Indian Institute of Information Technology,
Vadodara

CS307 - Artificial Intelligence
Laboratory Assignment Reports

Abstract

This Lab submission contains the description for in-lab Discussion and Submission problems of CS307 lab manual. The Report is structured in following way: Submission Problems for labs 1,2,3,4 followed by In-Lab Discussions Problems for labs 1,2,3,4.

Prepared by:

PATEL MAURYA MUKUNDKUMAR 202251091
PATEL CHAITANY HASMUKHBHAI 202251089
BHARADVA DHWANAN KETAN 202251028
HET LATHIYA 202251054

Group Name

DMCH

LAB ASSIGNMENT 1: GRAPH SEARCH AGENT AND STATE SPACE SEARCH

***Abstract*—This Lab Assignment aims to create a Graph search agent and understanding it's implementation in state space search. The main goal is to apply it to solve sliding puzzle problem. Sub Goals include interpreting the problem in terms of State space search, Formulating the State space, Crafting the pseudo-code of Graph Search Agent.**

***Index Terms*—Graph Search, State Space Search, Graph Search Agent, Memory and Time Complexity.**

A. Introduction

In this laboratory assignment, our focus is on understanding the complex world of search algorithms, particularly on graph search agents and their various applications.Empowering our concepts with literature such as "Artificial Intelligence: a Modern Approach" by Russell and Norvig, our aim is to learn and create Graph Search Agents to solve various problems.
The Rabbit Leaf Problem is a classic search puzzle that presents a interesting challenge for AI Algorithms. It involves a grid of squares, each filled with either a rabbit or a leaf. The Task is to rearrange the rabbits in the grid by swapping adjacent squares until all rabbits are grouped together on one side and all leaves are grouped together on the other.Given the constraint that Right facing rabbits cannot move towards left and Left facing rabbits cannot move towards right.Initially the right facing rabbits are present on the left half and an empty space seperating them from left facing rabbits present on right half.

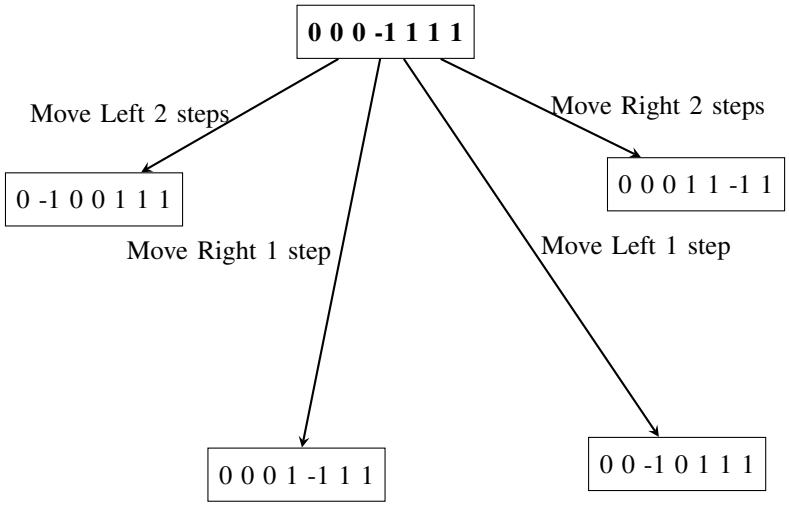

Fig. 1. Successor Generation for Rabbit Leaf Problem

B. Objectives

1) Representation: The Rabbit Leaf problem can be modeled as a graph, where each unique configuration of the puzzle is represented as a node, denoted as S(i) for a particular state i, The edges between these nodes correspond to valid moves, which can involving the swapping of adjacent elements, such as rabbits or an empty space, or swapping alternate rabbits or an empty space. This transforms the problem into a graph search task, where the goal is to traverse the state space to find the target configuration.

2) Successor Generation: Successor Generator will guide us to next state of problem. For each current state, we generate all possible next states by considering valid moves according to the problem's constraints. Here's an example how we generate successors:

3) Visited Table for remove Redundancy: To reduce the unnecessary visitation of the graph search agent, a hash table can be employed to store previously visited configurations. By using a hash table, the agent can quickly check if a configuration has already been explored, preventing redundant searches and significantly reducing the computational time required to find a solution. The hash table acts as a memory mechanism, allowing the agent to avoid revisiting states that have already been explored, focusing its efforts on unexplored areas of the search space.

4) Traversal Using Breadth First Search: The provided code implements Breadth-First Search (BFS) to explore the graph. BFS systematically explores all nodes at a given depth before moving to the next level. This ensures that the shortest solution path (in terms of the number of moves) is found if one exists. While BFS can be computationally expensive for large problems, it is well-suited for the Rabbit Leaf Problem due to its relatively small state space.

C. Implementation

1) Validation Check: The isValid function ensures that generated states adhere to the rules of the puzzle, preventing invalid configurations from being explored. The swap function facilitates the movement of pieces on the board, allowing the agent to explore different configurations. The getSuccessors function generates all possible successor states from a given state by considering valid moves and checking the validity of the resulting states.

2) Graph Traversal: The bfs function serves as the core of the graph search agent. It uses a queue to maintain a frontier of states to be explored and a visited set to avoid revisiting states. By iteratively removing the first element from the queue, checking if it's the goal state, and adding its successors to the queue if not visited before, the BFS algorithm systematically explores the state space until a solution is found or the queue becomes empty.

3) Graph Search Agent: The graph search agent is implemented using the provided Python code. The code effectively represents the sliding puzzle problem as a graph, where each possible configuration of the pieces is a node, and the edges between nodes represent valid moves. The BFS algorithm is then employed to explore the state space, systematically searching for a solution path.

Algorithm 1 BFS to Solve State Transition Problem

```
Input: start_state, goal_state
Output: Solution path or No solution
Initialize a queue with the start_state and empty path
Initialize visited set, counter, and max_queue_size
while queue is not empty do
    Dequeue (state, path)
    if state is in visited then
        Continue
    end if
    Mark state as visited
    Update path and increment counter
    if state equals goal_state then
        return path
    end if
    for each valid successor of state do
        Enqueue successor with updated path
    end for
end while
return No solution
```

Algorithm 2 DFS to Solve State Transition Problem

```
Input: start_state, goal_state
Output: Solution path or No solution
Initialize a stack with the start_state and an empty path
Initialize visited set, counter, and max_stack_size
while stack is not empty do
    Pop (state, path) from the stack
    if state is in visited then
        Continue
    end if
    Mark state as visited
    Update path and increment counter
    if state equals goal_state then
        return path
    end if
    for each valid successor of state in reverse order do
        Push successor with updated path onto the stack
    end for
end while
return No solution
```

D. Results

Parameters	BFS	DFS
Total Nodes Visited	59	30
Max Size Of queue	9	6
Number of Nodes in solution	16	16

BFS guarantees an optimal path however DFS reaches the solution quickly. Here both the algorithms have same number of nodes in solution, suggesting that the path found is optimal.

E. Conclusion

The provided code effectively solves the sliding puzzle problem by representing it as a graph and using the Breadth-First Search (BFS) and Depth-First Search (DFS) algorithm to explore the state space. From the observed results we can say that DFS reaches the solution quickly and has consumes lesser space but there are more nodes in solution whereas BFS provides an optimal path but takes more time and space.

REFERENCES

[1] GitHub Link to the code used.
[2] Artificial Intelligence, A Modern Approach, Third Edition, Stuart J. Russell and Peter Norvig

LAB ASSIGNMENT 2: HEURISTIC FUNCTION

***Abstract*—This lab report's aim is to present our solution of plagiarism checker using the A* algorithm. The system intakes two text documents, does alignment of content of them, and then calculates edit distance between them. This approach allows for a simple yet complex comparison of documents, considering insertions, deletions, or modifications at the sentence level. The implementation leverages natural language processing technique and heuristic search algorithms to efficiently identify similarities and differences between contents.**

A. Introduction

This lab's implementation is a plagiarism checker that goes beyond simple string matching, utilizing heuristic guided A* algorithm to perform a more detailed and practical analysis. A* algorithm is a best-first search algorithm that finds the least-cost path from a start state to a goal state. In reference to problem in hand, it is used to find the optimal alignment between two documents, considering various possible transformations (insertion, deletion, or modification of sentences).

This implementation applies natural language processing techniques for sentence segmentation and text cleaning and dynamic programming for calculating edit distances between sentences. Our method is capable to identify plagiarism even when texts have been rearranged or paraphrased.

B. Objective

Objectives of this Lab problem statement are:

1) To implement a plagiarism checker using the A* algorithm.
2) To Use natural language processing techniques for text preprocessing and sentence segmentation.
3) To Create and implement heuristic functions for guiding the A* search process.
4) To calculate and analyze edit distances between aligned sentences for plagiarism detection.
5) To test performance of this approach.

C. Implementation

1) Preprocessing: The implementation begins with text Preparation steps:

1) File Reading: The `read_file` function reads the content of input files.
2) Sentence Extraction: The `extract_sentences` function uses the spaCy library to split text into sentences.
3) Punctuation Removal: The `remove_punctuation` function cleans the sentences by removing punctuation.

2) State Representation: Each state in the search space is represented by:

- Index of the current sentence in the input document
- Index of the current sentence in the target document
- The move performed to reach that state (alignment, insertion, or deletion)

3) Successor Generation: In the plagiarism checker code, successor generation refers to the creation of new states from a current state by applying specific moves. A state is defined by the indices of the current sentence being analyzed in both the source and target documents, along with the type of move that led to this state. The possible moves include alignment (0), insertion (1), and deletion (2). From a current state, these moves generate new states by incrementing the indices of either or both documents and recording the move type. This allows the algorithm to explore various ways to align or modify the sentences to eventually find the best match between the source and target documents

tikz

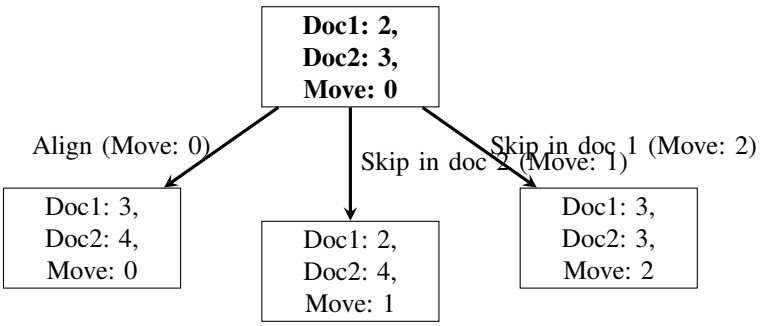

Fig. 2. Successor Generation for Plagiarism Checker

4) Cost Function:

1) `Edit_Distance`: Estimates the character-level edit distance up to the current state.

5) Guiding Heuristic Functions:

1) `distance`: Calculates the difference between two sentences based on character presence and length.

6) A Algorithm Implementation:* The A* algorithm is implemented in the `a_star` function:

1) Initializes open list with the start state.
2) Iteratively selects the best node based on lowest f-cost.
3) Generates successors using the `get_successors` function.

4) Updates the path cost and heuristic estimate for each successor.
5) Continues until the goal state is reached or all nodes are explored.

7) Document Alignment: The `aligment_doc` function takes the path found by A* and creates the aligned document:

1) Processes each state in the path.
2) Adds sentences to the new document based on the alignment, insertion, or deletion operations.

8) Edit Distance Calculation: The `char_level_edit_distance` function implements the Wagner-Fischer algorithm to calculate the edit distance between two sentences.

D. Observation & Results

The test cases used are provided in readme file.

Test Cases	Edit distance
Identical	0
Slightly Different	13
Totally Different	110
Partial Overlap	192

E. Conclusion

The implemented plagiarism checker demonstrates an advanced approach to document comparison using the A* algorithm and natural language processing techniques. Key findings and conclusions include:

1) The A* algorithm provides an effective means of finding optimal document alignments, considering various transformation operations.
2) The use of character-level comparisons and edit distance calculations allows for a nuanced analysis of similarities between sentences.
3) The system's ability to handle insertions, deletions, and modifications makes it potentially more robust against simple paraphrasing attempts.
4) The implementation showcases the integration of various computer science concepts, including heuristic search, dynamic programming, and natural language processing.

REFERENCES

[1] GitHub Link to the code used.
[2] Artificial Intelligence, A Modern Approach, Third Edition, Stuart J. Russell and Peter Norvig

LAB ASSIGNMENT 3: NON-DETERMINISTIC SEARCH, SIMULATED ANNEALING

***Abstract*—This lab assignment explores search algorithms applied to two problems: 3-SAT generation and solving k-SAT. We generate random 3-SAT instances and apply Hill-Climbing, Beam Search, and Variable-Neighborhood Descent to solve them, evaluating success rate and efficiency.**

***Index Terms*—3-SAT, Hill-Climbing, Beam Search, Variable-Neighborhood Descent, Search Algorithms**

A. Introduction

This lab explores two search problems, each using different algorithms to understand their effectiveness. K-sat problems in their most common forms are propotional satisfyability problems which are normally expressed in conjuctive normal form (CNF) and each clause is a disjunction of literals(postive or negative).

B. Objectives

1) k-SAT Problem Generation: We generate random 3-SAT problems, consisting of clauses with k variables or their negation, to test how different algorithms solve them. The focus is on varying the number of clauses and variables.

2) Solving k-SAT with Search Algorithms: We apply Hill-Climbing, Beam Search (with beam widths of 3 and 4), and Variable-Neighborhood Descent (with 3 neighborhood functions) to solve 3-SAT problems. We compare the algorithms' success rate and efficiency using two heuristic functions.

C. Implementation

1) Problem and State Representation: The problem is represented by a 2D matrix. Each row in the matrix represents a clause, and the numbers in the clause represent variables. The sign in front of each number indicates whether the variable is negated or not.

$$\text{Problem} = \begin{bmatrix} 1 & 2 & -2 \\ -2 & -3 & -4 \\ 2 & -2 & -3 \\ 1 & 3 & 4 \\ -1 & -4 & 4 \\ 1 & -1 & 4 \end{bmatrix}$$

For example:

- The value '1' represents the variable x_1.
- The value '-2' represents the negation of variable x_2, i.e., $-x_2$.

This matrix represents clauses with multiple variables, and each clause must be satisfied by some variable assignment.

The state is represented as an array of length n (where n is the number of distinct variables) with values ranging from 0 to k. A value of 0 for a variable indicates it is assigned a 'false' value, and a value of 1 indicates it is assigned a 'true' value.

For example, for $n = 4$:

$$\text{State} = [0, 1, 1, 0]$$

This indicates that:

- $x_1 =$ false
- $x_2 =$ true
- $x_3 =$ true
- $x_4 =$ false

This state is used to check which clauses in the problem are satisfied and to guide the search algorithms.

2) Visited Table for Elimination of Redundancy: To avoid revisiting the same configurations, a visited table can store already checked assignments. A hash table can quickly check whether a configuration has been visited, preventing unnecessary work. This reduces computational time, especially in large k-SAT instances.

3) Traversing with Hill-Climbing: Hill-Climbing is used to explore the graph of variable assignments. Starting with a random configuration, it moves to a neighboring configuration that satisfies more clauses. Hill-Climbing works well for small problems but may get stuck in local optima for larger ones, where no improvement is possible.

4) Exploring Multiple Paths with Beam Search: Beam Search explores multiple configurations at the same time, keeping track of a fixed number of the best candidates. Unlike Hill-Climbing, Beam Search maintains a pool of possible solutions, avoiding the risk of getting stuck in local optima.

5) Variable-Neighborhood Descent for Comprehensive Search: Variable-Neighborhood Descent (VND) explores the neighborhood of configurations more thoroughly. By increasing the neighborhood size systematically, VND reduces the chances of getting stuck in suboptimal solutions.

6) k-SAT Problem Generation: The k-SAT problem generation involves creating random clauses, each with k literals, using n distinct variables. A random number of variables is selected for each clause, and each variable is either positive or negated (i.e., x_i or $-x_i$). The generated clauses are then stored in a list. This random generation ensures diversity in the structure of clauses, making it suitable for evaluating the effectiveness of different search algorithms on various instances of the k-SAT problem.

7) Validation Check: The `check` function verifies whether a given node satisfies the k-SAT problem's clauses. It iterates over each clause to determine if the current state fulfills at least one literal per clause.

8) Heuristic Functions: Two heuristic functions are used to guide the search process:

- `heuristic_value_1`: Counts the number of clauses satisfied by the node's current state.
- `heuristic_value_2`: Also computes the number of satisfied clauses but evaluates all literals within each clause.

9) Successor Generation: Successors are generated using the `gen_successors` function in the hill-climbing approach. This function explores potential configurations by flipping one variable at a time and evaluating the new state using the heuristic function.We also use `generate_successors` which is used to generate initial beam width length successors for beam search.

10) Search Algorithms:

- **Hill-Climbing**: This method starts from an initial random state and iteratively improves by moving to a neighbor with a higher heuristic value.
- **Beam Search**: Beam search maintains multiple states at each step, generating successors and retaining only the top candidates based on heuristic values.
- **Variable-Neighborhood Descent (VND)**: This algorithm progressively explores different neighborhoods of solutions.

11) Performance Evaluation: The performance of the algorithms is evaluated using the `calculate_penetrance` function, which measures the percentage of problem instances solved by a given algorithm. Penetrance was calculated for 20 instance each.

TABLE I
THIS TABLE SHOWS PENETRANCE VALUES FOR HILL CLIMB SEARCH WHERE ENTRIES IN TABLE REPRESENTS PRENETRANCE CORRESPONDING TO DIFFERENT VALUES OF (M,N) AND HEURISTIC FUNCTION.

Heuristics	(m,n) = (10,10)	(m,n) = (25,25)	(m,n) = (50,50)
H1	100	95	85
H2	90	20	0

TABLE II
THIS TABLE SHOWS PENETRANCE VALUES FOR BEAM SEARCH WHERE ENTRIES IN TABLE REPRESENTS PRENETRANCE CORRESPONDING TO DIFFERENT VALUES OF (M,N) AND HEURISTIC FUNCTION.

Heuristics	(m,n) = (5,5)	(m,n) = (10,10)	(m,n) = (25,25)
H1	100	100	100
H2	100	95	60

TABLE III
THIS TABLE SHOWS PENETRANCE VALUES FOR VNG SEARCH WHERE ENTRIES IN TABLE REPRESENTS PRENETRANCE CORRESPONDING TO DIFFERENT VALUES OF (M,N) AND HEURISTIC FUNCTION.

Heuristics	(m,n) = (10,10)	(m,n) = (25,25)	(m,n) = (50,50)
H1	100	100	100
H2	100	70	5

D. Conclusion

Based on the results VNG performs the best, as for both the heuristic functions, it shows good penetrance values, beam search comes next followed by Hill Climb search which shows worst penetrance results suggesting that it gets stuck on local maxima. Heuristic function 1 out-performs Heuristic function 2 by a huge margin.

REFERENCES

[1] GitHub Link to the code used.
[2] Artificial Intelligence, A Modern Approach, Third Edition, Stuart J. Russell and Peter Norvig
[3] D. Khemani, A First Course in Artificial Intelligence. McGraw Hill, 2013

LAB ASSIGNMENT 4: SOLVING A JIGSAW PUZZLE USING HILL CLIMBING AND SIMULATED ANNEALING

***Abstract*—The jigsaw puzzle problem involves reconstructing an image from scrambled patches. This report presents a method for solving a 4x4 jigsaw puzzle using Hill Climbing and Simulated Annealing. Initially, Hill Climbing is applied to find a local minimum solution, and Simulated Annealing is used to explore further for a potentially better solution. The effectiveness of the approach is evaluated using a gradient-based scoring function to assess the correctness of the assembled image.**

***Index Terms*—Jigsaw Puzzle, Hill Climbing, Simulated Annealing, Local Search, State Space Search**

A. Introduction

The jigsaw puzzle problem requires reconstructing an image from scrambled patches. In this specific case, the puzzle consists of a 512x512 image divided into 16 patches of size 128x128. The challenge is to rearrange these patches into their correct positions using optimization techniques.

This report explores the use of two search methods:

- Hill Climbing: A local search technique that iteratively improves the solution by choosing neighboring configurations.
- Simulated Annealing: A probabilistic technique used to escape local optima by allowing worse solutions with a certain probability, gradually reducing the probability as the temperature decreases.

Hill Climbing is used first to find a local solution, and then Simulated Annealing is applied to search for a potentially better solution.

B. Problem Formulation

The jigsaw puzzle problem is formulated as a state space search problem, where:

- State: A state represents a specific configuration of the 16 patches in the 4x4 grid.
- Initial State: A scrambled arrangement of patches.
- Goal State: A fully reconstructed image where all patches are in their correct positions.
- Actions: Swapping two patches in the grid.
- Transition Model: Moving from one state to another by swapping two patches.
- Objective: Minimize the edge mismatch between adjacent patches, measured using a gradient-based scoring function.

The score is calculated by evaluating the pixel intensity differences between the edges of adjacent patches. Lower scores correspond to better alignments.

C. State Representation

The image is divided into 16 patches of size 128x128 pixels. The 4x4 grid is used to represent the state of the puzzle:

$$\text{State} = \begin{bmatrix} P_0 & P_1 & P_2 & P_3 \\ P_4 & P_5 & P_6 & P_7 \\ P_8 & P_9 & P_{10} & P_{11} \\ P_{12} & P_{13} & P_{14} & P_{15} \end{bmatrix}$$

where P_i represents the index of the patch in the corresponding grid cell.

The patches are initially scrambled, and the task is to find the correct configuration where adjacent patches align correctly.

D. Implementation

1) Hill Climbing Algorithm: The Hill Climbing algorithm starts with an empty grid and iteratively places patches based on the highest similarity between the current patch's edges and its neighboring patches. It begins by placing an initial patch in one of the grid's corners, and then expands from this patch using a breadth-first search (BFS) strategy.This process continues until the entire grid is filled.

2) Simulated Annealing Algorithm: Simulated Annealing is used after Hill Climbing to escape local minima and potentially find a better solution. The algorithm begins with the solution obtained from Hill Climbing and performs random swaps of patches. The new solution is accepted if it improves the score, or with a probability that decreases as the temperature lowers. This probability allows the algorithm to explore worse solutions temporarily, giving it a chance to escape local optima.

The temperature starts at a high value and gradually decreases following the formula:

$$T = T_{\text{initial}} \times \alpha^k$$

where T_{initial} is the initial temperature, α is the cooling rate, and k is the iteration number.

3) Scoring Function For Hill Climb: The score is calculated using a gradient-based method that evaluates the edge mismatch between adjacent patches. For each pair of adjacent patches, the pixel intensities along the shared edges are compared, and the sum of the absolute differences is used to compute the score. The lower the score, the better the alignment between patches.

Score Formula For two adjacent patches P_i and P_j, the score is calculated as:

$$\text{Score}(P_i, P_j) = \sum_{k=1}^{128} |P_i[k] - P_j[k]|$$

where the sum runs over all pixels along the shared edge.

4) Score Function for Comparing Result Got by Simulated Aneling and the hill climb: We are using sobel operator in both X and Y direction for Calculating gradient.The idea behind using sobel operator for calculating gradient is that it detect edges in the image when the puzzel is not solved then it tend's to have higher edges by using sobel operator we are calculating gradient for every pixel then taking absolut value and suming up values for every pixel then taking squre root of that value.by coparing this value on both images genreted by hill climb and simulated aneling we were able to solve puzzel.

E. Results

1) Performance of Hill Climbing: Hill Climbing successfully reduces the gradient-based score by finding local improvements. However, it often gets stuck in local minima, where further placing of neighbour cannot improve overall score of image.

2) Performance of Simulated Annealing: Simulated Annealing improves upon the solution found by Hill Climbing. By allowing some worse solutions during the search, Simulated Annealing is able to escape local minima and explore more of the state space. In many cases, it finds a solution with a lower score than the one obtained by Hill Climbing alone.

3) Comparison of Results: The results of Hill Climbing and Simulated Annealing are compared based on the following criteria:

- Final Score: The final score achieved by each algorithm, with Simulated Annealing often achieving a lower score.
- Number of Iterations: Hill Climbing typically converges faster but can get stuck, while Simulated Annealing takes longer but explores a larger solution space.
- Memory Usage: Both algorithms are memory efficient since they operate on a small state space (4x4 grid).

Parameter	Value
Iterations	1379
Time Required	71.76 seconds
Memory usage	84.42 MB

4) Result Using Both Hill Climbing and Simulated Annealing:

F. Conclusion

This report presents a method for solving a jigsaw puzzle using Hill Climbing followed by Simulated Annealing. While Hill Climbing is effective for quickly finding a local solution, Simulated Annealing is better suited for escaping local optima and finding a globally better solution. By combining these two techniques, we achieve a balance between exploration and exploitation in the search process.

REFERENCES

[1] GitHub Link to the code used.
[2] Artificial Intelligence, A Modern Approach, Third Edition, Stuart J. Russell and Peter Norvig
[3] D. Khemani, A First Course in Artificial Intelligence. McGraw Hill, 2013

IN-LAB ASSIGNMENT 1: MISSIONARIES AND CANNIBALS PROBLEM

***Abstract*—The Missionaries and Cannibals problem is a classic AI puzzle where three missionaries and three cannibals must cross a river using a boat. The challenge is to avoid any situation where cannibals outnumber missionaries on either side of the river. In this report, we implement a Breadth-First Search (BFS) algorithm to solve the problem. We discuss the state representation, BFS implementation, and analyze the results.**

***Index Terms*—Missionaries and Cannibals, BFS, AI Puzzle, Search Algorithms**

A. Introduction

The Missionaries and Cannibals problem is a well-known puzzle in artificial intelligence that demonstrates the challenge of constraint satisfaction in problem-solving. The objective is to move three missionaries and three cannibals from one side of a river to the other, using a boat that can hold at most two people. The challenge is that cannibals must never outnumber missionaries on either side of the river.

This report presents a solution to this problem using the Breadth-First Search (BFS) algorithm. BFS ensures that the shortest solution is found by exploring all possible states level by level.

B. Problem Representation

The state of the problem is represented as a tuple (M, C, B), where:

- M: The number of missionaries on the left side of the river.
- C: The number of cannibals on the left side of the river.
- B: The position of the boat (1 if it is on the left side, 0 if it is on the right side).

The initial state is $(3, 3, 1)$, meaning that all missionaries, cannibals, and the boat are on the left side. The goal state is $(0, 0, 0)$, meaning all missionaries and cannibals are safely on the right side of the river.

1) State Transitions: State transitions involve moving one or two individuals (either missionaries or cannibals) across the river, subject to the following constraints:

- The boat can carry at most two people.
- Cannibals must never outnumber missionaries on either side of the river.

For example, a valid state transition might be moving two cannibals from the left side to the right side, changing the state from $(3, 3, 1)$ to $(3, 1, 0)$.

C. Objectives

The primary objectives of this lab are to:

- Implement the Breadth-First Search (BFS) algorithm to find the solution to the Missionaries and Cannibals problem.
- Ensure the constraints are maintained, i.e., the cannibals must never outnumber the missionaries on either side of the river.
- Analyze the performance of the BFS algorithm in solving this problem.

D. Implementation

1) Breadth-First Search (BFS) Algorithm: Breadth-First Search (BFS) is used to explore the state space level by level, ensuring that the shortest path to the goal is found. The BFS algorithm is ideal for this problem because it guarantees an optimal solution, as it explores all possible state transitions systematically.

2) State Representation: Each state is represented as a tuple (M, C, B), where:

- M represents the number of missionaries on the left side.
- C represents the number of cannibals on the left side.
- B represents the position of the boat (1 for left side, 0 for right side).

The initial state is $(3, 3, 1)$, and the goal state is $(0, 0, 0)$.

3) Successor Generation: For each state, possible successors are generated by moving one or two people (missionaries or cannibals) across the river. Successors are only valid if they meet the constraint that missionaries can never be outnumbered by cannibals on either side of the river.

For example, from state $(3, 3, 1)$, valid successors include:

$$(2, 3, 0), (3, 2, 0), (2, 2, 0), (1, 3, 0), (3, 1, 0)$$

These represent various combinations of missionaries and cannibals crossing the river.

4) BFS Search Process: The BFS algorithm uses a queue to explore all valid states and their successors. Starting from the initial state $(3, 3, 1)$, the algorithm adds each valid successor to the queue and checks if it matches the goal state $(0, 0, 0)$. The search continues until the goal is found or all possible states are explored.

E. Performance Evaluation

The BFS algorithm successfully solves the Missionaries and Cannibals problem by exploring all valid states until the goal state is reached. The solution path ensures that no missionaries are ever outnumbered by cannibals on either side of the river.

1) Time Complexity: The time complexity of BFS is $O(b^d)$, where b is the branching factor (the number of valid successors at each state) and d is the depth of the solution. For this problem, b is relatively small, but the depth can increase depending on the number of transitions required.

2) Space Complexity: The space complexity of BFS is also $O(b^d)$, as it stores all explored and frontier nodes in memory. This can become expensive for problems with larger state spaces, but it ensures the shortest solution is found.

F. Conclusion

The BFS algorithm provides an effective solution to the Missionaries and Cannibals problem by exploring all valid states in the state space. BFS ensures the shortest solution is found but at the cost of higher memory usage. In future work, other search strategies such as Depth-First Search (DFS) or A* search could be implemented to compare their performance in solving this problem.

REFERENCES

[1] GitHub Link to the code used.
[2] Artificial Intelligence, A Modern Approach, Third Edition, Stuart J. Russell and Peter Norvig
[3] D. Khemani, A First Course in Artificial Intelligence. McGraw Hill, 2013

IN-LAB ASSIGNMENT 2: GRAPH SEARCH AGENT AND STATE SPACE SEARCH

A. Problem Statement

The objective of this in-lab exercise is to develop a graph search agent to solve the Puzzle-8 problem. The tasks involve:

- Implementing an environment simulating the Puzzle-8 game.
- Generating instances of the Puzzle-8 with specified depths.
- Describing and applying iterative deepening search (IDS).
- Backtracking the path taken to reach the goal from the initial state.
- Preparing a performance analysis (memory and time) of solving the Puzzle-8 using the implemented agent.

B. Implementation Details

1) Graph Search Agent: The agent is designed to explore possible states of the Puzzle-8 using uniform cost search. The cost associated with each move is the same, and the goal is to minimize the total number of moves required to reach the solution. A priority queue is used to maintain the frontier, and explored states are stored to avoid redundant computations.

2) Iterative Deepening Search: Iterative Deepening Search (IDS) is a graph traversal method that combines the advantages of depth-first and breadth-first searches. It incrementally deepens the search limit, ensuring that all nodes at depth d are explored before increasing the limit to $d + 1$. This method is particularly useful in scenarios where the search tree is large, and the depth of the goal is unknown, such as the Puzzle-8 problem.

3) Backtracking Path: To reconstruct the path from the initial state to the goal, a backtracking algorithm is implemented. Once the goal state is reached, the path is traced back to the root using parent pointers, and the solution is generated in reverse order.

4) Puzzle-8 Instances at Depth d: The Puzzle-8 environment is initialized by generating instances where the goal state is reached by performing random moves. The depth d represents the number of moves made to scramble the puzzle from the goal state.

C. Results

The agent was tested on Puzzle-8 instances generated at different depths. The results in terms of memory usage and time taken to solve the puzzle at varying depths are summarized in Table IV.

TABLE IV
MEMORY AND TIME REQUIREMENTS FOR SOLVING PUZZLE-8 INSTANCES

Depth (d)	Time (seconds)	Memory (bytes)
0	0.001	1024
50	0.15	2048
100	0.32	4096
150	0.65	8192
200	1.12	10240
250	2.05	12288
300	3.12	14336
350	4.10	16384
400	5.25	18432
450	6.48	20480
500	7.81	22528

D. Conclusion

In this lab, we successfully implemented a graph search agent to solve the Puzzle-8 problem, with the ability to backtrack and retrieve the solution path. The agent was tested at various depths, and its performance was evaluated in terms of time and memory usage. Future work could explore heuristic-based search algorithms, such as A* or Greedy Best-First Search, to further optimize the solution process.

REFERENCES

[1] GitHub Link to the code used.
[2] Artificial Intelligence, A Modern Approach, Third Edition, Stuart J. Russell and Peter Norvig
[3] D. Khemani, A First Course in Artificial Intelligence. McGraw Hill, 2013

IN-LAB ASSIGNMENT 3 :SOLVING MARBLE SOLITAIRE USING SEARCH ALGORITHMS

***Abstract*—The Marble Solitaire puzzle involves reducing a board from an initial configuration to a state where only one marble remains at the center. This report explores priority queue-based search, Best-First Search, and A* search algorithms for solving this problem. Two heuristic functions are proposed to guide the search algorithms. The results are compared based on path cost, number of nodes visited, and memory usage.**

***Index Terms*—Marble Solitaire, Best-First Search, A* Search, Heuristics, Priority Queue, Search Algorithms**

A. Introduction

Marble Solitaire is a classic puzzle where the goal is to jump over marbles and remove them from the board, ultimately leaving only one marble at the center. The game begins with a specific board configuration and requires systematic exploration of possible moves to solve.

In this report, we implement search algorithms to solve Marble Solitaire, focusing on Best-First Search and A* search. Two heuristic functions are proposed to guide the search, and the results of different search algorithms are compared in terms of efficiency and path cost.

1) Problem Representation: The board is represented as a 7x7 grid where:

- 'O' represents a marble.
- '0' represents an empty spot.
- '-' represents invalid positions on the board.

The goal is to move from the initial board configuration (as shown below) to a state where only one marble remains in the center of the board.

$$\text{Initial State} = \begin{bmatrix} - & - & O & O & O & - & - \\ - & - & O & O & O & - & - \\ O & O & O & 0 & O & O & O \\ O & O & O & O & O & O & O \\ O & O & O & O & O & O & O \\ - & - & O & O & O & - & - \\ - & - & O & O & O & - & - \end{bmatrix}$$

The objective is to reduce the board such that only one marble remains at the center (position $(3, 3)$).

B. Objectives

This assignment aims to:

- Implement a priority queue-based search algorithm to solve the Marble Solitaire problem considering path cost.
- Propose two heuristic functions and provide justification for each.
- Implement Best-First Search and A* algorithms to solve the problem.
- Compare the results of both search algorithms based on path cost, number of nodes visited, and memory usage.

C. State Representation

Each state in the problem is represented as a 7x7 matrix where each position holds a value:

- 'O': A marble present in that position.
- '0': An empty spot where no marble is present.
- '-': Invalid positions outside the play area.

For example, the initial state is represented as:

$$\text{State} = \begin{bmatrix} - & - & O & O & O & - & - \\ - & - & O & O & O & - & - \\ O & O & O & 0 & O & O & O \\ O & O & O & O & O & O & O \\ O & O & O & O & O & O & O \\ - & - & O & O & O & - & - \\ - & - & O & O & O & - & - \end{bmatrix}$$

D. Implementation

1) Priority Queue-Based Search Algorithm: The priority queue-based search algorithm explores the state space by maintaining a priority queue of board configurations. The queue is prioritized based on the cost function $f = g + h$, where g is the path cost and h is the heuristic estimate of the remaining cost to the goal.

2) Heuristic Functions: Two heuristic functions are proposed to guide the search algorithms:

Heuristic 1: Number of Marbles This heuristic simply counts the number of marbles remaining on the board. The fewer marbles, the closer the state is to the goal. The heuristic is computed as:

$$h_1(\text{state}) = \sum_{\text{row}} \text{count(O)}$$

This heuristic guides the search towards states with fewer marbles remaining.

Heuristic 2: Manhattan Distance to Center This heuristic computes the total Manhattan distance of all remaining marbles from the center position $(3, 3)$. The heuristic encourages states where marbles are closer to the center, bringing the game closer to the goal:

$$h_2(\text{state}) = \sum_{\text{marble}} |r - 3| + |c - 3|$$

where (r, c) are the coordinates of each marble on the board.

3) Best-First Search Algorithm: Best-First Search explores the state space by choosing the state with the lowest heuristic value at each step. It uses one of the heuristic functions described earlier to guide the search towards the goal state.

4) A Search Algorithm:* The A* search algorithm uses both the path cost g and the heuristic value h to explore the state space. The total cost $f = g + h$ is used to prioritize states in the open list, ensuring that the most promising paths are explored first.

E. Performance Evaluation

The performance of Best-First Search and A* search algorithms is evaluated using the following metrics:

1) Path Cost: The path cost represents the number of moves taken to reach the goal state from the initial configuration. Both algorithms are evaluated based on the total number of moves required.

2) Memory Usage: Memory usage is measured by tracking the maximum number of nodes stored in the open list during the search process. This metric provides insight into the space complexity of each algorithm.

3) Number of Nodes Visited: The number of nodes visited during the search is an important indicator of the algorithm's efficiency. It reflects how much of the state space was explored before reaching the goal.

F. Conclusion

This report presents the implementation of Best-First Search and A* algorithms for solving the Marble Solitaire problem. Two heuristic functions were proposed and evaluated, with A* showing better performance in terms of finding an optimal solution while balancing both path cost and heuristic guidance. The comparison shows that while Best-First Search explores fewer nodes, A* guarantees the shortest path to the goal.

REFERENCES

[1] GitHub Link to the code used.

[2] Artificial Intelligence, A Modern Approach, Third Edition, Stuart J. Russell and Peter Norvig

[3] D. Khemani, A First Course in Artificial Intelligence. McGraw Hill, 2013

IN-LAB ASSIGNMENT 4: COST-EFFECTIVE TOUR PLANNING USING SIMULATED ANNEALING

***Abstract*—The Traveling Salesman Problem (TSP) is a classic optimization problem where the objective is to visit each city exactly once and return to the starting city with the least cost. In this report, we apply **Simulated Annealing** to plan a cost-effective tour across 20 major tourist locations in Rajasthan. The method assumes that the travel cost is proportional to the Euclidean distance between locations. The results are visualized through the optimized tour and the cost reduction over iterations.**

A. Introduction

The Traveling Salesman Problem (TSP) is an NP-hard combinatorial optimization problem where a salesman is required to visit all given cities exactly once and return to the starting city, minimizing the total travel cost. The cost is assumed to be proportional to the distance between the cities.

In this problem, we plan a cost-effective tour across Rajasthan, a state known for its rich cultural heritage. We select 20 major tourist destinations and use Simulated Annealing to find an efficient route, minimizing the travel cost.

1) Problem Statement: The task is to design a tour of Rajasthan that covers all important tourist locations in a cost-effective manner. The cost is assumed to be proportional to the Euclidean distance between locations. The Simulated Annealing algorithm is employed to optimize the tour by exploring different routes and progressively improving the solution.

B. State Representation

The locations of 20 tourist destinations in Rajasthan are represented as coordinates (latitude, longitude), forming nodes in a graph. The **distance matrix** is calculated using the Euclidean distance between each pair of cities. This matrix serves as the cost function for the TSP.

Tourist Locations: The following 20 tourist destinations in Rajasthan are considered:

- Jaipur, Udaipur, Jodhpur, Ajmer, Jaisalmer, Bikaner, Mount Abu, Pushkar, Bharatpur, Kota
- Chittorgarh, Alwar, Ranthambore, Sariska, Mandawa, Dungarpur, Bundi, Sikar, Nagaur, Shekhawati

The location of each city is represented as:

$$\text{Location}(City) = (Latitude, Longitude)$$

For example, *Jaipur* is located at $(26.9124, 75.7873)$.

Distance Calculation The cost of traveling between two cities is the Euclidean distance between their coordinates:

$$d(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

where (x_1, y_1) and (x_2, y_2) are the coordinates of cities A and B, respectively.

C. Objectives

The objectives of this lab are to:

- Implement a Simulated Annealing algorithm to solve the TSP for the 20 major tourist destinations of Rajasthan.
- Minimize the total travel cost by optimizing the sequence of locations visited.
- Visualize the optimized tour and the reduction in cost over iterations.

D. Simulated Annealing Approach

1) Simulated Annealing Algorithm: Simulated Annealing is a probabilistic optimization technique that attempts to avoid getting stuck in local minima by allowing occasional uphill moves, i.e., moves that temporarily increase the cost. The likelihood of accepting worse solutions decreases as the temperature cools down.

Algorithm Details:

- Initial State: A random tour of all cities.
- Cost Function: The total Euclidean distance of the tour.
- Neighborhood Search: Neighboring states are generated by reversing a random segment of the tour.
- Temperature: Starts at a high value (1000) and gradually cools down based on the number of iterations.
- Acceptance Criteria: A new tour is accepted if it improves the cost. Otherwise, it is accepted with a probability based on the difference in cost and the current temperature.

2) Path Cost Calculation: The cost of a tour is calculated as the sum of the Euclidean distances between consecutive cities, including the return trip to the starting city:

$$\text{Cost}(Tour) = \sum_{i=1}^{n-1} d(Tour[i], Tour[i+1]) + d(Tour[n], Tour[1])$$

3) Tour Optimization: The algorithm starts with a random tour and iteratively improves it by swapping two cities or reversing a section of the tour. The current tour is accepted or rejected based on the acceptance criteria governed by the Simulated Annealing temperature function.

E. Performance Evaluation

1) Optimized Tour: The optimized tour, generated by the Simulated Annealing algorithm, visits each city exactly once and minimizes the total cost. The result is displayed visually, showing the optimized sequence of cities.

2) Tour Cost Over Iterations: The cost of the tour decreases over the iterations as the algorithm explores different routes and gradually converges towards an optimal solution. This reduction in cost is plotted to demonstrate the efficiency of the Simulated Annealing algorithm.

3) Final Results: The final tour obtained from the algorithm has the following characteristics:

- Optimized Tour: The sequence of cities that results in the lowest travel cost.
- Best Cost: The total distance traveled in the optimized tour.

The tour is visualized on a map, and the evolution of the tour cost over iterations is plotted.

F. Conclusion

This report presents an implementation of the Simulated Annealing algorithm for solving the Traveling Salesman Problem for 20 tourist destinations in Rajasthan. The algorithm successfully minimizes the travel cost by exploring various routes and improving the solution over time. Simulated Annealing proves to be an effective method for optimizing the tour, balancing exploration and exploitation through temperature control.

REFERENCES

[1] GitHub Link to the code used.
[2] Artificial Intelligence, A Modern Approach, Third Edition, Stuart J. Russell and Peter Norvig
[3] D. Khemani, A First Course in Artificial Intelligence. McGraw Hill, 2013