ARTIFICIAL INTELLIGENCE LAB REPORT

Group Name: Acode

Tejas Pakhale-202211061, Rajat Kumar Thakur-202211070, Tanay Patel-202211094, Abhi Tundiya-202211095

***Abstract*—This comprehensive report explores a variety of Artificial Intelligence (AI) techniques, beginning with state-space search methods to solve classical problems such as the Missionaries and Cannibals and the Rabbit Leap. Both problems are modeled as state-space graphs, where Breadth-First Search (BFS) and Depth-First Search (DFS) are employed to compare optimality, memory usage, and computational trade-offs. Extending the exploration of search techniques, the A* algorithm is used to align text documents for plagiarism detection and solve the 8-puzzle problem, focusing on time and memory requirements.**

The report then delves into non-deterministic search methods, applying heuristic-driven algorithms like Hill-Climbing, Beam Search, and Variable-Neighborhood Descent to solve complex problems such as marble solitaire and random k-SAT generation. Simulated Annealing is applied to the Traveling Salesman Problem (TSP), showcasing its utility in probabilistic optimization and tackling large search spaces.

In the domain of probabilistic graphical models, the report implements Bayesian Networks to explore relationships between student course grades, employing naive Bayes classifiers for internship qualification prediction. The classifier's accuracy is tested using repeated experiments, comparing independent and dependent variable assumptions across training and testing datasets. Through these diverse AI techniques and experiments, the report highlights the strengths and applications of search algorithms, heuristic functions, and probabilistic models in solving complex real-world problems.

I. INTRODUCTION

This report delves into fundamental AI concepts through hands-on experiments that explore state-space search, heuristic-driven optimization, and probabilistic models. By implementing algorithms such as Breadth-First Search (BFS), Depth-First Search (DFS), A*, and Simulated Annealing, participants tackle classical problems like the Missionaries and Cannibals, the 8-puzzle, and the Traveling Salesman Problem. Additionally, the report covers probabilistic reasoning through Bayesian Networks, providing insights into AI's capacity for decision-making under uncertainty.

Week 1: Modeling and Solving Classical Search Problems Using State Space Search Techniques
To be able to model a given problem in terms of state space search problem and solve the same using BFS/ DFS.

Week 2: State Space Search
To design a graph search agent and understand the use of a hash table, queue in state space search.

Week 3: Heuristic And Non-Classical Search
To understand the use of Heuristic function for reducing the size of the search space. Explore non-classical search algorithms for large problems.

Week 4: Non-deterministic Search and Simulated Annealing
Non-deterministic Search | Simulated Annealing For problems with large search spaces, randomized search becomes a meaningful option given partial/ full-information about the domain.

Week 5: Graphical Models and Bayesian Networks Inference and Classification in R
Understand the graphical models for inference under uncertainty, build Bayesian Network in R, Learn the structure and CPTs from Data, naive Bayes classification with dependency between features.

Challenge Problem : Generating melody using simulated annealing
Generate a melody in North Indian Classical Raag Bhairav using either simulated annealing or the genetic algorithm

II. WEEK 1: MODELING AND SOLVING CLASSICAL SEARCH PROBLEMS USING STATE SPACE SEARCH TECHNIQUES

A. *Introduction*

State space search techniques are particularly valuable for addressing problems, where the objective is to navigate through different configurations or states to arrive at a solution. This report examines two such problems—the Missionaries and Cannibals and the Rabbit Leap—which have been widely used as benchmarks in AI research.

In the Missionaries and Cannibals scenario, the primary focus is on maintaining the balance between the two groups during their transport, necessitating strategic planning at every step. Conversely, the Rabbit Leap problem emphasizes the dynamic movement of two distinct groups, showcasing how they can navigate past each other while adhering to specific rules.

B. *UNDERSTANDING*

*1) **Search Algorithms**:* Search algorithms provide systematic approaches to explore a state space, navigating through possible configurations to reach a solution. They are vital in artificial intelligence (AI) for addressing complex problems by transitioning from an initial state to a goal state based on predefined rules and constraints. Before diving into practical applications, it is important to understand the types of search algorithms, such as *Breadth-First Search (BFS)* and *Depth-First Search (DFS)*, and the distinction between *uninformed* and *informed* search strategies, including the role of heuristics in the latter.

*2) **State Space**:* A state space is the set of all possible configurations or arrangements that a problem can take. It forms the foundation for search techniques, as each state represents a unique point in the problem's progression.

Researchers must first define the state space carefully for problems like *Missionaries and Cannibals* or *Rabbit Leap*, outlining each possible combination of missionaries, cannibals, boats, or rabbits that could occur throughout the puzzle.

*3) **Optimality**:* Optimality ensures that a search algorithm finds not just any solution, but the best possible one. For instance, in puzzles like the *Rabbit Leap* problem, an optimal solution would be the one that requires the least moves or satisfies the goal with minimal cost.

The choice of the search alogrithm is largely dependent on the constraint of the problem as well as the available system resources. Researchers focus on search algorithms like BFS, which guarantees an optimal solution in terms of the shortest path, compared to DFS, which may find a solution but not necessarily the optimal one while they may also opt for DFS when time is a constraint, especially if the solution obtained is close to the optimal one and the difference is not significant. This approach is practical when the speed of finding a solution outweighs the need for exact optimality.

C. Problem Statement

*1) **Missionaries and Cannibals**:* The missionaries and cannibals problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.

*2) **The Rabbit Leap**:* In the rabbit leap problem, three east-bound rabbits stand in a line blocked by three west-bound rabbits. They are crossing a stream with stones placed in the east west direction in a line. There is one empty stone between them. The rabbits can only move forward one or two steps. They can jump over one rabbit if the need arises, but not more than that.

*3) **Problem I**:*

- Model the problem as a state space search problem. How large is the search space?
- Solve the problem using BFS. The optimal solution is the one with the fewest number of steps. Is the solution that you have acquired an optimal one? The program should print out the solution by listing a sequence of steps needed to reach the goal state from the initial state.
- Solve the problem using DFS. The program should print out the solution by listing a sequence of steps needed to reach the goal state from the initial state.
- Compare solutions found from BFS and DFS. Comment on solutions. Also compare the time and space complexities of both.

D. Missionaries and Cannibals Problem

1) Problem Formulation: We represent the state as a tuple (M, C, B) where:

- M: Number of missionaries on the left side
- C: Number of cannibals on the left side
- B: Position of the Boat(1: Left bank, 2: Right Bank)

Initial State: $(3, 3, 1)$

Goal State: $(0, 0, 0)$

For missionaries, we have 4 possibilities

(0, 1, 2, or 3 missionaries)

For cannibals, we also have 4 possibilities

(0, 1, 2, or 3 cannibals)

For the boat, we have 2 possibilities

(0 or 1)

$$=> 4 \times 4 \times 2 = 32 \text{ possible states} \tag{1}$$

2) BFS Solution:

Algorithm 1 Breadth first Space for Missionaries and Cannibals Problem

```
procedure BFS(start_state, goal_state)
    queue ← deque([(start_state, [])])
    visited ← ∅
    while queue is not empty do
        (state, path) ← queue.popleft()
        if state ∈ visited then
            continue
        end if
        nodes_visited ← nodes_visited + 1
        path ← path + [state]
        if state = goal_state then
            print nodes_visited
            return path
        end if
        for successor ∈ get_successors(state) do
            queue.append((successor, path))
        end for
    end while
    return None
end procedure
```

3) DFS Solution:

Algorithm 2 Depth-First Search for Missionaries and Cannibals Problem

```
procedure DFS(start_state, goal_state)
    stack ← deque([(start_state, [])])
    visited ← ∅
    while stack is not empty do
        (state, path) ← stack.pop()
        if state ∈ visited then
            continue
        end if
        visited.add(state)
        path ← path + [state]
        if state = goal_state then
            return path
        end if
        for successor ∈ get_successors(state) do
            stack.append((successor, path))
        end for
    end while
    return None
end procedure
```

Complexity Type	BFS	DFS
Time Complexity	$O(b^d)$	$O(b^d)$
Space Complexity	$O(b^d)$	$O(bd)$
Total visited	15	14

Table I: Complexity Comparison of BFS and DFS

4) Complexity Comparison: where b is branching factor and d is depth of shallowest goal state.

E. *Analysis*

The time complexity of the BFS algorithm is $O(b^d)$, where b is the branching factor (the average number of successors per state), and d is the depth of the shallowest goal state. In this case, the `get_successors` function generates all possible successors for a given state, contributing to the branching factor. The BFS function's while loop iterates through the queue of states until the goal is reached or all nodes are visited, resulting in a time complexity of $O(b^d)$. Similarly, the time complexity of the DFS algorithm is also $O(b^d)$, where b is the branching factor (number of possible moves from each state), and d is the depth of the search tree. In the Rabbit Leap Problem, the branching factor is 4 (1 step or 2 steps in either direction), while the depth varies depending on the specific instance of the problem.

Depth-First Search (DFS) is more space-efficient than Breadth-First Search (BFS) because it explores one path to its maximum depth before backtracking, only needing to store the nodes along the current path. This leads to a space complexity of $O(bd)$, where b is the branching factor and d is the maximum depth of the tree. In contrast, BFS explores all nodes at each depth level before proceeding, requiring it to store all nodes at a given depth, resulting in a space complexity of $O(b^d)$, where d is the depth of the shallowest solution. In large or wide search spaces, BFS can quickly consume exponentially more memory, making DFS far more space-efficient when memory resources are limited. For the Rabbit Leap problem, the state space is quite structured, meaning that DFS might find the goal faster than BFS due to its depth-first nature. Since BFS explores all nodes at the same depth level before moving deeper, it tends to visit more nodes, especially if the solution is located at a deeper level in the tree. DFS, on the other hand, can reach the solution sooner if it explores the right branches first.

III. Week 2: State Space Search

A. *Introduction*

This lab report consists of graph search algorithms to solve two distinct problems: plagiarism detection in text documents and the 8-puzzle. The goal is to design a graph search agent using techniques like A* and Iterative Deepening Search. By exploring these algorithms, we analyze their effectiveness in navigating problem spaces and their implications for memory and time efficiency. These introductions outline two AI problems: the 8-puzzle and plagiarism detection. The 8-puzzle involves rearranging tiles on a 3x3 grid, solved using state-space search algorithms like A* to find the optimal move sequence. Plagiarism detection compares documents to identify similar text, implementing A* search to align sentences by minimizing edit distance. Both problems demonstrate the application of search algorithms, particularly A*, to optimize solutions in distinct domains - one for puzzle-solving and the other for text analysis.

B. *Puzzle-8*

*1) **Problem Statement**:*

1) Write a pseudocode for a graph search agent. Represent the agent in the form of a flow chart. Clearly mention all the implementation details with reasons.
2) Write a collection of functions imitating the environment for Puzzle-8.
3) Describe what is Iterative Deepening Search.
4) Considering the cost associated with every move to be the same (uniform cost), write a function which can backtrack and produce the path taken to reach the goal state from the source/ initial state.
5) Generate Puzzle-8 instances with the goal state at depth "d".
6) Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth "d") using your graph search agent.

*2) **Solution**:*

1) The pseudocode for the following problem can be accessed through this link and the flowchart of the following is:
2) Collection of functions are:
    - **Initial State:** Any state can be the initial state where the configuration of blocks is stored in a 3×3 matrix.
    - **Actions:** Instead of moving the numbered blocks, we can consider the action as the blank space moving up, down, right, and left.

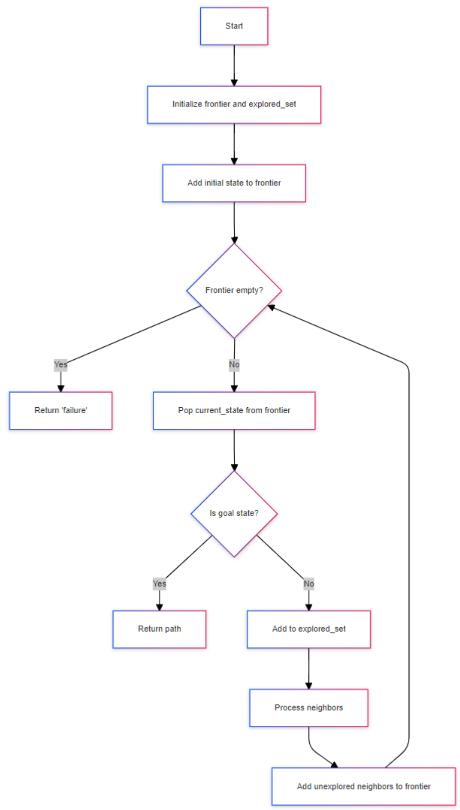

Figure 1: Flowchart for the Graph Search Agent

- **Goal State:** Although any state could be the goal, the correct configuration of blocks (the expected final state) is considered the goal state.
- **Goal Test:** Checking whether the new state is the same as the goal state or not.
- **Transitions:** Switching the positions of moved and empty blocks, resulting in a new state.
- **Path Cost:** Every step will have a cost, which will be added to the total cost.

3) Iterative deepening search applies depth-limited search repeatedly, increasing the depth limit each time. It has three possible outcomes per iteration: Finding a solution, Failing to find a solution within the current depth, Reaching a cutoff suggesting a deeper solution The method doesn't track visited states, potentially revisiting them. It starts with a depth limit of 0, incrementing until a solution is found. Time complexity is $O(b^d)$ when a solution exists at depth d, or $O(b^m)$ for no solution, where b is the branching factor and m is the maximum depth explored.
4) The pseudo-code for the following problem can be accessed through link
5) The pseudo-code for the following problem can be accessed through link
6) The table is as follows:

Depth	Nodes Explored	Time Elapsed (s)	Memory Used (bytes)
1	2.00	0.0000	0
2	2.60	0.0000	819
3	2.80	0.0000	0
4	3.00	0.0003	0
5	2.00	0.0000	0
6	3.20	0.0000	0
7	3.60	0.0002	0
8	7.80	0.0000	2458
9	4.40	0.0000	0
10	3.80	0.0002	0
11	6.60	0.0000	0
12	6.40	0.0002	0
13	7.00	0.0000	0
14	29.00	0.0002	12288
15	6.20	0.0002	0
16	7.40	0.0000	0
17	14.00	0.0002	0
18	12.40	0.0002	0
19	9.00	0.0000	0
20	31.40	0.0002	0

Figure 2: Table indicating memory and time requirements

C. *Plagiarism Detection System*

1) ***Problem Statement:***: Develop a plagiarism detection system that identifies similar or identical sequences of text between two documents by aligning their sentences or paragraphs. The system will leverage the A* search algorithm to efficiently find optimal text alignments based on edit distance, indicating potential instances of plagiarism.

2) ***Levenshtein Distance:*** Levenshtein distance is a similarity measure between two strings, representing the minimum number of edit operations (insertions, deletions, or substitutions) required to transform one string into another. For example, changing the word "test" into "tent" requires one substitution, so the Levenshtein distance is 1.
In the context of this lab, the Levenshtein distance is used to compute the cost of aligning sentences from two documents, which is key to detecting plagiarism. By aligning text based on minimal edit distance, the system identifies similar or identical sequences of text.

Example of Levenshtein Distance

Consider two words: "GUMBO" and "GAMBOL". The Levenshtein distance is 2 because transforming "GUMBO" into "GAMBOL" requires one substitution ("U" to "A") and one insertion ("L"). This concept scales to entire sentences in plagiarism detection, where the number of edits required to transform one sentence into another helps identify potentially copied content.

		G	U	M	B	O
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	1	2	3	4
M	3	2	2	1	2	3
B	4	3	3	2	1	2
O	5	4	4	3	2	1
L	6	5	5	4	3	2

Figure 3: Levenshtein Distance Matrix

3) ***Solution:*** The code for this problem statement can be accessed through this link.
Here's an overview of the steps taken to arrive at the solution:

1) Text Preprocessing: A 'preprocess_text' function was created that:
   - Splits the input text into sentences using regular expressions.
   - Converts each sentence to lowercase.
   - Removes leading and trailing whitespace from each sentence.
2) Similarity Calculation: To calculate similarity between sentences, the Levenshtein distance was utilized:
   - A 'levenshtein_distance' function was implemented to compute the edit distance between two strings.
   - A 'calculate_similarity' function was developed that normalizes the Levenshtein distance to a percentage similarity.
3) Plagiarism Detection: The 'detect_plagiarism' function was created to tie everything together:
   - Finds the best pairs of sentences between the documents using A* algorithm to align sentences in both documents.
   - Calculates an overall similarity percentage for the entire documents.

The Levenshtein distance between two strings s_1 and s_2 is given by the following recurrence relation:

$$lev_{s_1,s_2}(i,j) = \begin{cases} \max(i,j) \\ \min \begin{cases} lev_{s_1,s_2}(i-1,j)+1 \\ lev_{s_1,s_2}(i,j-1)+1 \\ lev_{s_1,s_2}(i-1,j-1)+1_{(s_1[i]\neq s_2[j])} \end{cases} \end{cases}$$

where $1_{(s_1[i]\neq s_2[j])}$ is the indicator function equal to 0 when $s_1[i]=s_2[j]$ and equal to 1 otherwise.

The percentage similarity between two strings can be calculated using the following formula:

$$\text{similarity}(s1,s2) = \left(1 - \frac{\text{levenshtein\_distance}(s1,s2)}{\max(|s1|,|s2|)}\right) \times 100$$

Where: - levenshtein_distance$(s1, s2)$ is the Levenshtein distance between the two strings $s1$ and $s2$. - $|s1|$ and $|s2|$ are the lengths of the strings $s1$ and $s2$ respectively.

*4) **A* Algorithm for document alignment**:* **Input:** Two documents doc1 and doc2.
Output: Optimal alignment between sentences

1) **Initialization:**
   - Create the initial state with pos1 = 0, pos2 = 0, cost = 0, and an empty path.
   - Initialize a priority queue (min-heap) with the initial state.
   - Initialize an empty set to keep track of visited positions.
2) **Main Loop:**
   - While the priority queue is not empty:
     – Pop the state with the lowest cost from the queue.
     – If both pos1 = |doc1| and pos2 = |doc2|, return the alignment path (goal reached).
     – If (pos1, pos2) has already been visited, continue to the next iteration.
     – Mark (pos1, pos2) as visited.
3) **State Expansion:**
   - If both pos1 < |doc1| and pos2 < |doc2|
     – Compute the cost using the Levenshtein distance between doc1[pos1] and doc2[pos2].
     – Create a new state advancing both pos1 and pos2, update the total cost, and append the alignment to the path.
     – Push the new state onto the priority queue with priority equal to the cost plus a heuristic.
   - If pos1 < |doc1| (skip sentence in doc2):
     – Create a new state advancing pos1 only, with an additional cost of 1, and push it onto the queue.
   - If pos2 < |doc2| (skip sentence in doc1):
     – Create a new state advancing pos2 only, with an additional cost of 1, and push it onto the queue.
4) **Heuristic:**
   - Use a heuristic function to estimate the remaining alignment cost by taking the minimum of the remaining sentences between doc1 and doc2:

$$h(\text{state}) = \min(|\text{doc1}| - \text{pos1}, |\text{doc2}| - \text{pos2})$$

*5) **Result of Test-Cases**:* The result of test cases can be seen below:

```
Test Case: Identical Documents
Document 1: This is a test. It has multiple sentences. We want to detect plagiarism.
Document 2: This is a test. It has multiple sentences. We want to detect plagiarism.
Detected 3 plagiarism cases:
  Sentence 1 in doc1 matches sentence 1 in doc2 with 100.00% similarity
  Sentence 2 in doc1 matches sentence 2 in doc2 with 100.00% similarity
  Sentence 3 in doc1 matches sentence 3 in doc2 with 100.00% similarity
Overall similarity: 100.00%
Verdict: Surely copied.

Test Case: Slightly Modified Document
Document 1: This is a test. It has multiple sentences. We want to detect plagiarism.
Document 2: This is an exam. It contains several phrases. We aim to identify copying.
Detected 0 plagiarism cases:
Overall similarity: 46.23%
Verdict: Possibly copied.
```

```
Test Case: Completely Different Documents
Document 1: This is about cats. Cats are furry animals. They make good pets.
Document 2: Python is a programming language. It is widely used in data science.
Detected 0 plagiarism cases:
Overall similarity: 19.93%
Verdict: Not copied.

Test Case: Partial Overlap
Document 1: This is a test. It has multiple sentences. We want to detect plagiarism.
Document 2: This is different. We want to detect plagiarism. This is unique.
Detected 1 plagiarism cases:
  Sentence 3 in doc1 matches sentence 2 in doc2 with 100.00% similarity
Overall similarity: 65.24%
Verdict: Likely copied.
```

Figure 4: Test Cases Verdict

IV. Week 3: Heuristic And Non-Classical Search

A. *Introduction*

Marble solitaire and **Boolean satisfiability (SAT)** problems are complex NP-hard challenges in computer science. Marble solitaire involves board-based marble removal, while **k-SAT problems** focus on satisfying Boolean variable assignments across multiple clauses. This experiment explores non-deterministic search methods with heuristics to find optimal or near-optimal solutions. These techniques have real-world applications in game AI, network design optimization, and task scheduling.

B. Non-deterministic Search

In the context of marble solitaire, non-deterministic search allows us to explore various sequences of moves that could lead to the desired configuration. Similarly, when tackling k-SAT problems, search algorithms can investigate different assignments of Boolean variables to find a combination that satisfies all clauses. The element of randomness in the search helps to prevent the algorithm from becoming trapped in local optima, enabling it to uncover more promising areas within the search space.

C. Marble Solitaire Problem

Marble solitaire is a classic puzzle where the objective is to remove all but one marble, ideally leaving the last marble in the center of the board. The game is played by jumping one marble over another into an empty hole, effectively removing the jumped marble from the board. This process continues until only one marble remains.

To solve marble solitaire, we employ the following strategies:

1) Implement a priority queue-based search considering path cost.
2) Propose two distinct heuristic functions, with justifications.
3) Implement the Best First Search algorithm.
4) Implement the A* algorithm.
5) Compare the results of these various search algorithms.

1) Algorithms for Marble Solitaire: **1. Priority Queue-Based Search**

Input: Initial board configuration and a set of valid moves.
Output: Sequence of moves leading to the goal state.
Algorithm:

Algorithm 3 Priority Queue Search

```
Initialize a priority queue
Add the initial state to the queue with priority 0
while the queue is not empty do
    Pop the state with the highest priority
    if the goal state is reached then
        Return the path to the goal
    end if
    for each valid move do
        Generate a new state
        Calculate the path cost
        Add the new state to the queue with the calculated
priority
    end for
end while
Return failure
```

Pros: This method guarantees that the first solution found is optimal, as it explores the most promising nodes first.
Cons: The priority queue can grow large, leading to high memory usage and longer search times, especially as the state space expands.

2. Heuristic Functions: We propose two heuristic functions to enhance our search:

- **Manhattan Distance Heuristic:** This measures how far each marble is from the center, providing a straightforward metric for guiding the search. By summing the distances of all marbles from their target positions, we gain insight into the potential moves needed to reach the goal.
- **Exponential Distance Heuristic:**This is similar, but with a larger bias towards the center. If H and V are the horizontal and vertical distances from the center respectively, then the heuristic's value is $2\hat{(}\max(H, V))$.

Pros of Heuristic Functions: Heuristics can significantly reduce the search space, leading to faster solutions.
Cons: Poorly designed heuristics may misguide the search, resulting in longer solution times.

3. Best First Search: **Input:** Initial board configuration and a set of valid moves.
Output: Optimal sequence of moves leading to the goal state.
Algorithm:

Algorithm 4 Best First Search

```
Initialize an empty priority queue
Add the initial state with cost using the heuristic
while the queue is not empty do
    Dequeue the state with the lowest cost
    if the goal state is reached then
        Return the path to the goal
    end if
    for each valid move do
        Generate a new state
        Calculate the cost using the heuristic
        Enqueue the new state with the updated cost
    end for
end while
Return failure
```

Pros: This approach efficiently finds solutions by employing an informed search strategy that takes the estimated cost into account.
Cons: It may be slower than other methods if the heuristic is poorly tuned, as it might explore suboptimal paths.

4. A* Search Algorithm: **Input:** Initial board configuration and a set of valid moves.
Output: Optimal sequence of moves leading to the goal state.
Algorithm:

Algorithm 5 A* Search Algorithm

```
Initialize an empty priority queue
Add the initial state with cost 0
while the queue is not empty do
    Dequeue the state with the lowest f(n) = g(n) + h(n)
    if the goal state is reached then
        Return the path to the goal
    end if
    for each valid move do
        Generate a new state
        Calculate g(n) and h(n)
        Enqueue the new state with f(n)
    end for
end while
Return failure
```

Pros: A* is highly efficient because it combines the cost to reach a node with the estimated cost to the goal, leading to a more directed search.
Cons: It requires more memory compared to simpler algorithms, as it keeps track of all nodes in the search space.

D. k-SAT Problem

The k-SAT problem is a well-known NP-complete problem that involves determining if a Boolean formula in conjunctive normal form (CNF) is satisfiable. In this experiment, we focus on the random generation of 3-SAT problems and solving them using various search algorithms.

A. Generating k-SAT Problems: To generate random k-SAT problems, we follow these steps:

1) Choose a number of variables n.
2) Choose a number of clauses m.
3) Randomly generate m clauses where each clause contains 3 literals, selecting from the set of variables and their negations.

Input: Number of variables n, number of clauses m.
Output: A random k-SAT problem in CNF.
Algorithm:

Algorithm 6 Generate Random 3-SAT Problem

```
Initialize an empty list of clauses
for i = 1 to m do
    Create a clause with 3 random literals
    Add the clause to the list
end for
Return the list of clauses
```

B. Solving k-SAT Problems: When it comes to solving k-SAT problems, we employ three algorithms:

1) Hill-Climbing Algorithm
2) Beam Search Algorithm
3) Variable-Neighborhood-Descent Algorithm

1) ***1. Hill-Climbing Algorithm:*** **Input:** A random 3-SAT problem.
Output: A satisfying assignment or failure.
Algorithm:

Algorithm 7 Hill-Climbing Algorithm

```
Initialize a random assignment of variables
while not satisfied do
    if current assignment satisfies the formula then
        Return current assignment
    end if
    Select a variable to flip
    Generate new assignment
    if new assignment is better then
        Update current assignment
    end if
end while
Return failure
```

Pros: The hill-climbing algorithm is straightforward and easy to grasp, making it a solid choice for those new to solving SAT problems.
Cons: However, it often gets stuck in local optima, which means it may miss the global solution.

2. Beam Search Algorithm: **Input:** A random 3-SAT problem.
Output: A satisfying assignment or failure.
Algorithm:

Algorithm 8 Beam Search Algorithm

```
Initialize beam with random assignments
while not satisfied do
    Expand each assignment in the beam
    Evaluate all new assignments
    Keep the best w assignments in the beam
end while
Return best assignment or failure
```

Pros: Beam search effectively narrows down the search space by maintaining only the best candidates, which can enhance performance.
Cons: The effectiveness of the solution is highly dependent on the beam width; if the beam is too narrow, it may overlook optimal solutions.

3. Variable-Neighborhood-Descent Algorithm: **Input:** A random 3-SAT problem.
Output: A satisfying assignment or failure.
Algorithm:

Algorithm 9 Variable-Neighborhood-Descent Algorithm

```
Initialize random assignment of variables
while not satisfied do
    for each neighborhood function do
        Generate new assignment
        if new assignment satisfies the formula then
            Return new assignment
        end if
    end for
    Update current assignment based on the best found
end while
Return failure
```

Pros: This algorithm expands the search space systematically by exploring various neighborhoods, which helps escape local optima.
Cons: It may require more iterations than simpler methods, potentially affecting overall efficiency.

Comparative Analysis: After solving the uniform random 3-SAT problems using the three different algorithms, we compare their performance based on:

- The number of iterations to reach a solution.
- The quality of solutions found.
- Time complexity.

It's anticipated that the Beam Search and Variable-Neighborhood-Descent algorithms will outperform the Hill-Climbing algorithm in terms of solution quality, especially as the complexity of SAT instances increases. This comparative analysis highlights the practical effectiveness of various problem-solving approaches.

V. WEEK 4: NON-DETERMINISTIC SEARCH AND SIMULATED ANNEALING

A. *Introduction*

The Travelling Salesman Problem (TSP) can be visualized as an undirected weighted graph, where cities are the vertices and the paths between them are the edges, each with a distance or cost. The goal is to find the shortest Hamiltonian cycle, a route that visits every vertex exactly once and returns to the starting point, minimizing the total travel distance. TSP is usually modeled as a complete graph, where every pair of cities is connected by an edge. If no direct path exists between cities, an artificial edge with a large weight can be added to ensure the graph remains complete without affecting the optimal solution.

B. *NON-DETERMINISTIC SEARCH*

Non-deterministic search is a problem-solving approach that explores multiple potential solutions simultaneously, allowing algorithms to make random moves at each step. This flexibility enables the discovery of various solutions, particularly in complex scenarios like the Travelling Salesman Problem, where exhaustive searches for optimal solutions can be time-consuming. Unlike deterministic algorithms, which yield consistent outputs, non-deterministic algorithms can exhibit different behaviors across executions, investigating multiple routes to uncover a range of outcomes. This variability enhances their effectiveness in optimization tasks.

C. *TRAVELLING SALESMAN PROBLEM*

The Travelling Salesman Problem (TSP) is a well-known question in combinatorial optimization that asks: "Given a list of cities and the distances between each pair, what is the shortest route that visits each city exactly once and returns to the starting point?" As an NP-hard problem, the TSP is significant in both theoretical computer science and operations research, as it exemplifies the challenges of optimizing routes and logistics in various practical applications.

Mathematical Formulation:

$$\text{Minimize } Z = \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij}$$

Constraints:

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \quad (1 \le i \le n)$$

$$\sum_{i=1}^{n} x_{ji} = 1 \quad \forall j \quad (1 \le j \le n)$$

- Z: Represents the total distance (or cost) of the tour.
- d_{ij}: The distance (or cost) between city i and city j.
- x_{ij}: A binary variable that indicates whether the route travels from city i to city j (1 if yes, 0 if no).

D. *SIMULATED ANNEALING*

Simulated Annealing is a probabilistic optimization algorithm inspired by the annealing process in metallurgy, which allows materials to reach a low-energy crystalline state through controlled heating and cooling. By making random moves and accepting worse solutions with a certain probability—reflected in the change in evaluation value ΔE—the algorithm effectively balances exploration and exploitation in complex search spaces. As the algorithm progresses, the likelihood of accepting inferior moves decreases, mimicking the cooling process and enabling the search of both promising and sub-optimal regions, thereby facilitating the discovery of high-quality solutions in complex optimization problems across various fields, including operations research, engineering, and artificial intelligence.

Algorithm:

```
current ← INITIAL − STATE
T ← some large value
for t = 1 to ∞ do
    if termination criteria then
        break
    end if
    next ← a randomly selected successor of current
    ΔE ← VALUE(next) − VALUE(current)
    if ΔE > 0 then
        current ← next
    else
        current ← next only with probability 1/(1 + e^(−ΔE/T))
    end if
    T ← cooling − function(T)
end for
```

In this context, ΔE represents the energy difference, and we assess the probability of making a move based on this value within the conditional check. The variable T denotes the temperature, which starts at a high value and gradually decreases through a cooling function. Initially, all moves are equally likely due to this high temperature.

E. PROBLEM STATEMENTS

1) Given a list of at least twenty important tourist locations in Rajasthan, and assuming the cost of travel between two locations is proportional to the distance between them, use the simulated annealing algorithm to plan a cost-effective tour. The goal is to find a tour that visits each location exactly once and returns to the starting point, minimizing the total travel cost.

Problem Definition:

- **Input:** A graph representing cities as nodes, with the distances between them represented as weighted edges.
- **Output:** A route that goes to each city only one time and comes back to the starting city, while keeping the total travel cost (or distance) as low as possible.

Start-up:

1) Identify 20 significant tourist locations in Rajasthan to serve as nodes in the graph.
2) Set the initial temperature T to a high value (e.g., $T = 1000$).
3) Define a minimum temperature T_{min} to terminate the annealing process (e.g., $T_{\text{min}} = 10$ or $T_{\text{min}} = 102$).
4) Establish a cooling schedule by reducing the temperature by a fraction α after each iteration.
5) Generate an initial tour by creating a random permutation of the selected cities.
6) Calculate the cost (distance) of the initial tour using the distances between the identified cities.

Simulated Annealing

1) Repeat the following steps until the temperature reaches T_{min}:
   a) **Generate Neighbor:** Create a neighboring tour by perturbing the current tour. This may include:
      - Swapping two cities in the tour.
      - Reversing a sequence of cities in the tour.
      - Shifting a subset of cities within the tour.

   b) **Calculate Cost:** Determine the total distance of the new tour.

   c) **Acceptance Probability:** Compute the acceptance probability using the formula:

   $$\text{Acceptance Probability} = e^{-\frac{\Delta f}{T}},$$

   where Δf is the difference in cost between the new tour and the current tour (new cost - current cost), and T is the current temperature.

   d) **Accept or Reject Neighbor:** Accept the new tour based on the acceptance probability. Always accept the new tour if it has a lower cost than the current tour.

   e) **Update Tour:** If the new tour is accepted, update the current tour to the new tour and adjust the current cost accordingly.

   f) **Cooling:** Decrease the temperature following the established cooling schedule.
2) Track the best tour and its corresponding distance found throughout the iterations.

Solution:

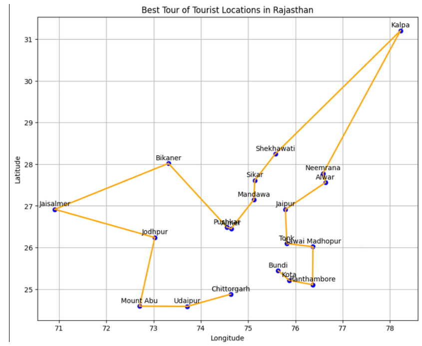

Figure 5: Best Tour Path.

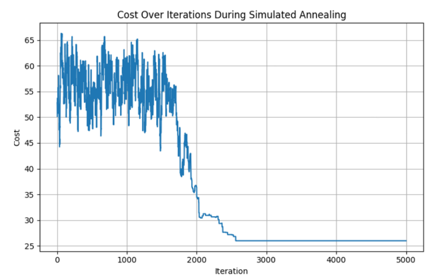

Figure 6: Cost vs Iteration

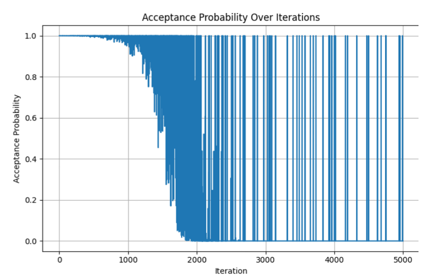

Figure 7: Acceptence Probability.

2) You need to solve a jigsaw puzzle represented as a 512x512 pixel image. The image is divided into smaller pieces and the objective is to rearrange these pieces to reconstruct the original image.

Problem Definition:

- **Input:** The input consists of a file named `scrambled_lena.mat`, which contains the scrambled image data represented as a matrix of pixel values. The image dimensions are 512x512 pixels, and it can be divided into smaller square pieces, such as 64 pieces of 128x128 pixels each. The initial configuration is represented by a random arrangement of the jigsaw pieces, serving as the starting state for the simulated annealing algorithm.
- **Output:** The desired output is the reconstructed image, which should reflect the solved configuration of the original Lena image. This reconstructed image will be properly arranged and can be saved or displayed as a 512x512 pixel image, representing the final state after the simulated annealing process has been completed.

Simulated Annealing in Puzzle Solving:

*1) **Initial State**:* The initial state consists of a random arrangement of the tiles. Each tile represents a 128×128 pixel block of the image. This scrambled configuration serves as the starting point for the algorithm.

*2) **Energy Calculation**:* The **energy function** quantifies how "disordered" the current arrangement of tiles is. This is computed based on the differences in pixel values between adjacent tiles:

- **Left-Right Energy:** Measures pixel differences along the vertical edges of tiles.
- **Up-Down Energy:** Measures pixel differences along the horizontal edges of tiles.

The total energy is the sum of energy contributions from all tiles, reflecting how well the tiles fit together.

*3) **Neighbor Generation**:* **Neighboring Solutions:** The algorithm generates neighboring configurations by swapping two tiles. This represents a small perturbation in the current arrangement of pixels. Each swap results in a new arrangement of the tiles, leading to a different configuration of the overall image.

*4) **Acceptance Probability**:* The algorithm calculates the change in energy (ΔE) after a swap. If the new configuration has lower energy (i.e., a better fit), it is accepted. If the new configuration has higher energy (worse fit), it may still be accepted with a certain probability, which decreases as the algorithm progresses and temperature lowers. This allows for exploration of less optimal configurations to escape local minima.

*5) **Cooling Schedule**:* The temperature T starts high, allowing for more random swaps and exploration of the solution space. Over time, the temperature decreases, reducing the likelihood of accepting worse configurations, and focusing more on refining the best found solution.

*6) **Final State**:* The process continues until the temperature reaches a predefined minimum (T_{min}), at which point the algorithm stops. The resulting arrangement of tiles represents the reconstructed image, ideally closely resembling the original.

Solution:

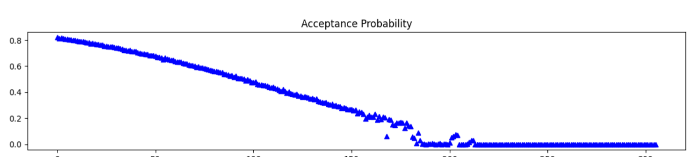

Figure 8: Acceptance Probability.

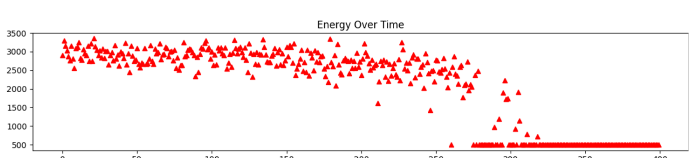

Figure 9: Energy Over Time.

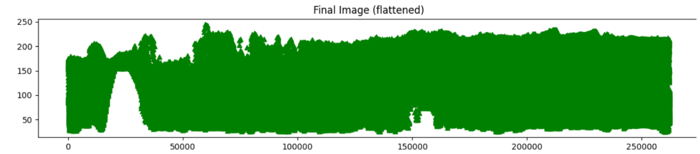

Figure 10: Image Flattened

Figure 11: Solved Puzzle

VI. Week 5: Graphical Models and Bayesian Networks Inference and Classification in R

A. *Introduction*

In this lab, we explore how to use graphical models and Bayesian inference to handle uncertainty in data. The focus is on educational data, where we will build Bayesian Networks using R to model the relationships between student grades in different courses. We also explore classification, specifically using the naive Bayes method, which is a simple but effective way to classify data based on probability. The dataset includes student grades and their qualification status for an internship. The goal is to train and evaluate the naive Bayes classifier, both by assuming the grades are independent and by considering the dependencies between them. This lab serves as an introduction to using probabilistic models for data analysis and classification.

B. *Fundamentals*

*1) **Bayesian Inference and Graphical Models**:* Graphical models use simple diagrams to show how different variables are related, which helps us understand uncertainty better. Bayesian inference is a method that lets us change our beliefs about unknown things when we get new information. Together, these ideas help us make sense of complex data and make better decisions.

*2) **Constructing Bayesian Networks in R**:* Bayesian Networks are types of graphical models that show how different variables are related using a directed acyclic graph (DAG), which is a diagram that connects the variables without any cycles. In R, there are packages like ‘bnlearn‘ that make it easy to build, visualize, and analyze these Bayesian Networks.

*3) **Learning Dependencies from Data**:* Bayesian Networks often rely on expert insights to determine how variables are related, but these relationships can also be learned from data. Methods like structure learning and parameter learning help uncover these connections and build Conditional Probability Tables (CPTs) from observed data. Participants will learn how to use data to identify the structure and parameters of Bayesian Networks, exploring algorithms such as constraint-based methods, score-based methods, and hybrid approaches to discover these relationships

*4) **Naive Bayes Classification**:* Naive Bayes is a straightforward probabilistic model that applies Bayes' theorem while assuming that the features are independent of one another. Although it is a basic method, it is commonly employed in applications such as document classification, sentiment analysis, and email filtering.

Students will comprehend the underlying principles of naive Bayes classification, including the calculation of class probabilities using Bayes' theorem and the assumption of feature independence.

Bayes Theorem:

$$P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}$$

Naive Bayes Classification Formula:

$$P(C|X) \propto P(C) \cdot P(x_1|C) \cdot P(x_2|C) \cdot \ldots \cdot P(x_n|C)$$

Where:

- $P(C|X)$ is the **posterior probability**: the probability of class C given the features X.
- $P(X|C)$ is the **likelihood**: the probability of the features X given the class C.
- $P(C)$ is the **prior probability** of the class C.
- $P(X)$ is the **marginal likelihood** or evidence: the total probability of the features X.

*5) **Classifier Implementation and Evaluation**:* Setting up a Naive Bayes classifier involves training the model on labeled data by calculating class priors (how often each outcome occurs) and conditional probabilities (how likely each feature is for each outcome). Once trained, the model can predict outcomes for new data based on these probabilities. To assess the model's performance, you use evaluation metrics like accuracy, precision, recall, and F1 score. Learners will gain practical experience using R to implement this classifier, covering data preparation, model training, prediction, and evaluation, ultimately applying these techniques to real-world educational data analysis tasks

C. *Problem Statement*

*1) **Problem I**:* The dataset *2020_bn_nb_data.txt* includes student grades from various courses. The objective is to model the relationships between these courses and learn the Conditional Probability Tables (CPTs). Furthermore, the aim is to predict a student's grade in PH100 based on their grades in other courses.

We will use the last column, which indicates internship qualification, to train a Naive Bayes classifier with 70% of the data. This classifier will predict if a student qualifies for an internship based on their grades, assuming that the courses are independent. We'll evaluate its accuracy on the remaining 30% of the data over 20 runs. Afterward, we will repeat the experiment, this time considering any potential relationships between the grades earned in different courses..

D. *Implementation Method*

1) **Data Preprocessing**:
   - Load the dataset (*2020_bn_nb_data.txt*) into R.
   - Analyze the data to gain insights into its structure and content.
   - Clean the data as needed by addressing any missing values or outliers.
2) **Bayesian Network Construction**:
   - Use the `bnlearn` package to construct a Bayesian Network.
   - Establish the network structure using domain knowledge or apply structure learning algorithms to derive it from the data.
   - Estimate Conditional Probability Tables (CPTs) for each node in the network.
3) **Grade Prediction in PH100**:
   - Given a student's grades in other courses, use the Bayesian Network to predict the grade in PH100.
   - Use the known grades to query the network and determine the most probable grade in PH100.
4) **Naive Bayes Classifier**:
   - Split the dataset into training and testing sets (70% training, 30% testing).
   - Train a naive Bayes classifier using the training data, assuming independence between course grades.
   - Evaluate the classifier's performance on the testing data using accuracy and other relevant metrics.
5) **Assessing Classifier Accuracy**:
   - Repeat the training and testing of the Naive Bayes classifier 20 times using randomly selected data.
   - Document the accuracy of the classifier during each iteration.
6) **Considering Dependencies**:
   - Adjust the Naive Bayes classifier to account for possible relationships between course grades.
   - Redo the training and testing process, evaluating the accuracy of the classifier based on this revised approach.

E. *Solution*

VII. Challenge Problem : Generating melody using simulated annealing

A. *Problem Statement*

Create a computer program to generate a melody in the style of North Indian Classical Raag Bhairav music. Use advanced algorithms (simulated annealing or genetic algorithms) to ensure the generated melody follows the traditional rules and patterns of Raag Bhairav, including key phrases typically used in this style of music

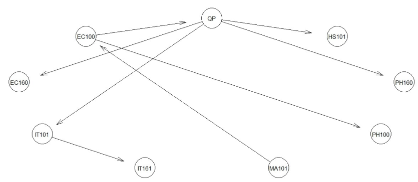

Figure 12: Dependencies

```
>
> library(bnlearn)
> library(caret)
Loading required package: ggplot2
Loading required package: lattice
> library(e1071)
>
> grades <- c("AA", "AB", "BB", "BC", "CC", "CD", "DD", "F")
>
> course.grades <- read.table("C:/Users/Abhi Patel/Downloads/2020_bn_nb_data.txt", header=TRUE)
>
> set.seed(100)
>
> tIndex <- createDataPartition(course.grades$QP, p=0.7, list=FALSE)
>
> train <- course.grades[tIndex, ]
> test <- course.grades[-tIndex, ]
>
> nbc <- naiveBayes(QP ~ EC100 + EC160 + IT101 + IT161 + MA101 + PH100 + PH160 + HS101, data=train)
>
> printALL <- function(model) {
+   trainPred <- predict(model, newdata = train, type = "class")
+   trainTable <- table(train$QP, trainPred)
+   trainAcc <- sum(diag(trainTable)) / sum(trainTable)
+
+   testPred <- predict(model, newdata = test, type = "class")
+   testTable <- table(test$QP, testPred)
+   testAcc <- sum(diag(testTable)) / sum(testTable)
+
+   message("Accuracy")
+   print(round(cbind("Training Accuracy" = trainAcc, "Test Accuracy" = testAcc), 4))
+ }
>
> printALL(nbc)
Accuracy
     Training Accuracy Test Accuracy
[1,]            0.9939        0.9855
>
```

Figure 13: Final Solution

B. Key Components

1) Sargam and Note Mapping:

- Sargam Notes: Sa, Re, Ga, Ma, Pa, Dha, Ni
- MIDI Note Mapping:

Sargam	MIDI Note
Sa	60
Re	62
Ga	64
Ma	65
Pa	67
Dha	69
Ni	71

2) Target Phrase:

```
Target Phrase: Sa Re Ga Pa Ma Pa Dha Ni Sa
```

3) Simulated Annealing:

- Mutation: Randomly changes a note in the melody.
- Acceptance Probability: Determines the likelihood of accepting a worse solution based on temperature.

4) MIDI File Generation:

- Uses a MIDI library (e.g., ‘mido‘) to create a MIDI file from the generated melody.

C. Simulated Annealing Algorithm in Melody Generation

1) Key Parameters:

- P_i: Initial phrase
- P_t: Target phrase
- T_i: Initial temperature
- T_min: Minimum temperature
- α: Cooling rate
- $S(P)$: Score of phrase P
- ΔS: Change in score

2) Iterative Process:

1) Temperature Update:
$$T = \max(T_i \cdot \alpha^i, T_min)$$
2) Mutation:
$$P_{new} = M(P_i)$$
3) Score Calculation:
$$\Delta S = S(P_{new}) - S(P_i)$$
4) Acceptance:
$$\text{Accept } P_{new} \text{ with probability } \min\left(1, \exp\left(\frac{\Delta S}{T}\right)\right)$$

3) Scoring Function:

$$S(P) = \sum_{i=1}^{n} 2 \cdot \delta(P_i, P_t[i]) + \delta(P_i \in P_t, 1)$$

where $\delta(x, y)$ is the Kronecker delta:

$$\delta(x, y) = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{otherwise} \end{cases}$$

VIII. Conclusion

Our AI Lab experience provided key insights into efficient AI techniques. We explored methods like state-space search, heuristic optimization, probabilistic reasoning, and machine learning, building a strong foundation in AI problem-solving. Through practical experiments and algorithm implementation, we learned to apply AI to complex, real-world problems. This experience prepared us for future challenges in the rapidly evolving field of artificial intelligence.

IX. References

- Russell, S., & Norvig, P. (2020). Artificial intelligence: A modern approach (4th ed.). Pearson
- Masrat, A., Gawde, H., Makki, M.A., & Parekh, U. (2021). Document Plagiarism Detection Tool using Edit Distance Text.
- Gardahadi, G. (2019). Plagiarism Detection Using Levenshtein Distance With Dynamic Programming.

X. Github Repository

For additional resources and detailed codes, please refer to our GitHub repository link.