Mid Semester Lab Report(September 2024)

Shreya Manyam (202211080), Vibhansh Goel (202211097) Battu Ujwal Reddy (202211010), Erramshetti Abhilash (202211022)

Abstract

This report investigates several algorithmic challenges in artificial intelligence, specifically the Rabbit Leap problem, k-SAT problem, plagiarism detection, and the Traveling Salesman Problem (TSP). The Rabbit Leap problem is modeled as a state space search, utilizing both Breadth-First Search (BFS) and Depth-First Search (DFS) to explore state transitions, with BFS proving to be more efficient for optimal solutions. The k-SAT problem is addressed through the generation of random expressions based on variable constraints, highlighting its combinatorial complexity. A novel plagiarism detection system employs the A* search algorithm to align sentences by minimizing edit distances, successfully identifying similar content. Finally, the TSP is approached with the Simulated Annealing (SA) algorithm, which yields near-optimal routes for traveling among cities in Rajasthan, India. Overall, the findings emphasize the significance of algorithm selection based on problem characteristics and contribute valuable insights into effective algorithmic design in AI.

Keywords: Artificial Intelligence, Breadth-First Search (BFS), Depth-First Search (DFS), A* Search Algorithm, Optimization.

I. Rabbit leap problem

The Rabbit Leap problem involves modeling the movements of two groups of rabbits—Easter bunnies and wild rabbits—across a linear arrangement. The objective is to transition from an initial state, where the groups are separated by an empty space, to a goal state where they have swapped positions.

This problem serves as a practical example for exploring algorithmic techniques like Breadth-First Search (BFS) and Depth-First Search (DFS). By employing these methods, we can systematically evaluate the potential states of the rabbits and efficiently determine a sequence of valid moves to achieve the desired configuration. This study highlights the interplay between simple rules and complex behaviors, making it a compelling subject for state space search in artificial intelligence.

A. Modeling the Problem as a State Space Search

The problem involves six rabbits—three moving eastward (denoted as E) and three moving westward (denoted as W). They are positioned on stones in the following way:

$$\text{Initial State: } EEE\_WWW$$

$$\text{Goal State: } WWW\_EEE$$

The empty stone (_) is crucial because rabbits can only move forward or jump over one rabbit. A valid state transition occurs when a rabbit moves or jumps to the empty stone.

1 State Representation

Each state is represented as a string of length 7 (since there are six rabbits and one empty spot). The initial state is `"EEE_WWW"`, and the goal state is `"WWW_EEE"`.

B. Size of the Search Space

There are 7 positions, and the rabbits are distinguishable by their directions. Therefore, the total number of configurations (i.e., states) is:

$$\frac{7!}{3! \cdot 3! \cdot 1!} = \frac{7!}{6} = 840$$

This gives us a search space of 840 possible states.

C. Solving the Problem Using BFS

Breadth-First Search (BFS) is ideal for finding the optimal solution.

```
from collections import deque

# Rabbit leap problem initial setup
initial_state = "EEE_WWW"
goal_state = "WWW_EEE"

def bfs_rabbit_leap():
     # (current_state, path)
    queue = deque([(initial_state, [])])
```

Vol. 1, **No.** 1, 2024.
http://dx.doi.org/

```
    visited = set([initial_state])

    while queue:
        current_state, path = queue.popleft()

        # If we reached the goal, return solution path
        if current_state == goal_state:
            return path + [goal_state]

        # Generate all possible valid moves
        for new_state in generate_moves(current_state):
            if new_state not in visited:
                visited.add(new_state)
                queue.append((new_state,path+[current_state]))

    return None

def generate_moves(state):
    # Find the index of the empty space
    empty_idx = state.index('_')
    rabbits = list(state)
    moves = []

    # E moves one step right
    if empty_idx - 1 >= 0 and rabbits[empty_idx - 1] == 'E':
        (new_state[empty_idx], new_state[empty_idx - 1]) =
        (new_state[empty_idx - 1], '_')

        moves.append(''.join(new_state))

    # E jumps over one rabbit
    if empty_idx - 2 >= 0 and rabbits[empty_idx - 2] == 'E':
        new_state = rabbits[:]
        (new_state[empty_idx], new_state[empty_idx - 2]) =
        (new_state[empty_idx - 2], '_')
        moves.append(''.join(new_state))

    # W moves one step left
    if empty_idx + 1 < 7 and rabbits[empty_idx + 1] == 'W':
        new_state = rabbits[:]
        (new_state[empty_idx], new_state[empty_idx + 1]) =
        (new_state[empty_idx + 1], '_') =
        (new_state[empty_idx + 1], '_')
        moves.append(''.join(new_state))

    # W jumps over one rabbit
    if empty_idx + 2 < 7 and rabbits[empty_idx + 2] == 'W':
        new_state = rabbits[:]
        (new_state[empty_idx], new_state[empty_idx + 2]) =
        (new_state[empty_idx + 2], '_')
        moves.append(''.join(new_state))

    return moves

# Run BFS
solution_bfs = bfs_rabbit_leap()
print("BFS Solution:", solution_bfs)
```

D. Solving the Problem Using DFS

Depth-First Search (DFS) explores as far as possible along each branch before backtracking. It is not guaranteed to find the optimal solution unless you explore all paths.

```
def dfs_rabbit_leap(current_state, path, visited):
    # If we reached the goal, return the solution path
    if current_state == goal_state:
        return path + [goal_state]

    visited.add(current_state)

    # Generate all possible valid moves
    for new_state in generate_moves(current_state):
        if new_state not in visited:
            result =
            dfs_rabbit_leap(new_state,
            path + [current_state], visited)
            if result:
                return result

    return None

# Run DFS
visited_dfs = set()
solution_dfs = dfs_rabbit_leap(initial_state, [], visited_dfs)
print("DFS Solution:", solution_dfs)
```

E. Comparison of BFS and DFS

1 BFS

- Guarantees finding the optimal solution with the fewest steps.
- Explores all nodes level by level.
- Time Complexity: $O(b^d)$, where b is the branching factor, and d is the depth of the optimal solution.
- Space Complexity: $O(b^d)$, because it stores all nodes in memory.

2 DFS

- May not find the optimal solution.
- Explores each branch to its deepest level before backtracking.
- Time Complexity: $O(b^m)$, where m is the maximum depth.
- Space Complexity: $O(b \cdot m)$, since it only needs to store the current path.

II. k-SAT problem

The k-SAT problem is a special case of boolean satisfiability problem(SAT) which is based on fundamental concepts of computer science and mathematical logical reasoning. SAT problem check for assignment of truth or boolean value to variable which will make the formula or expression to be true or 1(given the formula or expression). k-SAT is a SAT problem where each clause has exactly k literals(variable or its negation) joined by disjunction(logical OR) and each clause joined by conjuctor(logical AND), m is the number of clauses and n is number of variables.

A. Constrain on value of k due to n

k: number of literals in a clause

n: number of variables

Each clause can contain exactly k literals and we have only n variables to choose from. A literal can be either a variable or its negation so we can have k always less than equal to n otherwise we will use same variable twice in a clause.

B. Modeling the Problem as a State Space Search

Given the expression to be checked or evaluated using k-SAT, we can assign True/False value to each variable in the expression. We can define states as partial or complete assignment of truth values to the variables in the formula in k-SAT problem and assigning value to new variable as action to go from one state to another.

1 State Representation

Let

$$x_1, x_2, x_3 ..... x_n$$

be the Boolean variables in the formula.

Each state can be represented as a tuple showing truth value assigned to some or all variables. Example:

$$(x_1 = True, x_2 = False)$$

$$(x_1 = False, x_2 = True, ...x_n = True)$$

2 Initial State Representation

Initial state will be represented as an empty tuple.

$$S_0 = ()$$

3 Final State Representation

Final state will be represented as tuple having a truth value assigned to each variable.

4 Action

Assigning a truth value to an unassigned variable.

5 Transition function

Transition function will take a state and action as input and gives the state as output which will be reached after performing the input action on input state. Eg:

State A:

$$(x_1 = True)$$

Action: Assign False to x_2

State B:

$$(x_1 = True, x_2 = False)$$

C. Size of the Search Space

m: number of clauses

k: number of literals

n: number of variables

To choose k literal from n variables we have $\binom{2n}{k}$ ways beacause each literal can be negated or not so each clause can be choosen from a set of size $\binom{2n}{k}$. Choosing m clauses from this set will result in $\binom{\binom{2n}{k}}{m}$ possible expressions for fixed k,m,n.

D. Program to generate random k-SAT problem expressions with given k,m,n

This program accepts the value of k(literals in a clause), m(number of clauses) and n(number of variable to choose from) and gives 5 random expression with m clauses each having k literals choosen from n variables or their negations.

```
from string import ascii_lowercase,ascii_uppercase
from random import sample
from itertools import combinations

m = int(input("Enter the number of clauses: "))
n = int(input("Enter number of variables: "))
k = int(input("Enter the number of variables in a clause: "))
N = int(input("Enter the number of such expression to generate:
"))

def createProblem(m, k, n,N):
    positive_var = (list(ascii_lowercase))[0:n]
    negative_var = (list(ascii_uppercase))[0:n]
    variables = positive_var + negative_var
    problems = []
    all = list(combinations(variables, k))

    for i in range(N):
        s = sample(all, m)
        if s not in problems:
            problems.append(list(s))
    return problems

problems = createProblem(m, k, n,N)

for i in range(len(problems)):
    print(problems[i])
```

III. Plagiarism Detection

Plagiarism detection is a critical task in educational and research environments. With the increasing availability of digital content, the need for an automated system to detect text similarities has grown. This paper introduces a novel system that uses the A* search algorithm to align sentences between two documents, minimizing the total edit distance between them. The alignment helps in detecting plagiarism by identifying sentences that are highly similar based on their edit distances.

The primary objective of this work is to align sentences optimally using A*, which ensures that the total similarity cost (edit distance) between two documents is minimized. This allows the system to efficiently detect sentence-level plagiarism.

A. Problem Definition

Given two documents, the task is to align their sentences using the A* search algorithm, minimizing the edit distance. The system flags sentences with low edit distances, which indicates high similarity and potential plagiarism.

B. Text Preprocessing

The input documents are first preprocessed to ensure consistency. This involves converting the text to lowercase, removing punctuation, and tokenizing the text into sentences. This step is crucial for ensuring that the A* algorithm works with normalized content.

C. A* Search Algorithm for Sentence Alignment

- **State Representation**: Each state in the A* search represents a partial alignment of sentences between two documents. Each state holds the index positions of the aligned sentences and the accumulated edit distance (cost).
- **Transition Function**: The system can transition between three possible states:
  1. Align the current sentence from Document 1 with the current sentence from Document 2.
  2. Skip a sentence from Document 1 or Document 2.
- **Cost Function ($\mathbf{g(n)}$)**: The cost is calculated as the accumulated Levenshtein distance (edit distance) of the aligned sentences up to the current state.
- **Heuristic ($\mathbf{h(n)}$)**: The heuristic estimates the minimum possible alignment cost for the remaining sentences, guiding the A* search toward the optimal solution.

D. Levenshtein Distance

The Levenshtein distance, or edit distance, is used to measure the similarity between two sentences by counting the number of insertions, deletions, and substitutions required to transform one sentence into another. A lower edit distance indicates higher similarity between the sentences.

E. Plagiarism Detection Approach

After aligning sentences, those with a low edit distance are flagged as potential plagiarism cases. The threshold is set such that sentences with an edit distance less than 50% of their length are considered similar.

F. Test Cases and Results

The system was evaluated on several test cases:

1 Test Case 1: Identical Documents

Input: Document 1: "The quick brown fox jumps over the lazy dog. It swiftly runs across the yard. Testing examples are essential for validation."
Document 2: "The quick brown fox jumps over the lazy dog. It swiftly runs across the yard. Testing examples are essential for validation."
Expected Output: All sentences should align perfectly with zero edit distance, indicating no plagiarism.

Output:

```
Comparing: 'the quick brown fox jumps over the lazy dog'
with 'the quick brown fox jumps over the lazy dog',
Edit Distance: 0
Comparing: 'it swiftly runs across the yard'
with 'it swiftly runs across the yard',
Edit Distance: 0
Comparing: 'testing examples are essential for validation'
with 'testing examples are essential for validation',
Edit Distance: 0
[('the quick brown fox jumps over the lazy dog',
'the quick brown fox jumps over the lazy dog'),
('it swiftly runs across the yard',
'it swiftly runs across the yard'),
('testing examples are essential for validation',
'testing examples are essential for validation')]
```

2 Test Case 2: Slightly Modified Document

Input: Document 1: "The quick brown fox jumps over the lazy dog. It swiftly runs across the yard. Testing examples are essential for validation."
Document 2: "A fast brown fox leaps over a lazy dog. It quickly moves through the field. Example tests are important for verification."
Expected Output: Most sentences align with low edit distances, indicating potential plagiarism.

Output:

```
Comparing: 'it swiftly runs across the yard'
with 'a fast brown fox leaps over a lazy dog',
Edit Distance: 31
Comparing: 'it swiftly runs across the yard'
with 'it quickly moves through the field',
Edit Distance: 17
Comparing: 'testing examples are essential for validation'
with 'it quickly moves through the field',
Edit Distance: 35
Comparing: 'testing examples are essential for validation'
with 'example tests are important for verification',
```

```
Edit Distance: 26
[]
```

3 Test Case 3: Completely Different Documents

Input: Document 1: "The quick brown fox jumps over the lazy dog. It swiftly runs across the yard. Testing examples are essential for validation."
Document 2: "This document is completely unrelated. There are no similarities here."
Expected Output: High edit distances, indicating no similarity or plagiarism.

Output:

```
Comparing: 'it swiftly runs across the yard'
with 'this document is completely unrelated',
Edit Distance: 30
Comparing: 'testing examples are essential for validation'
with 'this document is completely unrelated',
Edit Distance: 35
Comparing: 'testing examples are essential for validation'
with 'there are no similarities here',
Edit Distance: 34
[]
```

4 Test Case 4: Partial Overlap

Input: Document 1: "The quick brown fox jumps over the lazy dog. It swiftly runs across the yard. Testing examples are essential for validation."
Document 2: "The quick brown fox jumps over the lazy dog. Another random sentence. Testing examples are essential for validation."
Expected Output: Low edit distances for overlapping content, indicating potential plagiarism.

Output:

```
Comparing: 'the quick brown fox jumps over the lazy dog'
with 'the quick brown fox jumps over the lazy dog',
Edit Distance: 0
Comparing: 'it swiftly runs across the yard'
with 'the quick brown fox jumps over the lazy dog',
Edit Distance: 31
Comparing: 'it swiftly runs across the yard'
with 'another random sentence',
Edit Distance: 25
Comparing: 'testing examples are essential for validation'
with 'testing examples are essential for validation',
Edit Distance: 0
[('the quick brown fox jumps over the lazy dog',
'the quick brown fox jumps over the lazy dog'),
('testing examples are essential for validation',
'testing examples are essential for validation')]
```

G. Conclusion

In this paper, we developed a system for detecting plagiarism using the A* search algorithm to align sentences from two documents. By minimizing the edit distance between sentences, the system can efficiently detect plagiarism. The A* search ensures optimal alignment and can handle both similar and dissimilar documents. The test cases demonstrate the effectiveness of the system in identifying potential plagiarism across various document types.

IV. Traveling Salesman Problem

The Traveling Salesman Problem (TSP) involves finding the shortest route that visits each city exactly once and returns to the starting point. Given the problem's NP-hard nature, solving it efficiently for a large number of cities is computationally infeasible using brute-force techniques.

This section explores the use of the Simulated Annealing (SA) algorithm to solve the TSP for 20 cities in Rajasthan, India. Simulated Annealing is a probabilistic algorithm that mimics the physical process of annealing, where a material is heated and then cooled to remove defects, enabling the discovery of a low-energy state.

A. Problem Definition

The cities included in this problem are major tourist and historical destinations in Rajasthan. The goal is to compute the shortest route that connects the following cities:

- Jaipur, Udaipur, Jodhpur, Jaisalmer, Ajmer, Pushkar
- Bikaner, Mount Abu, Chittorgarh, Alwar, Ranthambore, Bharatpur
- Kota, Bundi, Shekhawati, Kumbhalgarh, Neemrana, Barmer, Pali, Sikar

Each city is represented by its geographical coordinates (latitude and longitude), and the objective is to minimize the total distance traveled while visiting all cities exactly once and returning to the starting city.

B. Simulated Annealing Approach for TSP

Simulated Annealing (SA) is a stochastic optimization technique that works by iteratively exploring the solution space while probabilistically accepting suboptimal solutions to escape local optima.

Key features of the SA algorithm:

- **Temperature** (T): The algorithm starts at a high temperature, which allows more freedom in accepting worse solutions. As the temperature cools, the algorithm becomes more conservative in accepting worse solutions.

- **Cooling Schedule**: The temperature is gradually reduced based on a cooling rate. In this report, we use an exponential cooling schedule.
- **Acceptance Probability**: If the new solution is better, it is always accepted. If it is worse, it is accepted with a probability proportional to the current temperature, allowing the algorithm to explore a larger solution space.

C. Results for TSP

The algorithm was applied with the following parameters:

- **Initial Temperature**: 10,000
- **Cooling Rate**: 0.995
- **Stopping Temperature**: 0.001

The SA algorithm successfully converged to a near-optimal solution, providing a tour with a minimized total travel distance.

D. Best Tour

The best tour found by the SA algorithm is as follows:

> Jaipur → Ranthambore → Bharatpur → Alwar → Neemrana → Sikar → Shekhawati → Pushkar → Ajmer → Udaipur → Kumbhalgarh → Pali → Jodhpur → Bikaner → Jaisalmer → Barmer → Mount Abu → Chittorgarh → Kota → Bundi

E. Tour Cost

The total travel distance for the best tour is approximately:

> **24.13 distance units (Euclidean distance based on latitude/longitude).**

F. Visual Representation

The following figure shows the best tour path found by the Simulated Annealing algorithm, along with the distances between cities.

V. Scrambled Image Puzzle

We are provided with a scrambled version of the well-known image of **Lena**. The original image is scrambled and stored in a '.mat' file format. The objective of this problem is to reorder the pieces of the scrambled image to form the original image using **Simulated Annealing**.

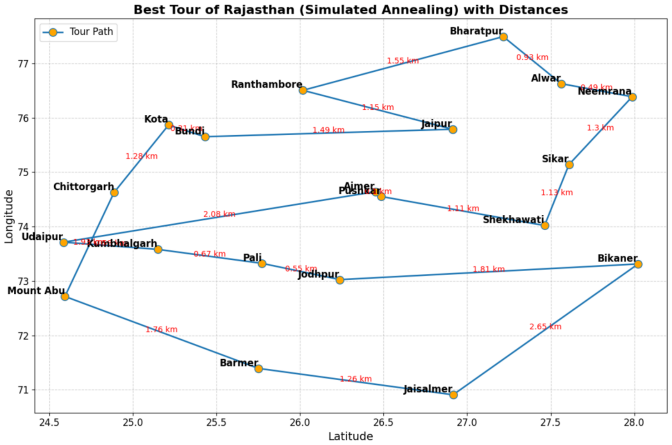

Figure 1. Best Tour of Rajasthan Cities using Simulated Annealing

The process involves the following steps: 1. Loading the scrambled image from the '.mat' file. 2. Dividing the image into smaller pieces (e.g., 4x4 grid). 3. Scrambling the pieces by shuffling their order. 4. Reassembling the scrambled pieces using **Simulated Annealing**, which minimizes the pixel mismatch between adjacent pieces.

A. Simulated Annealing Approach for Image Puzzle

The proposed approach for solving the image puzzle is based on the **Simulated Annealing (SA)** optimization algorithm. The key steps in this approach are:

1. **Image Loading and Preprocessing**: The scrambled image is loaded from the provided '.mat' file. It is then preprocessed by rotating and flipping the image to correct any distortions.

2. **Puzzle Setup**: The image is divided into smaller pieces. In this case, a 512x512 Lena image is divided into a 4x4 grid, resulting in 16 square pieces. Each piece is considered an independent element that can be moved and rotated.

3. **Simulated Annealing Algorithm**: Simulated Annealing is used to search for the optimal arrangement of the scrambled pieces. The SA algorithm mimics the process of physical annealing, where a material is heated and then slowly cooled to find a stable configuration with minimal energy.

- **Initial Configuration**: Start with a randomly shuffled arrangement of the image pieces. - **Successor Generation**: For each iteration, a neighboring configuration is generated either by swapping two pieces or by rotating a random piece. - **Cost Function**: The cost function is calculated by

comparing the boundaries (edges) of adjacent pieces. The pixel intensity differences between neighboring pieces are summed to compute the total cost. The objective is to minimize this cost. - **Acceptance Criterion**: If the new configuration has a lower cost, it is accepted. If it has a higher cost, it may still be accepted with a probability that decreases as the temperature decreases. - **Cooling Schedule**: The temperature is reduced gradually according to a cooling rate, which ensures that the algorithm eventually converges.

4. **Visualization**: The initial and final configurations of the puzzle are visualized to show the improvements achieved by the SA algorithm.

B. Results for Scrambled Image Puzzle

1. **Initial Scrambled Puzzle**: After loading the scrambled image from the '.mat' file, the image was divided into 16 pieces (4x4 grid). The pieces were then shuffled to simulate a scrambled puzzle.

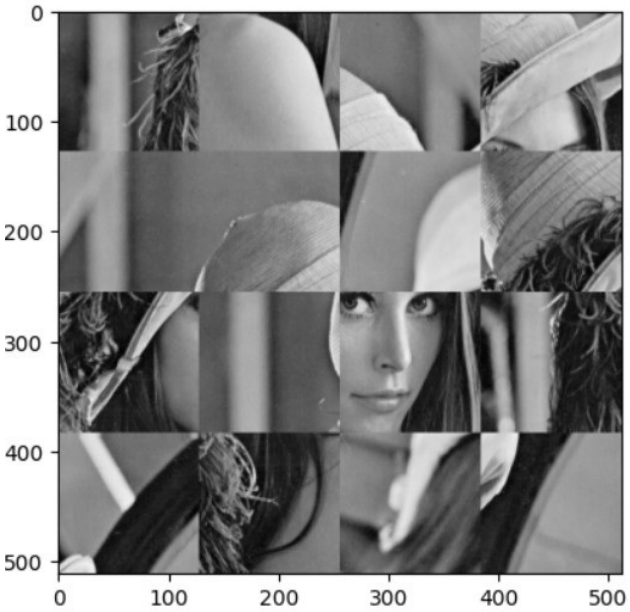

Figure 2. Initial Scrambled Image Puzzle

2. **Final Solved Puzzle**: After running the **Simulated Annealing** algorithm, the puzzle was reassembled into a near-optimal configuration.

- **Final Cost**: After the annealing process, the final cost was significantly lower, indicating that the pieces had been rearranged to better match their neighbors.

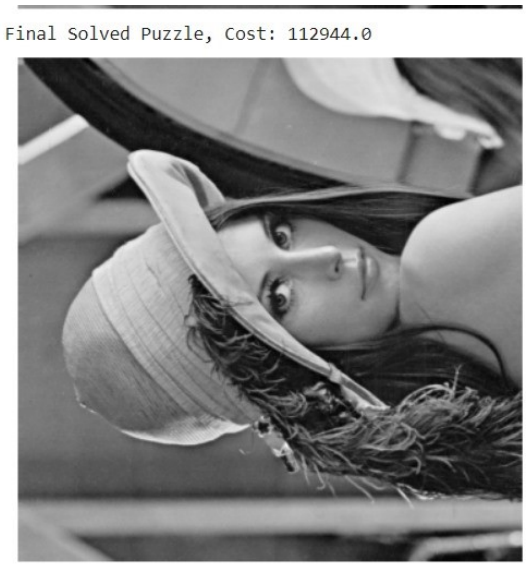

Figure 3. Final Solved Puzzle

References

[1] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed., Prentice Hall, 2010.

[2] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 3rd ed., Draft chapters available online, 2020.

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