_from to reconstruct path
        Return: path
    end if
    for each successor of node do
        if successor not in visited and not in queue then
            Add successor to queue
            came_from[successor] ← node
        end if
    end for
end while
Return: failure =0
```

Algorithm 7 Iterative Deepening Search (IDS)

Input: start state, goal state
Output: path from start to goal (if exists)
Initialize depth limit $d \leftarrow 0$
repeat
  Perform Depth-Limited Search with limit d
  if goal found **then**
    Return: path
  end if
  $d \leftarrow d + 1$
until goal found
Return: failure =0

Algorithm 8 Backtracking Path Retrieval

Input: parent mapping *came_from*, goal state
Output: path from start to goal
Initialize empty list *path*
$current \leftarrow$ goal
while $current \neq$ None **do**
  Append $current$ to *path*
  $current \leftarrow$ *came_from[current]*
end while
Return: reversed(*path*) =0

Algorithm 9 Puzzle Generation at Depth d

Input: goal state, depth d
Output: puzzle state at depth d
$state \leftarrow$ GOAL_STATE
for $i = 1$ to d **do**
  Generate successors of $state$
  Randomly select one successor as new $state$
end for
Return: $state$ =0

E. Analysis

Depth	Memory (MB)	Time (s)	Moves
6	0.00	0.0001	0
8	0.01	0.0002	2
10	0.00	0.0000	0
13	0.24	0.0065	9
15	0.03	0.0015	7
17	0.03	0.0015	7
20	0.01	0.0004	4
25	0.08	0.0043	9
40	0.01	0.0004	4

TABLE III: Performance Analysis of Puzzle-8 Search.

F. Conclusion

The implementation of graph search agents for the Puzzle-8 problem shows how **queues** and **hash tables** enable systematic state-space exploration. Using **breadth-first search (BFS)** or **uniform-cost search (UCS)** with *parent mapping*, the agent efficiently reconstructs the solution path. **Iterative Deepening Search (IDS)** demonstrates the trade-off between memory and time efficiency, while puzzle generation at varying depths confirms the impact of increasing complexity. Overall, the study highlights the **strengths** and **limitations** of classical search strategies in solving **Puzzle-8**.

IV. Plagarism Detector

A. Objective

The task is to design a **plagiarism detection system** that aligns sentences from two documents using the **A* search algorithm**. The system should minimize edit distance to identify overlapping or highly similar text. This enables efficient detection of potential plagiarism through **optimal text alignment**.

B. Problem Statement

The problem is to **detect plagiarism** between two given documents by aligning their sentences or paragraphs using the **A* search algorithm**. Each alignment step must *minimize the edit distance*, ensuring that similar or identical text segments are effectively identified. The system should represent states as partial alignments, with **transitions** covering sentence matches or skips. The goal is to achieve a complete alignment of both documents while keeping the total cost minimal. By leveraging A*'s combination of **path cost** and **heuristic**, the approach ensures efficient exploration of the alignment space. The outcome highlights pairs of sentences with low edit distance as potential cases of plagiarism.

C. Methodology

Text Preprocessing:

- **Normalization:** lowercase text, remove punctuation and control characters, replace multiple spaces with one.
- **Sentence Tokenization:** split text on '.', '!', '?' followed by whitespace; discard empty sentences.

State Representation:

Each state (i, j) represents the current sentence in Document A and Document B.
Total state space $= n \times m$, where n and m are the number of sentences in the documents.

Levenshtein Distance (Edit Distance):
Measures dissimilarity between sentences a and b:

$$d(a,b) = \min \begin{cases} d(a_{i-1}, b_j) + 1 & \text{(deletion)} \\ d(a_i, b_{j-1}) + 1 & \text{(insertion)} \\ d(a_{i-1}, b_{j-1}) + \text{cost} & \text{(substitution)} \end{cases}$$

$$\text{cost} = \begin{cases} 0 & \text{if characters match} \\ 1 & \text{otherwise} \end{cases}$$

Cost Function (Alignment Cost):
Total alignment cost:

$$C(S) = \sum \text{LevenshteinCost}(a_i, b_j) + \text{SkipPenalty} \times (\#\text{skips})$$

Goal: minimize $C(S)$ for optimal alignment.

Transition Function:

- **Match:** align current sentences a_i and b_j.
- **Skip A / Skip B:** skip sentence in one document, applying a skip penalty.

A Search Algorithm:*

- **Node cost:** $g(n) =$ accumulated alignment cost.
- **Heuristic:** $h(n) =$ estimated minimum remaining cost.
- **Total cost:** $f(n) = g(n) + h(n)$.

Use a priority queue (frontier) to select nodes with lowest $f(n)$.
Generate successors using the transition function.
Backtracking reconstructs the alignment path from goal (n, m).

Plagiarism Scoring:

Percentage similarity:

$$\text{Score}(\%) = \frac{\text{Exact + Strong + Moderate Matches}}{|A|} \times 100$$

Classification:

- **Low:** $0 < \text{score} \leq 40\%$
- **Moderate:** $40 < \text{score} \leq 70\%$
- **High:** $70 < \text{score} \leq 100\%$

D. Test Cases

The following test cases were used to evaluate the plagiarism detection system:

Test Case 1: The input consists of two identical documents. The expected output is that all sentences will align perfectly, with a zero edit distance between the two documents. This means there are no differences in the content, indicating that the documents are exact duplicates.

Test Case 2: One document has minor changes such as synonym replacements or slight alterations in word order. The expected output is that most sentences will still align, but with a low edit distance. This suggests minor modifications while retaining the original meaning, which could indicate potential paraphrasing or rephrasing.

Test Case 3: The input involves two completely different documents. The expected output is a high edit distance for most sentence alignments. This suggests that the documents do not share significant similarities, making it clear that there is no plagiarism or content duplication.

Test Case 4: Features two documents with some overlapping content. The expected output is that the overlapping sections will align with a low edit distance, while the rest of the content may have a higher edit distance. This indicates that portions of the documents are similar and might suggest partial plagiarism or reused content.

E. Solving Strategy

The documents are preprocessed by **normalizing text** and **tokenizing into sentences**. Each sentence pair is represented as a **state** in a grid, and the **Levenshtein distance** is used to compute **alignment costs**. The **A* search algorithm** explores this state space to find the **minimum-cost alignment**, using a **priority queue** to select nodes with the lowest combined path and heuristic cost. **Backtracking** reconstructs the optimal alignment, and the resulting sentence matches are used to calculate a **plagiarism score**.

Algorithm 10 A*-Based Plagiarism Detection

```
Input: fileA, fileB – two text documents
Output: path – optimal alignment of sentences
Preprocess documents:
    sentcA ← Normalize(Tokenize(ReadF(fileA)))
    sentcB ← Normalize(Tokenize(ReadF(fileB)))
Initialize:
    start ← (0, 0), goal ← (len(sentcA), len(sentcB))
    frontier.push((0, start)), gscore[start] ← 0, came_from ← {}
while frontier not empty do
    (f, (i, j)) ← frontier.pop_lowest()
    if (i, j) = goal then
        Return Reconstruct_Path(came_from, goal)
    end if
    Match: align sentcA[i] and sentcB[j] if possible
    Skip A: move forward in document A
    Skip B: move forward in document B
    Update frontier, gscore, and came_from for each new state
end while
Return: failure if no alignment found =0
```

F. Analysis

After running the code on the test cases, the system successfully detected potential plagiarism based on sentence alignments and edit distance:

*1) **Identical Documents**:* For identical documents, all sentences aligned with a similarity score of 1 (100% plagiarism).
Matched sentence pairs: 5
Exact matches: 5
Strong similarities: 0
Moderate similarities: 0
Exact match ratio: 100.00%
Strong similarity ratio: 0.00%
Moderate similarity ratio: 0.00%
Plagiarism coverage: 100.0% → High

*2) **Slightly Modified Documents**:* For slightly modified documents, most sentences had high similarity scores of around 0.83 (83% plagiarism).
Matched sentence pairs: 5
Exact matches: 3
Strong similarities: 2
Moderate similarities: 0
Exact match ratio: 60.00%
Strong similarity ratio: 40.00%
Moderate similarity ratio: 0.00%
Plagiarism coverage: 83.3% → High

*3) **Completely Different Documents**:* For completely different documents, the similarity scores were generally low around 0 (no plagiarism).
Matched sentence pairs: 5
Exact matches: 0
Strong similarities: 0
Moderate similarities: 0
Exact match ratio: 0.00%
Strong similarity ratio: 0.00%
Moderate similarity ratio: 0.00%
Plagiarism coverage: 0.0% → Low

*4) **Partial Overlap**:* For partially overlapping documents, the overlapping content was correctly flagged with a similarity score of around 0.4.
Matched sentence pairs: 5
Exact matches: 0
Strong similarities: 0
Moderate similarities: 2
Exact match ratio: 0.00%
Strong similarity ratio: 0.00%
Moderate similarity ratio: 40.00%
Plagiarism coverage: 40.0% → Medium

G. Conclusion

The **plagiarism detection system** successfully aligns sentences from two documents using **A* search**, minimizing the cumulative **alignment cost** based on **Levenshtein distance** and **skip penalties**. The approach efficiently identifies **exact**, **strong**, and **moderate** similarities, providing a quantitative measure of potential plagiarism. By combining **text preprocessing**, **sentence tokenization**, and **cost-based alignment**, the system delivers **accurate** and **interpretable** results, with the **A* search** ensuring an **optimal mapping** between sentences.

ASSIGNMENT 3

V. MARBLE SOLITAIRE

A. Objective

Marble Solitaire is a strategic puzzle game played on a cross-shaped board. The game begins with marbles occupying all but the center spot, and the goal is to eliminate marbles by jumping over adjacent ones until only one marble remains, ideally in the center. Solving this puzzle efficiently requires search algorithms that explore the board's state space to find an optimal solution.

B. Problem Statement

The problem involves solving the **Marble Solitaire** puzzle by reducing the initial board configuration to a **single marble at the center** of the board. The tasks include:

1) Implement a **priority queue-based search** considering path cost.
2) Describe **two heuristic functions** to guide the search.
3) Apply the **Best-First Search** algorithm.
4) Apply the **A* Search** algorithm.
5) Compare the performance of **Best-First Search** and **A* Search** in terms of **efficiency** and **solution quality**.

C. Methodology

State Representation:

Represent the board as a 2D grid, where each cell indicates the presence or absence of a marble.
The board is a 7×7 grid:

- 1 = marble
- 0 = empty spot
- -1 = invalid/non-playable space

Valid Moves:

A move consists of jumping a marble over an adjacent marble into an empty spot, removing the jumped marble.

Cost Function:

Each move has a uniform cost; total path cost is the number of moves.

Heuristic Functions:

Remaining Marbles Heuristic: Counts the number of marbles left on the board; fewer marbles indicate being closer to the goal.

$$h_1(n) = M$$

where M = number of marbles remaining.
Distance-to-Center Heuristic: Measures the sum of distances of all marbles from the center position; lower total distance suggests that the state is closer to the goal.

$$h_2(n) = \sum_{i=1}^{M} (|x_i - x_c| + |y_i - y_c|)$$

where (x_i, y_i) = coordinates of marble i,
$(x_c, y_c) = (3, 3)$ center of the board,
M = number of marbles.

Solution Tracking::

Backtrack from the goal state to reconstruct the sequence of moves leading to a single marble at the center.

D. Search Strategies

The **Marble Solitaire** problem is solved using **Best-First Search** and **A* Search** guided by the defined heuristics. States are expanded using valid moves, with a **priority queue** selecting the most promising states. Backtracking reconstructs the optimal sequence of moves leading to a single marble at the center of the board.

Algorithm 11 Best-First Search for Marble Solitaire

Input: initial state, goal state
Output: path from start to goal (if exists)
Initialize priority queue (frontier) with initial state, prioritized by $h(n)$
Initialize explored set
Initialize parent mapping: *came_from*[start] ← None
while frontier is not empty **do**
  node ← pop from frontier (with lowest $h(n)$)
  if node = goal state **then**
    Backtrack using *came_from* to reconstruct path
    Return: path
  end if
  for each successor of node **do**
    if successor not in explored and not in frontier **then**
      frontier ← frontier ∪ {successor} with priority $h(successor)$
      came_from[successor] ← node
    end if
  end for
  explored ← explored ∪ {node}
end while
Return: failure =0

Algorithm 12 Dijkstra (H1) for Marble Solitaire

Input: initial state, goal state
Output: path from start to goal (if exists)
Initialize priority queue (frontier) with initial state, prioritized by $g(n)$
Initialize explored set
Initialize parent mapping: *came_from*[start] ← None
while frontier is not empty **do**
  node ← pop from frontier (with lowest path cost $g(n)$)
  if node = goal state **then**
    Backtrack using *came_from* to reconstruct path
    Return: path
  end if
  explored ← explored ∪ {node}
  for each successor of node **do**
    if successor not in explored and not in frontier **then**
      frontier ← frontier ∪ {successor} with priority $g(n) + 1$
      came_from[successor] ← node
    end if
  end for
end while
Return: failure =0

Algorithm 13 A* Search (H2) for Marble Solitaire

Input: initial state, goal state
Output: path from start to goal (if exists)
Initialize priority queue (frontier) with initial state, prioritized by $f(n) = g(n) + h(n)$
Initialize explored set
Initialize parent mapping: *came_from*[start] ← None
while frontier is not empty **do**
  node ← pop from frontier (with lowest $f(n)$)
  if node = goal state **then**
    Backtrack using *came_from* to reconstruct path
    Return: path
  end if
  explored ← explored ∪ {node}
  for each successor of node **do**
    if successor not in explored and not in frontier **then**
      frontier ← frontier ∪ {successor} with priority $g(n) + h(n)$
      came_from[successor] ← node
    end if
  end for
end while
Return: failure =0

E. Analysis

Algorithm	Nodes Expanded	Time (s)	Solution Depth
Dijkstra	>10,000,000	> hours	31*
Best-First Search	524,012	11.7076	31
A* Search	523,255	11.134	31

TABLE IV: Performance Comparison of UCS, Best-First, and A* Search

*Estimated; Dijkstra could not finish in reasonable time.

F. Conclusion

The **Marble Solitaire** problem illustrates how **search algorithms** navigate a combinatorial state space. **Uniform-Cost Search** guarantees an optimal solution by expanding nodes based on path cost. **A* Search** uses heuristics to reach the goal more efficiently while maintaining optimality. **Greedy Best-First Search** prioritizes speed using the heuristic but may not find the shortest solution. Comparing these methods shows trade-offs between **solution quality**, **computation time**, and **memory usage**, guiding the choice of algorithm for complex puzzles.

VI. K-SAT Problem

A. Objective

The goal is to **generate random k-SAT instances** with a specified number of **variables**, **clauses**, and **clause size**. Each instance should contain m **clauses** of length k, with **literals** randomly chosen from n **variables**, ensuring that each clause contains distinct variables or their negations. These instances are intended for **testing SAT solvers** and evaluating **algorithm performance**.

B. Problem Statement

We are required to generate **uniform random k-SAT problem instances** based on the following parameters:

k: Number of literals per clause
n: Number of variables
m: Number of clauses

Each clause must consist of **k distinct variables** or their negations. The total number of possible clauses is given by:

$$\binom{n}{k} \cdot 2^k$$

State Space: All possible combinations of m clauses of length k drawn from n variables, with each literal possibly negated.

Goal: Generate valid k-SAT formulas that satisfy the input parameters. There is no specific solution, as the objective is to produce problem instances.

Constraints:

1) $k \leq n$
2) Each clause contains unique variables
3) Each literal is randomly assigned as positive or negated

C. Methodology

State Representation:

Let the set of variables be $V = \{x_1, x_2, \ldots, x_n\}$.
Each clause C of length k is represented as a list of integers:

$$C = [l_1, l_2, \ldots, l_k]$$

where $l_i = v$ for a positive literal or $l_i = -v$ for a negated literal.
A formula F is a collection of m clauses:

$$F = [C_1, C_2, \ldots, C_m]$$

Example (3-SAT, 4 variables, 5 clauses):

$$F = [[1, -3, 4], [-2, 3, -4], [1, -2, 3], [-1, 3, -4], [1, 2, -3]]$$

Clause Generation Procedure:

1) Randomly select k distinct variables from the set $\{1, 2, \ldots, n\}$ for a clause.
2) Assign a random sign (positive or negative) to each variable.
3) Repeat the above steps until m clauses are generated.

Mathematical Formulation:

Let $F = \{C_1, C_2, \ldots, C_m\}$ where each clause $C_i = \{l_1, \ldots, l_k\}$.
Each literal $l_j \in \{-n, \ldots, -1, 1, \ldots, n\}$.
Clause constraints:

$$|C_i| = k \quad \text{and} \quad \forall i, j \in C_i, |i| \neq |j|$$

Formula constraints: F consists of m clauses satisfying the above clause constraints.

D. Search Strategies

The **k-SAT instance generation** can be approached by randomly selecting k distinct variables for each clause and assigning each a positive or negative sign, ensuring **uniqueness within clauses**. This process is repeated m times to form the complete formula. The strategy ensures **uniform randomness**, adherence to **clause constraints**, and produces valid **SAT instances**.

Algorithm 14 Random k-SAT Generator

Input: k (literals per clause), n (variables), m (number of clauses)
Output: F = k-SAT formula with m clauses of length k
Initialize: empty list F, set random seed (optional)
for $i = 1$ **to** m **do**
    Randomly select k distinct variables from $\{1, \ldots, n\}$
    Assign each variable randomly as positive ($+v$) or negated ($-v$)
    Form clause $C = [l_1, l_2, \ldots, l_k]$
    Append C to formula F
end for
Return: F =0

E. Analysis

The program generates valid random k-SAT formulas as a list of clauses. Each clause contains exactly k literals with no repeated variables. Literals are randomly negated or positive. Example output for $k = 3$, $n = 4$, $m = 5$:

$$[1, -3, 4]\ [-2, 3, -4]\ [1, -2, 3]\ [-1, 3, -4]\ [1, 2, -3]$$

With a fixed seed, the same formula is reproducible. These formulas can be used to benchmark SAT solvers or test algorithms.

F. Conclusion

The implemented program successfully generates **uniform random k-SAT instances** based on the specified parameters. Each clause contains exactly k **distinct literals**, randomly assigned as **positive or negated**, ensuring both validity and diversity in the generated formulas. Using a fixed **random seed** allows reproducibility, making these instances suitable for **testing and benchmarking SAT solvers**. Overall, the methodology provides a **flexible and reliable framework** to create problem instances of varying sizes and complexities for experimental evaluation.

VII. 3-SAT Problem

A. Objective

The goal is to design and evaluate **Hill-Climbing**, **Beam Search**, and **Variable-Neighborhood-Descent (VND)** algorithms on **randomly generated 3-SAT instances**. The performance of these algorithms will be analyzed by varying the number of **variables** (n) and **clauses** (m), with a focus on **execution time**, **solution quality**, and the effectiveness of **heuristic functions** in guiding the search process.

B. Problem Statement

The problem involves solving **uniform random 3-SAT instances** with the following parameters:

- n: Number of Boolean variables
- m: Number of clauses

Each clause contains exactly three literals, where a literal may be a variable or its negation. The objective is to find a truth assignment that **maximizes the number of satisfied clauses**, ideally satisfying all.
The state space is defined as:

$$S = \{0, 1\}^n$$

representing all possible assignments of variables. The goal is to identify

$$s^* = \arg\max_{s \in S} f(s)$$

where $f(s)$ is the number of clauses satisfied by assignment s.
The algorithms to be implemented and evaluated are:

1) **Hill-Climbing**
2) **Beam Search** (beam widths = 3 and 4)
3) **Variable-Neighborhood-Descent (VND)** with three neighborhood functions

C. Methodology

State Representation:

Each state corresponds to a complete truth assignment of all Boolean variables $\{x_1, x_2, \ldots, x_n\}$.
The search space is $S = \{0, 1\}^n$, representing all possible assignments.

Evaluation Function:

For a given state s, the objective function is defined as:

$$f(s) = \sum_{i=1}^{m} C_i(s)$$

where $C_i(s) = 1$ if clause i is satisfied by assignment s, and 0 otherwise.
This function measures the quality of a candidate solution.

Heuristic Functions:

- **Basic Heuristic:** Selects moves that maximize $f(s)$ directly, i.e., the total number of satisfied clauses.
- **Make-Break Heuristic:** Evaluates the net effect of flipping a variable by considering the number of clauses satisfied (make) minus the number unsatisfied (break).

D. Search Strategies

Hill-Climbing: This iterative optimization technique evaluates neighboring states to find a better solution based on a heuristic function. It may get trapped in local optima.

Beam Search: This memory-efficient heuristic algorithm maintains a limited number of the best candidates at each search level, balancing between breadth and depth in exploration

Variable-Neighborhood-Descent (VND): VND systematically explores different neighborhoods by combining local search strategies to escape local optima. It alternates between various neighborhood configurations.

Algorithm 15 Hill-Climbing (HC) for 3-SAT

Input: clauses, number of variables n, max iterations *max_iters*, number of restarts
Output: best assignment score
best_score $\leftarrow -\infty$
for each restart **do**
  assignment $\leftarrow$ random assignment
  for iteration = 1 TO max_iters **do**
    current_score $\leftarrow$ Evaluate(assignment)
    improved $\leftarrow$ False
    for each variable v **do**
      Flip v
      new_score $\leftarrow$ Evaluate(assignment)
      if new_score $>$ current_score **then**
        improved $\leftarrow$ True
        break
      else
        Flip v back
      end if
    end for
    if not improved **then**
      break
    end if
    if current_score $>$ best_score **then**
      best_score $\leftarrow$ current_score
    end if
  end for
end for
Return: best_score =0

Algorithm 16 Beam Search (BS) for 3-SAT

Input: clauses, number of variables n, beam width *beam_width*, max iterations *max_iters*
Output: best assignment score
beam $\leftarrow$ random assignments (size *beam_width*)
best_score $\leftarrow -\infty$
for iteration = 1 TO max_iters **do**
  candidates $\leftarrow$ []
  for assignment IN beam **do**
    current_score $\leftarrow$ Evaluate(assignment)
    if current_score $>$ best_score **then**
      best_score $\leftarrow$ current_score
    end if
    for each variable v **do**
      neighbor $\leftarrow$ assignment with v flipped
      candidates.append((Evaluate(neighbor), neighbor))
    end for
  end for
  Sort candidates by score descending
  beam $\leftarrow$ top *beam_width* assignments
end for
Return: best_score =0

Algorithm 17 Variable-Neighborhood-Descent (VND) for 3-SAT

Input: clauses, n, max iterations *max_iters*
Output: best assignment score
assignment $\leftarrow$ random assignment
best_score $\leftarrow$ Evaluate(assignment)
for iteration = 1 TO max_iters **do**
  improved $\leftarrow$ False
  for each variable v **do**
    Flip v, new_score $\leftarrow$ Evaluate(assignment)
    if new_score $>$ best_score **then**
      best_score $\leftarrow$ new_score, improved $\leftarrow$ True, **break**
    end if
    Flip v back
  end for
  if improved **then**
    continue
  end if
  for each pair (v_1, v_2) **do**
    Flip v_1, v_2, new_score $\leftarrow$ Evaluate(assignment)
    if new_score $>$ best_score **then**
      best_score $\leftarrow$ new_score, improved $\leftarrow$ True, **break**
    end if
    Flip v_1, v_2 back
  end for
  if improved **then**
    continue
  end if
  for k = 1 TO 10 **do**
    Randomly select v_1, v_2, v_3, flip, new_score $\leftarrow$ Evaluate(assignment)
    if new_score $>$ best_score **then**
      best_score $\leftarrow$ new_score, improved $\leftarrow$ True, **break**
    end if
    Flip v_1, v_2, v_3 back
  end for
  if not improved **then**
    break
  end if
end for
Return: best_score =0

E. Analysis

The implemented search algorithms exhibit distinct strengths and trade-offs in solving uniform random 3-SAT instances.

Hill-Climbing is fast and often finds a feasible solution quickly but can become trapped in local optima, limiting solution quality.

Beam Search balances exploration and memory efficiency, with larger beam widths producing higher-quality assignments by considering multiple candidate states simultaneously.

Variable-Neighborhood-Descent (VND) systematically explores multiple neighborhoods, effectively escaping local optima and achieving improved clause satisfaction. Overall, the results demonstrate that heuristic-guided search and neighborhood strategies significantly impact performance, with VND generally providing the best balance between solution quality and robustness.

F. Conclusion

The study of uniform random 3-SAT instances using heuristic-based search algorithms demonstrates clear differences in performance and robustness. **Hill-Climbing** is fast and often finds feasible solutions quickly but may get trapped in local optima. **Beam Search** efficiently balances exploration and memory usage, producing higher-quality solutions with larger beam widths. **Variable-Neighborhood-Descent (VND)** effectively escapes local optima by systematically exploring multiple neighborhoods, achieving better clause satisfaction overall.

ASSIGNMENT 4

VIII. TOUR OF RAJASTHAN

A. Objective

The objective is to design a **Simulated Annealing-based algorithm** to plan a **cost-effective tour of Rajasthan**, visiting at least twenty key tourist locations while minimizing total travel distance and cost.

B. Problem Statement

Given a set of major tourist destinations in Rajasthan, the task is to determine an **efficient tour route** that visits all locations exactly once and returns to the starting point, minimizing the overall travel cost. The cost between two locations is assumed proportional to the distance. Using **Simulated Annealing**, the algorithm probabilistically explores possible routes, gradually refining the solution to find a **near-optimal itinerary** that balances travel efficiency, cost, and coverage of important sites.

C. Methodology

State Representation:

Each state represents a possible **tour sequence**, modeled as a permutation of 20 tourist cities in Rajasthan.

Cost Function:

The cost is the **total travel distance** of the tour, including the return to the starting city.
Distances are calculated using the **Haversine formula** for great-circle distance between two locations with latitude/longitude coordinates:

$$d = 2R \arcsin \sqrt{\sin^2(\Delta\varphi/2) + \cos\varphi_1 \cos\varphi_2 \, \sin^2(\Delta\lambda/2)}$$

where R is the Earth's radius, $\Delta\varphi$ is the latitude difference, and $\Delta\lambda$ is the longitude difference.

Simulated Annealing Algorithm:

Start with a random tour and high initial temperature T_0.
For each iteration:

- Generate a neighbor tour.
- Compute cost difference:

$$\Delta C = C_{\text{neighbor}} - C_{\text{current}}$$

- If $\Delta C \leq 0$, accept the neighbor.
- If $\Delta C > 0$, accept with probability:

$$P = e^{-\Delta C/T}$$

- Update the best tour if an improvement is found.
- Gradually reduce temperature using:

$$T_{k+1} = \alpha \cdot T_k, \quad 0 < \alpha < 1$$

Convergence Tracking:

Track the **tour cost** and **acceptance probability** over iterations to monitor optimization progress and solution stability.

Each state represents a possible **tour sequence**, modeled as a permutation of 20 tourist cities in Rajasthan.

D. Solving Strategy

The problem is solved using the **Simulated Annealing** approach:

1) **Initialization:** Start with a random tour of 20 cities.
2) **Cost Evaluation:** Calculate total travel distance.
3) **Neighbor Generation:** Apply a 2-opt move to create a new tour.
4) **Acceptance:** Accept better tours, or worse ones with probability depending on T.
5) **Cooling:** Reduce temperature T gradually using rate α.
6) **Termination:** Stop after max iterations or when T is very small.

Algorithm 18 Simulated Annealing for Rajasthan Tourist Tour

```
Input: Set of tourist locations with coordinates
Output: Best tour sequence, total distance
Initialize set of locations and compute pairwise distance matrix
Generate a random initial tour and compute its cost
Set current_tour ← initial_tour, best_tour ← current_tour
Initialize temperature T ← T_0, cooling rate α, max iterations
while iteration < max_iterations do
    Generate neighbor tour by reversing a random segment (2-opt move)
    Compute neighbor cost and calculate ΔC
    if ΔC ≤ 0 then
        Accept neighbor as current tour
    else
        Accept neighbor with probability p
    end if
    Update best_tour if C_current < C_best
    Reduce temperature: T ← α · T
    Record current cost for convergence tracking
end while
Return: best tour sequence, total distance, and convergence plot =0
```

E. Analysis

The **Simulated Annealing** algorithm produces an optimized itinerary covering all 20 tourist locations in Rajasthan. The results are as follows:

Tour Sequence: Bikaner → Jaisalmer → Jodhpur → Pali → Mount Abu → Udaipur → Dungarpur → Banswara → Chittorgarh → Kota → Bundi → Ranthambore → Sawai Madhopur → Ajmer → Pushkar → Jaipur → Bharatpur → Alwar → Shekhawati → Nagaur → Bikaner (return to start).
Total Distance: 2400.89 km
Estimated Travel Time: 40.0 hours (approx. 5 days at 8 hours/day)
Convergence Behavior: The cost decreases over iterations, showing the algorithm's ability to approach a near-optimal solution.

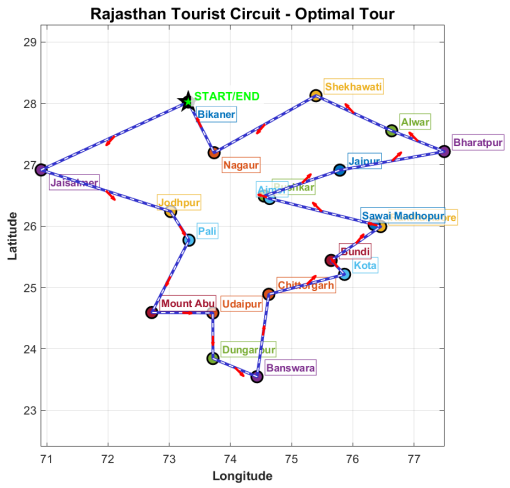

Fig. 2: Tour Route.

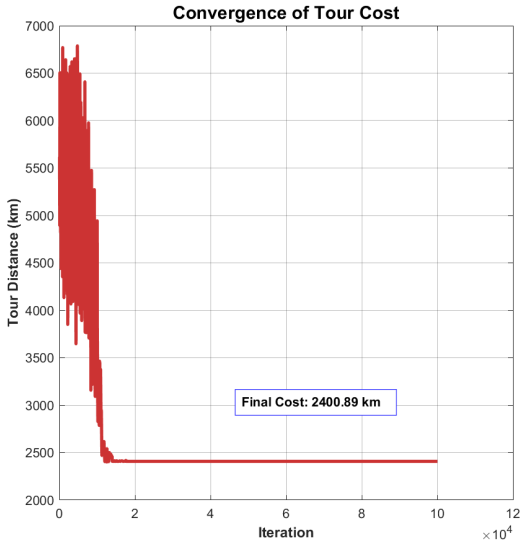

Fig. 3: Convergence of Tour Cost.

F. Conclusion

The **Simulated Annealing** approach effectively generates a **near-optimal tour** covering major tourist locations in Rajasthan. By **probabilistically accepting worse solutions** at higher temperatures and gradually cooling, the algorithm balances exploration and **convergence**. The final itinerary is practical, cost-effective, and supported by visualization and analysis. This methodology is adaptable and can be extended to other regions or **larger-scale TSP problems**.

ASSIGNMENT 4

IX. JIGSAW PUZZLE

A. Objective

Jigsaw Puzzle require arranging a set of scrambled pieces to reconstruct the original image. The objective of this task is to design an intelligent agent capable of solving a scrambled jigsaw puzzle using the **Simulated Annealing (SA)** optimization technique.

B. Problem Statement

State Space: The state space consists of all possible configurations of the 16 tiles. For a 4×4 puzzle, the total number of configurations is

$$16! \approx 2.1 \times 10^{13}.$$

Each state represents a particular arrangement of the tiles on the grid.

Initial State: The initial state is the scrambled version of the image, where tiles are randomly arranged.

Goal State: The goal state is an arrangement in which the edges of neighboring tiles align correctly, thereby reconstructing the original image. Since the correct arrangement is initially unknown, the task is formulated as an **optimization problem** where the objective is to minimize an **edge-mismatch cost function**.

Transition Model: A transition involves swapping two tiles on the grid, generating a new state. The algorithm iteratively explores the state space by performing such swaps.

Cost Function (Objective Function): The objective function $f(s)$ measures the total mismatch between the edges of adjacent tiles in a given arrangement s. The task is to find the arrangement s^* that minimizes this cost:

$$s^* = \arg\min_{s \in S} f(s)$$

where S is the set of all possible tile arrangements, and $f(s)$ quantifies the pixel-level discontinuities between neighboring tiles.

C. Methodology

State Representation:

The state of the problem is represented as a permutation of tiles:

$$S = [s_1, s_2, \ldots, s_{16}], \quad s_i \in T$$

where s_i denotes the tile at position i in the 4×4 grid.
Initial State: A random arrangement of the 16 tiles.
Goal State: An arrangement in which the edges of neighboring tiles align correctly, reconstructing the original image.
Total Possible States: $16! \approx 2.1 \times 10^{13}$

State Transition Model:

A new state is generated by swapping two tiles:

$$S' = \text{swap}(S, p, q)$$

where p and q are randomly selected positions.
This allows exploration of neighboring states while maintaining valid tile arrangements.

Cost Function (Energy Function):

The energy function quantifies the mismatch between edges of adjacent tiles:

$$C(S) = \sum_{(i,j) \in N} E(s_i, s_j)$$

where N is the set of horizontally and vertically adjacent tiles, and $E(s_i, s_j)$ is the pixel intensity difference along the shared edge.
The goal is to **minimize** $C(S)$, achieving correct tile alignment.

Simulated Annealing Strategy:

Initialization: Start with initial state S_0 and high temperature T_0.

Iteration:

- Generate a neighbor S' by swapping two tiles.
- Compute energy change: $\Delta C = C(S') - C(S_k)$
- **Acceptance Rule:**
  - If $\Delta C \leq 0$, accept S' (improvement).
  - If $\Delta C > 0$, accept S' with probability $P = e^{-\Delta C/T_k}$.

Cooling Schedule:

$$T_{k+1} = \alpha \cdot T_k, \quad 0 < \alpha < 1$$

Termination: Stop when T_k reaches minimum $T_{\min}$ or no improvement occurs over iterations.

Final State:

The resulting tile arrangement represents the reconstructed image, minimizing edge discontinuities and closely resembling the original image.

D. Search Strategies

The puzzle is solved using **Simulated Annealing (SA)**. Starting with a random arrangement, tiles are swapped to generate new states. A **cost function** measures edge mismatches, and swaps are accepted based on improvement or a probability controlled by **temperature**, which decreases over time. This balance of **exploration** and **refinement** guides the search toward the correct image reconstruction.

Algorithm 19 Jigsaw Puzzle Solving using Simulated Annealing

Input: tiles, number of rows, number of columns
Output: best_arrangement
total_tiles ← rows × cols
current_arrangement ← random ordering of tiles
best_arrangement ← current_arrangement
current_cost ← ComputeCost(current_arrangement, tiles)
best_cost ← current_cost
temperature ← initial value
while stopping criteria not met (e.g., max iterations or $T_{\min}$ reached) **do**
    new_arrangement ← swap two tiles in current_arrangement
    new_cost ← ComputeCost(new_arrangement, tiles)
    improvement ← current_cost − new_cost
    if improvement > 0 **then**
        current_arrangement ← new_arrangement
        current_cost ← new_cost
        if new_cost < best_cost **then**
            best_arrangement ← new_arrangement
            best_cost ← new_cost
        end if
    else
        Accept new_arrangement with probability $P = e^{-\text{improvement}/\text{temperature}}$
    end if
    temperature ← $\alpha\times$ temperature {cooling schedule}
end while
Return: best_arrangement =0

Algorithm 20 ComputeCost Function for Jigsaw Puzzle

Input: arrangement, tiles
Output: cost
cost ← 0
for each tile in the grid **do**
    if right neighbor exists **then**
        cost ← cost + mismatch between right edges
    end if
    if bottom neighbor exists **then**
        cost ← cost + mismatch between bottom edges
    end if
end for
Return: cost =0

E. Analysis

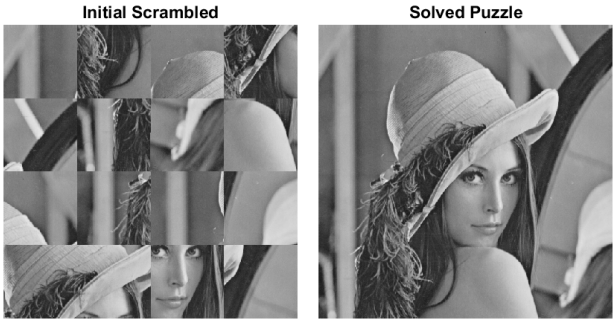

Fig. 4: Reconstructed puzzle using the simulated annealing algorithm.

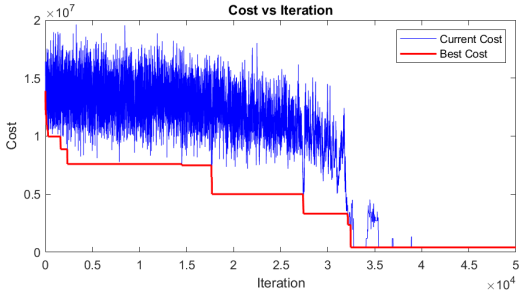

Fig. 5: Cost Vs Iteration.

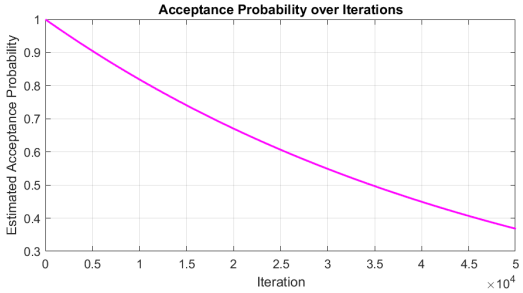

Fig. 6: Acceptance Probability Over Iterations.

F. Conclusion

The **Simulated Annealing (SA)** algorithm offers an effective strategy for addressing the jigsaw puzzle reconstruction problem. Although the state space is extremely large (16! possible configurations), SA efficiently balances exploration and exploitation through its probabilistic acceptance criterion. While it may not always guarantee a perfectly solved puzzle, the approach consistently minimizes mismatches between adjacent tiles and demonstrates the strength of **metaheuristic optimization** in tackling complex combinatorial image reconstruction tasks.

CHALLENGE PROBLEM

X. GENERATING MELODY USING SIMULATED ANNEALING

A. Objective

The objective of this project is to design an algorithm that generates melodies in the style of **Raag Bhairav**, ensuring adherence to its traditional structure, including the **Aroh** (ascending scale), **Avroh** (descending scale), characteristic phrases (**pakad**), and emphasis on key notes (**vadi** and **samvadi**). The aim is to preserve the raga's musical essence while enabling computational creativity.

B. Problem Statement

The problem is to develop a computer program that can automatically generate melodies conforming to the grammar and rules of **Raag Bhairav** using evolutionary or metaheuristic techniques such as **genetic algorithms** or **simulated annealing**. The system must produce sequences that respect the raga's scales, characteristic phrases, and tonal emphasis, thereby capturing both its structural and aesthetic qualities.

C. Methodology

State Representation:: A melody is represented as a sequence of notes with durations:

$$M = [(n_1, t_1), (n_2, t_2), \ldots, (n_L, t_L)]$$

where
$n_i \in \{S, r, G, m, P, d, N, S'\}$ are the valid notes from Raag Bhairav's **Arohana** and **Avarohana**.
t_i is the duration of the note selected from predefined rhythmic patterns.

Initialization:

Generate an initial population (GA) or random melody (SA).
Parameters: population size, melody length L, mutation rate, number of generations, or temperature T_0 (for SA).

Cost / Fitness Function:

The fitness (or cost) evaluates how well a melody conforms to **Raag Bhairav grammar**:

Penalties:

Invalid notes outside Raag Bhairav.
Large melodic jumps: $|index(n_{i+1}) - index(n_i)| > 3$.
Excessive repetition of notes.
Rhythm discontinuity.

Rewards:

Presence of **signature patterns (pakad)** such as *G m r G, r S, d P m G*.
Emphasis on **Vadi (d)** and **Samvadi (r)**.
Melodies starting/ending on **Sa (S)**.
Smooth note progression with longer sustained notes.

Mathematical Formulation:

$$C(M) = \sum_{i=1}^{L-1} P_{\text{jump}}(n_i, n_{i+1}) + \sum_{i=1}^{L} P_{\text{invalid}}(n_i) - \sum_{p \in \text{patterns}} R_{\text{pattern}}(p \subseteq M) - \sum_{n \in \{d,r\}} R_{\text{emphasis}}(n) + .$$

The goal is to minimize $C(M)$.

Neighbor Generation (SA) / Variation Operators (GA):

Mutation: Replace a note with another valid note or modify its duration.
Crossover (GA only): Combine segments of two parent melodies.
Neighbor move (SA): Swap, replace, or alter short subsequences to create new melodic variants.

Optimization Strategy:

Genetic Algorithm (GA):

1) Initialize population of random melodies.
2) Evaluate fitness of each melody.
3) Select parents and apply crossover/mutation to create offspring.
4) Replace old population with new generation.
5) Continue for set generations; output best melody.

Simulated Annealing (SA):

1) Start with random melody M_0, cost $C(M_0)$.
2) Iteratively generate a neighbor melody M'.
3) Accept M' if it has lower cost, or probabilistically if worse:
$$P_{accept} = e^{\frac{C(M) - C(M')}{T}}$$
4) Gradually reduce temperature: $T \leftarrow \alpha T$.
5) Stop when $T \to 0$ or iteration limit reached.

D. Solving Strategy:

The melody generation is treated as a **combinatorial optimization problem**. Using **Genetic Algorithms (GA)** or **Simulated Annealing (SA)**, melodies are iteratively refined to **minimize rule violations** and **maximize adherence** to Raag Bhairav's *scales*, *signature phrases*, and *tonal structure*. After optimization, it will return the **best melody** M^* that best follows Raag Bhairav's grammar, captures its emotional essence, and maintains structural integrity.

Algorithm 21 GenerateRaagBhairavMelody using Simulated Annealing

Input: L (melody length)
Output: $best\_melody$, $best\_score$
$current\_melody \leftarrow$ RandomMelody(L)
$current\_score \leftarrow$ Evaluate($current\_melody$)
$best\_melody \leftarrow current\_melody$
$best\_score \leftarrow current\_score$
$T \leftarrow T_0$ (initial temperature)
for $iteration = 1$ to $MaxIterations$ **do**
  $neighbor \leftarrow$ CreateVariation($current\_melody$)
  $neighbor\_score \leftarrow$ Evaluate($neighbor$)
  $\Delta Score \leftarrow neighbor\_score - current\_score$
  if $\Delta Score \leq 0$ **or** $random() < e^{-\Delta Score/T}$ **then**
    $current\_melody \leftarrow neighbor$
    $current\_score \leftarrow neighbor\_score$
    if $current\_score < best\_score$ **then**
      $best\_melody \leftarrow current\_melody$
      $best\_score \leftarrow current\_score$
    end if
  end if
  $T \leftarrow T \times CoolingRate$
end for
Return: $best\_melody$, $best\_score$ =0

E. Analysis:

Generates melodies of specified length that adhere to **Raag Bhairav grammar**.
Produces a **MIDI file** using a library such as `mido`. Each note and its duration are mapped to MIDI pitches and timings, creating a playable digital version of the Raag Bhairav melody.
Provides **visualizations** to analyze the melody:

- **Melody contour** over time.
- **Optimization score** progression across iterations.
- **Note frequency distribution**, highlighting key notes (Vadi: d, Samvadi: r).
- **Note durations** throughout the melody.

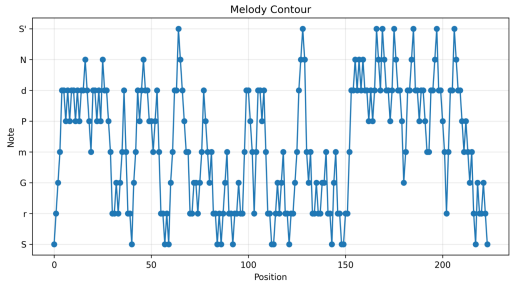

Fig. 7: Melody Contour.

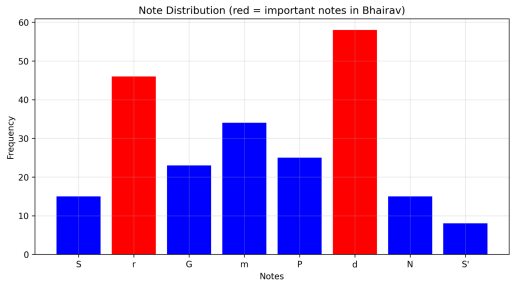

Fig. 8: Distribution of Notes.

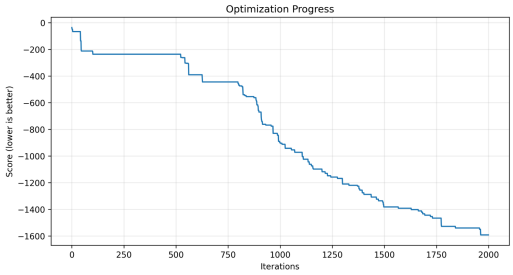

Fig. 9: Optimization Progress.

F. Conclusion

The implementation successfully generates melodies in **Raag Bhairav** using a Simulated Annealing-inspired optimization. By emphasizing signature phrases and the **Vadi/Samvadi notes**, the generated melodies maintain the raga's stylistic and tonal integrity. Visualization graphs provide insights into melodic contour, rhythmic flow, and optimization progress, while the MIDI output allows direct auditory verification against classical references.

REFERENCES

1) D. Delahaye, S. Chaimatanan, and M. Mongeau, *Simulated Annealing: From Basics to Applications*.
2) D. Khemani, *A First Course in Artificial Intelligence*. McGraw-Hill Education, Chapter 4.
3) S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th Edition. Pearson, Chapters 3–4.
4) Pratikiiitv, "CS307 GitHub Repository," Available: https://github.com/pratikiiitv/CS307/tree/main.
5) GeeksforGeeks, "What is Simulated Annealing?" Available: https://www.geeksforgeeks.org/artificial-intelligence/what-is-simulated-annealing/.
6) PyPI, "MIDIUtil," Available: https://pypi.org/project/MIDIUtil/.
7) Tanarang, "Raag Bhairav," Available: https://tanarang.com/raag-bhairav/.

GITHUB REPOSITORY

Synapse3-AI-labs: Source code and implementations are available at our GitHub Repository https://github.com/saranshnaik/Synapse3-AI-labs