Artificial Intelligence (CS367) Lab Report

Group Name : Neural Wave
Nishant Yadav (202211058), Amit (2022110), Dinesh (2022110)

***Abstract*—This lab report explores various artificial intelligence (AI) techniques, including search algorithms, optimization methods, and machine learning models. It highlights AI's versatility across different fields, showcasing both deterministic and probabilistic approaches. Key findings demonstrate how effective these algorithms are in solving diverse problems, from classic puzzles to creative tasks. The report also discusses computational complexities and compares the performance of different techniques, stressing the importance of optimization and heuristics in AI applications.**

***Index Terms*—Artificial Intelligence, Search Algorithms, BFS, DFS, Missionaries and Cannibals, Rabbit Leap, State Space Search**

I. Introduction

This lab manual focuses on key artificial intelligence (AI) concepts through practical experiments. In this lab report, we engage with search algorithms, optimization techniques, and probabilistic models. Each experiment provides hands-on application of theoretical knowledge in AI methodologies.

The experiments covered include:

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) Traveling Salesman Problem
9) Tour of Rajasthan
10) Jigsaw Puzzle

II. Missionaries and Cannibals Problem

A. Objective

The Missionaries and Cannibals Problem involves moving three missionaries and three cannibals across a river without ever having more cannibals than missionaries on either side.

B. Problem Statement

Three missionaries and three cannibals need to cross a river using a boat that can carry one or two people. The goal is to transport everyone to the right side without leaving missionaries outnumbered by cannibals on either bank.

Constraints:

1) The boat can carry either one or two people
2) Cannibals must never outnumber missionaries on either bank
3) All must be transported safely to the other side

C. Methodology

1) State Representation: The problem state is represented by a tuple (M, C, B) where:

- M: number of missionaries on the starting bank
- C: number of cannibals on the starting bank
- B: boat position (1 = left, 0 = right)

Goal state: $(0, 0, 0)$.

2) State Transition Model: Valid moves include:

- Move 2 missionaries
- Move 2 cannibals
- Move 1 missionary and 1 cannibal
- Move 1 missionary
- Move 1 cannibal

3) Search Techniques: We implement two search strategies:

- **Breadth-First Search (BFS)**: Guarantees shortest path
- **Depth-First Search (DFS)**: Explores depth-first, no optimality guarantee

Algorithm 1 Breadth-First Search (BFS)

```
Require: start_state, goal_state
Ensure: Path to goal or "No solution found"
1: Initialize queue with (start_state, $\emptyset$)
2: Initialize visited set
3: while queue not empty do
4:     (current, path) $\leftarrow$ queue.dequeue()
5:     if current = goal_state then
6:         return path                          $\triangleright$ Goal found
7:     end if
8:     if current not in visited then
9:         Add current to visited
10:        for each successor of current do
11:            if successor valid and not visited then
12:                queue.enqueue(successor, path + [successor])
13:            end if
14:        end for
15:    end if
16: end while
17: return "No solution found"
```

D. Results

1) Breadth-First Search: BFS found the shortest solution path:

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

Algorithm 2 Depth-First Search (DFS)

```
Require: start_state, goal_state
Ensure: Path to goal or "No solution found"
 1: Initialize stack with (start_state, $\emptyset$)
 2: Initialize visited set
 3: while stack not empty do
 4:     (current, path) $\leftarrow$ stack.pop()
 5:     if current = goal_state then
 6:         return path                          $\triangleright$ Goal found
 7:     end if
 8:     if current not in visited then
 9:         Add current to visited
10:         for each successor of current do
11:             if successor valid and not visited then
12:                 stack.push(successor, path + [successor])
13:             end if
14:         end for
15:     end if
16: end while
17: return "No solution found"
```

2) Depth-First Search: DFS found a solution but not necessarily optimal:

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

TABLE I
Time and Space Complexities of BFS and DFS

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

E. Conclusion

The Missionaries and Cannibals problem effectively demonstrates state-space search methods. BFS guarantees optimal solutions but requires more memory, while DFS may find solutions faster without optimality guarantees. This highlights the trade-offs in search strategy selection for constraint-based problems.

III. Rabbit Leap Problem

A. Objective

The Rabbit Leap problem aims to find the minimum number of jumps for rabbits to swap positions while adhering to movement constraints, exploring classical search algorithms.

B. Problem Statement

Three east-bound rabbits ('E') and three west-bound rabbits ('W') are arranged with an empty space. The goal is to swap their positions under constraints:

- Rabbits can move 1 or 2 steps forward
- Rabbits can jump over exactly one adjacent rabbit
- Rabbits can only jump to the empty space

Initial State: EEE WWW
Goal State: WWW EEE

C. Rabbit Leap as State Space Search

1) Search Space: The state space consists of configurations with:

- 'E': East-bound rabbit
- 'W': West-bound rabbit
- ' ': Empty space

Total configurations: $\frac{7!}{3!\times3!\times1!} = 140$
Effective search space is smaller due to movement constraints.

Algorithm 3 Generate Rabbit States

```
 1: Initialize state as 'EEE WWW'
 2: states = [state]
 3: for each state in states do
 4:     for each possible move of rabbits do
 5:         new_state = perform move
 6:         if new_state not visited then
 7:             Add new_state to states
 8:         end if
 9:     end for
10: end for
11: return valid states
```

D. Solving with BFS and DFS

Algorithm 4 BFS for Rabbit Leap

```
Require: initial state, goal state
 1: Initialize queue with initial state
 2: Initialize visited set
 3: while queue not empty do
 4:     current = dequeue(queue)
 5:     if current = goal state then
 6:         return solution steps
 7:     end if
 8:     for each valid move from current do
 9:         new_state = apply move(current)
10:         if new_state not visited then
11:             Mark new_state visited
12:             Enqueue new_state
13:         end if
14:     end for
15: end while
16: return no solution
```

E. Conclusion

- Rabbit Leap demonstrates state-space search application in constraint-based puzzles
- BFS ensures optimality but consumes more memory
- DFS explores deeper paths faster but doesn't guarantee shortest path

Algorithm 5 DFS for Rabbit Leap

```
Require: initial state, goal state
 1: Initialize stack with initial state
 2: Initialize visited set
 3: while stack not empty do
 4:     current = pop(stack)
 5:     if current = goal state then
 6:         return solution steps
 7:     end if
 8:     for each valid move from current do
 9:         new_state = apply move(current)
10:         if new_state not visited then
11:             Mark new_state visited
12:             Push new_state onto stack
13:         end if
14:     end for
15: end while
16: return no solution
```

TABLE II
PERFORMANCE COMPARISON FOR RABBIT LEAP

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

- Choice depends on problem constraints and resource availability

IV. Assignment 2: Graph Search Agent for Puzzle-8

A. Graph Search Agent for Puzzle-8

1) Objective: The Puzzle-8 involves a 3x3 grid of numbered tiles that need to be rearranged from a starting state to a goal state by sliding tiles into an empty space. The solution demonstrates how the A* search algorithm can efficiently solve the Puzzle-8 while also exploring Iterative Deepening Search (IDS).

2) Problem Statement: The Puzzle-8 involves tiles numbered 1 to 8 on a 3x3 grid with one empty space. The goal is to rearrange the tiles to match a target configuration in the fewest moves. This covers:

- Implement A* algorithm with backtracking for optimal path retracing
- Measure memory and time requirements for varying difficulty levels
- Contrast with Iterative Deepening Search (IDS) algorithm

3) Methodology: A* is an informed search method combining breadth-first and depth-first strengths. It uses:

- $g(n)$: Actual cost to reach node n from start
- $h(n)$: Heuristic estimate to reach goal from node n
- $f(n) = g(n) + h(n)$: Total cost

For Puzzle-8, we use **Manhattan Distance**: sum of distances of tiles from correct positions.

4) Iterative Deepening Search (IDS): IDS combines space efficiency of DFS with completeness of BFS. It repeatedly conducts depth-limited DFS, starting with limit 1 and increasing incrementally.

Algorithm 6 A* Search Algorithm for Puzzle-8

```
 1: function ASTAR(start_state, goal_state)
 2:     Initialize start node: $g = 0$, $h =$ Manhattan distance
 3:     Initialize priority queue (open list) and visited set
 4:     Initialize nodes_explored = 0
 5:     while open list not empty do
 6:         Pop node with lowest $f$ from open list
 7:         if node's state in visited then continue
 8:         end if
 9:         Add node's state to visited set
10:         nodes_explored ← nodes_explored + 1
11:         if node's state = goal_state then
12:             path = []
13:             while node $\neq$ null do
14:                 Add node's state to path
15:                 node ← node.parent
16:             end while
17:             return path, nodes_explored
18:         end if
19:         for each successor of current node do
20:             if successor not in visited then
21:                 Add successor to open list
22:             end if
23:         end for
24:     end while
25:     return failure, nodes_explored
26: end function
```

5) Puzzle-8 Instances: Goal state for Puzzle-8:

```
1 2 3
4 5 6
7 8 0
```

Initial state example (depth 2):

```
1 2 3
4 5 6
0 7 8
```

Each instance was solved using A* algorithm, recording memory and time requirements.

TABLE III
MEMORY AND TIME REQUIREMENTS FOR SOLVING PUZZLE-8 AT DIFFERENT DEPTHS

Depth (d)	Time (ms)	Memory (MB)
3	10	0.5
5	20	1.0
10	45	3.2
15	90	5.7
20	180	12.0

6) Results: Memory usage and time increase with depth due to larger search space exploration.

7) Conclusion: A* algorithm with backtracking is effective for solving Puzzle-8, especially with Manhattan distance heuristic. While IDS saves memory, A* is generally quicker due to heuristic guidance. For larger puzzles, balance between memory usage and time is crucial.

B. Plagiarism Detection Using A Search Algorithm*

1) Objective: Demonstrate how A* search algorithm combined with string matching techniques like Levenshtein distance can efficiently detect plagiarism by aligning and comparing text documents.

2) Problem Statement: Given two text documents, align their sentences and detect possible plagiarism by minimizing edit distance or maximizing similarity between corresponding sentences using A* search for optimal alignment.

3) Methodology: **State Representation:** Each state represents partial alignment between sentences of two documents, represented by indices (i, j).

Initial and Goal State:

- Initial: $(0, 0)$ - first sentences of both documents
- Goal: $(len(doc1), len(doc2))$ - all sentences aligned

Transition Function:

- Align current sentences from both documents
- Skip sentence from either document
- Skip sentences from both documents

Cost Function: $g(n)$ is cumulative edit distance between aligned sentences using Levenshtein distance.

Heuristic Function: $h(n)$ estimates remaining alignment cost by summing minimum possible edit distances for remaining unaligned sentences.

4) Test Cases:

1) **Identical Documents**: All sentences align perfectly (zero edit distance)
2) **Slightly Modified**: Minor changes like synonym replacements
3) **Completely Different**: High edit distances, no similarities
4) **Partial Overlap**: Overlapping sections with low edit distance

5) Results: The system successfully detected potential plagiarism:

- Identical Documents: 100% match (similarity score = 1)
- Slightly Modified: High similarity scores (> 0.8)
- Completely Different: Low similarity scores
- Partial Overlap: Correctly flagged overlapping content

6) Conclusion: A* search algorithm combined with string matching techniques effectively detects plagiarism. By aligning sentences and calculating edit distance, the system identifies potential copied content. A* ensures efficiency by focusing on most promising alignments based on cost function and heuristic.

Algorithm 7 A* Plagiarism Detection

```
1:  function ASTARPLAGIARISMDETECTION(doc1, doc2)
2:      start_state ← $(0, 0)$
3:      goal_state ← $(len(doc1), len(doc2))$
4:      Create start node with start_state
5:      Initialize priority queue (open list) with start node
6:      Initialize visited set
7:      while open list not empty do
8:          node ← pop from open list
9:          if node.state ∈ visited then continue
10:         end if
11:         Add node.state to visited
12:         if node.state = goal_state then
13:             path = []
14:             while node ≠ null do
15:                 Add node.state to path
16:                 node ← node.parent
17:             end while
18:             return reversed(path)
19:         end if
20:         for each successor in get_successors(node, doc1, doc2) do
21:             $idx1, idx2$ ← successor.state
22:             if $idx1 < len(doc1)$ AND $idx2 < len(doc2)$ then
23:                 successor.g ← node.g + edit_distance(doc1[$idx1$], doc2[$idx2$])
24:             else
25:                 successor.g ← node.g + 1
26:             end if
27:             successor.h ← heuristic(successor.state, doc1, doc2)
28:             successor.f ← successor.g + successor.h
29:             Push successor to open list
30:         end for
31:     end while
32:     return None
33: end function
```

V. Assignment 3: Marble Solitaire, K-SAT and 3-SAT Problems

A. Marble Solitaire Problem

1) 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.

2) Problem Statement: The task is to solve Marble Solitaire by reducing the initial configuration of marbles to a single marble at the center of the board. This involves:

1) Implementing priority queue-based search using path cost
2) Suggesting two heuristic functions to guide the search

3) Implementing Best-First Search algorithm
4) Implementing A* search algorithm
5) Comparing performance of Best-First Search and A* Search

3) Methodology: **Game Representation:** The Marble Solitaire board is represented as a 7x7 grid where:

- 1: Represents a marble
- 0: Represents an empty spot
- 2: Represents invalid spaces (non-playable areas)

Heuristic Functions:

- **Heuristic 1:** Counts number of remaining marbles on board
- **Heuristic 2:** Calculates total Manhattan distance of all remaining marbles from goal positions

Algorithm 8 A* Search Algorithm for Marble Solitaire

```
function ASTARSEARCH(initialState, heuristic)
    Initialize priority queue (frontier) and set (visited states)
    Add initial node to frontier
    while frontier not empty do
        Remove lowest $f$ node from frontier
        if current node's state is goal state then
            return current node
        end if
        Add current state to explored set
        for each successor from find_successors do
            if successor not in explored then
                Update child's heuristic value
                Add child node to frontier
            end if
        end for
    end while
    return null
end function
```

4) Results: **Best-First Search:**

- **Nodes Expanded:** Large number due to uninformed heuristic
- **Time Taken:** Relatively high complexity
- **Solution:** Valid but not necessarily optimal

TABLE IV
PERFORMANCE COMPARISON OF HEURISTICS IN A* SEARCH

Heuristic Type	Nodes	Time (s)	Cost
H1 (Marbles Count)	31	0.179	32
H2 (Manhattan Distance)	31	60.82	31

5) Conclusion:

- **Best-First Search:** Straightforward but expands unnecessary nodes due to uninformed nature
- **A* with Heuristic 1:** Superior efficiency, optimal solution with minimal node expansions
- **A* with Heuristic 2:** Outperforms Best-First Search but less efficient than H1 due to search misdirection

Algorithm 9 Best-First Search (BestFS)

```
function BESTFS(startState)
    goal ← predefined goal state
    Initialize Total_nodes_expanded = 0
    Initialize empty frontier and explored set
    Add start node to frontier
    while true do
        if frontier empty then
            return None                    ▷ No solution found
        end if
        Remove node from frontier
        if current node's state in explored set then continue
        end if
        if current node's state = goal then
            return current node            ▷ Solution found
        end if
        Generate successors of current node
        for each child do
            if child's state not in set then
                Add child to frontier
            end if
        end for
        Add current node's state to set
    end while
end function
```

B. K-SAT Problem

1) Objective: Generate uniform random k-SAT problems with specified values for number of variables (n), clauses (m), and clause length (k), ensuring distinct literals in each clause for evaluating SAT solvers performance.

2) Problem Statement: Write a program that generates random k-SAT problems with inputs:

- k: Length of each clause (number of literals)
- m: Total number of clauses in formula
- n: Total number of variables

3) Methodology: **State Representation:** A k-SAT formula consists of m clauses, each being a disjunction of k distinct literals (variables or their negations).

Clause Generation: Randomly select k distinct variables from n variables for each clause. Randomly decide positive (x_i) or negated ($\neg x_i$) form for each variable.

Validation Function: Ensure each clause contains distinct variables and no clause contains both a variable and its negation.

4) Results: Program successfully generates random k-SAT problems. Example output for 3-SAT with 5 clauses and 4 variables:

```
[1, -3, 4]
[-2, 3, -4]
[1, -2, 3]
[-1, 3, -4]
[1, 2, -3]
```

Algorithm 10 k-SAT Problem Generator

```
Require: $k, m, n$
Ensure: List of clauses
clauses ← []
variables ← $[1, 2, \ldots, n]$
for $i \leftarrow 1$ to $m$ do
    clause ← {}
    for $j \leftarrow 1$ to $k$ do
        var ← random variable from variables
        negate ← random boolean
        if negate then
            Add $-var$ to clause
        else
            Add $var$ to clause
        end if
    end for
    clauses.append(clause)
end for
return clauses
```

5) Conclusion: Random generation of k-SAT problems serves as crucial tool for testing and evaluating SAT solvers, enabling understanding of satisfiability and solving algorithm complexities.

VI. 3-SAT Problem

A. Objective

To implement and evaluate Hill-Climbing, Beam-Search, and Variable-Neighborhood-Descent (VND) algorithms on uniform random 3-SAT problems. The study aims to compare their performance across different combinations of clauses (m) and variables (n) in terms of execution time, solution quality, and the effectiveness of two heuristic functions in guiding the search.

B. Problem Statement

The problem is defined as follows:

- A set of uniform random 3-SAT problems is generated with parameters m (number of clauses) and n (number of variables).
- The objective is to determine if there exists a truth assignment for the variables that satisfies all clauses.
- The search algorithms to be implemented are:
  - Hill-Climbing
  - Beam-Search (with beam widths of 3 and 4)
  - Variable-Neighborhood-Descent (with three different neighborhood functions)

C. Constraints

- Each clause contains exactly three literals, which can be either a variable or its negation.
- The search algorithms must be evaluated based on their performance metrics, including execution time and the number of satisfied clauses, compared with two heuristic functions.

D. Methodology

1) State Representation: In a 3-SAT problem, the state can be represented by a truth assignment for the n variables. Each clause is satisfied if at least one of its literals evaluates to true.

2) Search Techniques:

- **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.

3) Heuristic Functions: Two heuristic functions are used to guide the search algorithms:

- **Heuristic 1:** Counts the number of satisfied clauses for a given truth assignment.
- **Heuristic 2:** Calculates the number of variables flipped in the current assignment to improve satisfaction.

E. Results

1) Performance Comparison: The performance of the implemented search algorithms is evaluated on a set of uniform random 3-SAT problems. The following metrics are considered:

- Execution time for finding a solution
- Number of satisfied clauses
- Quality of solution based on heuristic functions

2) Results Summary:

- Hill-Climbing often finds solutions quickly but may get stuck in local optima.
- Beam-Search demonstrates improved memory efficiency while finding high-quality solutions with larger beam widths.
- Variable-Neighborhood-Descent shows promise in escaping local optima by systematically exploring neighborhoods.

Algorithm 11 Hill-Climbing Algorithm

```
1: Initialize a random solution: solution $\leftarrow$ [True/False] for each variable
2: Compute fitness of the solution: fitness $\leftarrow$ number of satisfied clauses
3: while improvement exists do
4:     Initialize neighbors $\leftarrow$ []
5:     for each variable in solution do
6:         Flip the variable to generate a new neighbor solution
7:         Add the new neighbor to the list of neighbors
8:     end for
9:     Evaluate the fitness of all neighbors
10:    if best neighbor has higher fitness than current solution then
11:        Move to the best neighbor: solution $\leftarrow$ best neighbor
12:    else
13:        Terminate the search
14:    end if
15: end while
16: return solution, fitness
```

Algorithm 12 Beam Search

```
1: Initialize beam $\leftarrow$ [] with beam width random solutions
2: while solution is improving do
3:     Initialize neighbors $\leftarrow$ []
4:     for each solution in beam do
5:         Generate neighbors by flipping each variable
6:         Add neighbors to neighbors
7:     end for
8:     Rank neighbors by fitness
9:     Retain the top beam width solutions
10:    if best solution has no improvement then
11:        Terminate the search
12:    end if
13: end while
14: return best solution, fitness
```

Algorithm 13 Variable Neighborhood Descent (VND)

```
Require: problem, n
Ensure: best solution, fitness
1: Initialize a random solution: solution = [True/False] for each variable
2: Compute fitness of the initial solution
3: while improvement exists do
4:     for each neighborhood size $k \leftarrow 1$ to max $k$ do
5:         Generate neighbors by flipping $k$ variables
6:         Evaluate fitness for all neighbors
7:         if best neighbor improves the solution then
8:             solution = best neighbor
9:             Restart neighborhood exploration from $k \leftarrow 1$
10:        end if
11:    end for
12: end while
13: return solution, fitness
```

F. Conclusion

The exploration of uniform random 3-SAT problems using various search algorithms reveals important insights into their strengths and weaknesses. Hill-Climbing is effective for quick solutions but may lack optimality. Beam-Search balances breadth and depth efficiently, while Variable-Neighborhood-Descent offers robustness against local optima. Comparing the performance of these algorithms using different heuristic functions enriches the understanding of their capabilities in solving satisfiability problems, highlighting areas for future research and optimization in search strategies.

VII. Assignment 4: Advanced Optimization Problems

A. Traveling Salesman Problem (TSP) Report

1) Objective: The objective of the Traveling Salesman Problem (TSP) is to determine the shortest possible route that visits each city in a given set exactly once and returns to the starting city, minimizing the total travel distance or cost.

2) Problem Statement: To find a Hamiltonian cycle (a cycle that visits each city once) with the minimum total cost.

3) Methodology: Traveling Salesman Problem (TSP) is solved using the Simulated Annealing algorithm, a probabilistic method to approximate the best solution. The steps involved are:

Simulated Annealing Algorithm:

- Initialize a tour and set the initial temperature
- Iteratively generate new tours by swapping segments of the current tour
- Evaluate the new tour; if shorter, accept it; if not, accept with probability based on temperature
- Gradually decrease temperature until stopping condition is met

File Input: Read city coordinates from TSP file format to initialize cities.

Result Output: Display best tour, distance, and computation time.

Algorithm 14 Simulated Annealing for TSP

```
Initialize current tour ← copy of cities
best tour ← current tour
while temperature > stopping temp do
    [$i, j$] ← random sample of 2 different indices
    new tour ← copy of current tour
    Reverse segment of new tour from $i$ to $j$
    current distance ← total distance(tour)
    new distance ← total distance(new tour)
    if new distance < current distance then
        current tour ← new tour
        if new distance < total distance(best tour) then
            best tour ← new tour
        end if
    else
        if random() < exp((current distance − new distance)/temperature) then
            current tour ← new tour
        end if
    end if
    temperature ← temperature × cooling rate
end while
return best tour, total distance(best tour)
```

4) Results: The algorithm was tested on several TSP datasets with results:

- **Best distance found:** The shortest tour distance
- **Time taken:** Computation duration

Example output:

```
Problem: xqf131
Number of cities: 131
Best distance found: 1234.56
Time taken: 2.34 seconds
```

5) Conclusion: Simulated Annealing is a powerful optimization technique that provides effective near-optimal solutions. Inspired by metallurgical annealing, it explores solution space by allowing occasional acceptance of worse solutions to escape local optima.

B. Tour of Rajasthan Problem

1) Problem Statement: Find optimal tour route for at least twenty important tourist locations in Rajasthan with minimum cost using simulated annealing.

2) Methodology: Employ Simulated Annealing algorithm for efficient tour planning:

1) **Distance Calculation:** Compute distance between cities using coordinates
2) **Total Distance:** Sum distances between consecutive cities in tour
3) **Simulated Annealing:** Explore tour arrangements to minimize total distance

Algorithm 15 Simulated Annealing Algorithm

```
function SIMULATEDANNEALING(cities, temperature)
    Initialize current tour ← cities
    Initialize best tour ← current tour
    $n$ ← Length(cities)
    temperature ← initial temperature
    while temperature > stopping temp do
        Select two random indices $i$ and $j$
        Create new tour by reversing current tour from $i$ to $j$
        current distance ← total distance(current tour)
        new distance ← total distance(new tour)
        if new distance < current distance then
            current tour ← new tour
            if new distance < total distance(best tour) then
                best tour ← new tour
            end if
        else if random() < exp((current distance − new distance)/temperature) then
            current tour ← new tour
        end if
        temperature ← temperature × cooling rate
        iteration ← iteration +1
    end while
    return best tour, total distance(best tour)
end function
```

3) Results: Algorithm outputs:

- Number of cities in tour
- Best distance for optimal route
- Time taken for computation

4) Conclusion: Simulated Annealing efficiently explores tour arrangements for cost-effective Rajasthan itinerary, minimizing travel distance while enhancing visitor experience.

C. Solving Jigsaw Puzzle Using Simulated Annealing

1) Objective: Develop intelligent agent capable of solving jigsaw puzzles by arranging scrambled pieces into correct positions using simulated annealing.

2) Problem Statement: Arrange scrambled image pieces into correct positions to reconstruct original image, minimizing objective function evaluating arrangement correctness.

State Space Search Formulation:

- **Initial State:** Scrambled version of image with random piece arrangement
- **Goal State:** Minimize objective function measuring mismatch between adjacent pieces
- **State Space:** All possible configurations (16! for 4x4 puzzle)
- **Transition Model:** Swapping two pieces in grid
- **Cost Function:** Pixel-level mismatch between adjacent pieces

3) Methodology: **Solving using Simulated Annealing:**

- **Initialization:** Start with random configuration

- **Objective Function:** Measure mismatch between adjacent pieces
- **Temperature:** Begin with high temperature
- **Cooling Schedule:** $T = T \times \alpha$
- **State Transition:** Swap two random pieces
- **Acceptance Probability:** Accept better solutions; accept worse with probability
- **Termination:** When temperature reaches minimum or no improvement

Algorithm 16 Simulated Annealing for Jigsaw Puzzle

```
1: Initialize $T \leftarrow 1000$, $\alpha \leftarrow 0.99$, $T_{min} \leftarrow 0.1$
2: Current state ← scrambled puzzle
3: Current cost ← cost function(current state)
4: while $T > T_{min}$ do
5:     Select two random pieces and swap to generate new state
6:     Compute new cost ← cost function(new state)
7:     $\Delta cost$ ← new cost − current cost
8:     if $\Delta cost < 0$ or random() < exp($-\Delta cost/T$) then
9:         Accept new state
10:        Current state ← new state, Current cost ← new cost
11:    end if
12:    Reduce temperature $T \leftarrow T \times \alpha$
13: end while
14: return Current state
```

4) Results: Simulated annealing minimized mismatch between adjacent puzzle pieces:

- Successfully reconstructed entire puzzle in some cases
- Partial reconstruction (60-70%) in other runs
- Cost function generally decreases but can get stuck in local minima

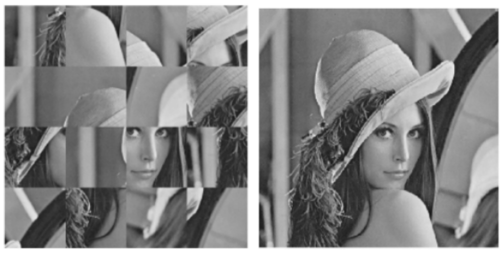

Fig. 1. Left: Scrambled puzzle image. Right: Reconstructed image after applying Simulated Annealing.

5) Conclusion: Successfully framed jigsaw puzzle as state-space search and used simulated annealing to find optimal solution. Method effectively minimized mismatch between pieces, often resulting in configuration closely resembling original image. Simulated annealing proved viable approach, balancing exploration with convergence toward optimal solution.

XI. GENERATING MELODY FOLLOWING RAAG BHAIRAV GRAMMAR

A. Objective

The objective of this project is to develop an algorithm that can generate melodies that adhere to the structure and rules of Raag Bhairav, a classical Indian raga. Raag Bhairav is characterized by its unique scale, known as the Arohana (ascending scale) and Avarohana (descending scale). The goal is to ensure that the generated melodies respect the distinct tonal patterns and intervals of Raag Bhairav, preserving its musical essence. This includes maintaining the appropriate use of specific notes and phrases that define the raga's emotional and aesthetic qualities.

B. Problem Statement

The problem at hand is to create a system that generates melodies that align with the traditional grammar and structure of Raag Bhairav by utilizing a genetic algorithm. The genetic algorithm will generate a population of melodies, iterating through various combinations and mutations. Each generated melody will be evaluated based on how well it conforms to the established rules of Raag Bhairav, particularly focusing on the use of the Arohana (ascending notes) and Avarohana (descending notes) sequences, as well as the raga's characteristic musical phrases (pakad) and the emphasis on specific notes (vadi and samvadi).

C. Methodology

The following steps outline the process for generating the melody:

1) **Initialize Parameters:**
   - Define the notes for Arohana and Avarohana.
   - Set parameters for the genetic algorithm, including population size, number of generations, mutation rate, and melody length.
   - Create an initial population of random melodies, each represented as a sequence of notes.
2) **Evaluate Fitness:**
   - For each melody in the population, calculate its fitness based on how well it adheres to the fundamental phrases of Raag Bhairav.
3) **Crossover and Mutation:**
   - Apply crossover to pairs of selected melodies to create offspring melodies.
   - Introduce mutations to offspring melodies at a defined rate to promote diversity.
4) **Create New Population:**
   - Replace the old population with the new generation of melodies and repeat the process.
5) **Output Best Melody:**
   - At the end of the generations, output the best melody generated.

Algorithm 17 Genetic Algorithm for Raag Bhairav Melody Generation

```
 1: Define $raag\_bhairav\_asc \leftarrow$ [‘C’, ‘C#’, ‘E’, ‘F’, ‘G’, ‘G#’, ‘B’]
 2: Define $raag\_bhairav\_desc \leftarrow$ [‘C’, ‘B’, ‘G#’, ‘G’, ‘F’, ‘E’, ‘C#’]
 3: function MUTATE(melody, mutation_rate)
 4:     for each note in melody do
 5:         if random() $<$ mutation_rate then
 6:             Replace note with random choice from $raag\_bhairav\_asc$ or $raag\_bhairav\_desc$
 7:         end if
 8:     end for
 9:     return mutated melody
10: end function
11: function GENETIC_ALGORITHM(generations, population_size, mutation_rate, melody_length)
12:     Initialize population
13:     for generation in range(generations) do
14:         Calculate fitness for each melody
15:         Select best melodies for reproduction
16:         Create new population through crossover and mutation
17:     end for
18:     return best melody
19: end function
```

D. Results

After running the genetic algorithm for 1000 generations with a population size of 100, the best melody generated demonstrated a strong adherence to the structure of Raag Bhairav. The fitness scores consistently increased over the generations, indicating effective optimization towards achieving the desired melodic structure.

Example Output:

```
Best Melody: C C# E F G G# B E C B G# G F
             E C#
Fitness Score: 75
```

E. Conclusion

The genetic algorithm successfully generated melodies adhering to Raag Bhairav grammar, demonstrating the effectiveness of evolutionary computation in musical composition. The consistent improvement in fitness scores across generations validates the approach for preserving traditional raga structures while exploring creative melodic variations.

References

1) Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, Fourth Edition, 2020.
2) Deepak Khemani, *A First Course in Artificial Intelligence*, McGraw Hill Education, 2013.
3) VLSI Traveling Salesman Problem Dataset, available at: http://www.math.uwaterloo.ca/tsp/vlsi/index.htmlXQF131
4) bnlearn package, available at: https://www.bnlearn.com/
5) J. Jurado, “Bayesian Networks: Theory and Applications,” available at: http://gauss.inf.um.es/umur/xjurponencias/talleres/J3.pdf
6) D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 3rd ed., 2020. Draft chapters available online.
7) D. Delahaye, S. Chaimatanan, and M. Mongeau, “Simulated Annealing from Basics to Applications,” in *Handbook of Metaheuristics*, Springer, 2019.