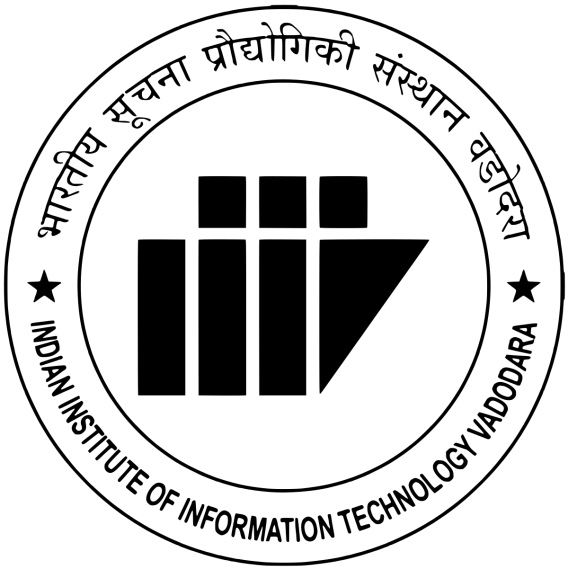


Indian Institute of Information Technology, Vadodara

CS367 Artificial Intelligence

Lab Report - MIDSEM

Group Name: LOGIQ

Group Members:

Heet Shah	202251121
Smit Shah	202251122
Parv Thummar	202251143
Tanuj Saini	202251141

Source Code: github.com/Shahsmit075/CS307_LAB-SUBMISSIONS

Lab Assignment 01 :

BFS and DFS Applications to Missionaries and Cannibals, and Rabbit Leap Problems

***Abstract*—This report analyzes two classic AI challenges: the Missionaries and Cannibals Problem and the Rabbit Leap Problem. We explore their state-space representations and solve them using Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms. The report compares the solutions, discussing optimality, time complexity, and space complexity.**

I. Introduction

Search algorithms play a crucial role in artificial intelligence for solving complex problems. This report focuses on two fundamental search techniques: Breadth-First Search (BFS) and Depth-First Search (DFS). We apply these algorithms to two classic AI puzzles:

1) Missionaries and Cannibals Problem: Three missionaries and three cannibals must cross a river using a boat that can carry at most two people. The challenge is to find a sequence of moves that transports everyone across without ever leaving a group of missionaries outnumbered by cannibals on either bank.
2) Rabbit Leap Problem: Three east-bound rabbits and three west-bound rabbits are separated by a single empty space on a line of stones. The rabbits can only move forward one or two steps, including jumping over one rabbit. The goal is to find a sequence of moves that allows the rabbits to cross each other.

II. State Space Representation

A. Missionaries and Cannibals Problem

Each state is represented as a tuple (m, c, b), where:

- m: Number of missionaries on the starting bank (0-3)
- c: Number of cannibals on the starting bank (0-3)
- b: Boat location (1 if on the starting bank, 0 if on the other bank)

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

Search Space Size: The total number of possible states is 4 * 4 * 2 = 32. However, not all of these states are valid due to the constraint that missionaries cannot be outnumbered by cannibals. After eliminating invalid states, the actual search space size is 16 states.

B. Rabbit Leap Problem

Each state is represented as a tuple of 7 elements, where:

- 'E': East-bound rabbit
- 'W': West-bound rabbit
- 'O': Empty stone

The initial state is ('E', 'E', 'E', 'O', 'W', 'W', 'W'), and the goal state is ('W', 'W', 'W', 'O', 'E', 'E', 'E').

Search Space Size: The total number of possible arrangements is 7! / (3! * 3! * 1!) = 140. However, not all of these states are reachable due to the movement constraints. The actual search space size is smaller but still considerable.

III. Search Algorithms

A. Breadth-First Search (BFS)

BFS explores the state space level by level, ensuring that the shallowest goal state is found first. It uses a queue data structure to maintain the frontier of unexplored states.

Approach:

1) Initialize a queue with the start state.
2) While the queue is not empty:
    a) Dequeue a state.
    b) If it's the goal state, return the path.
    c) Generate all valid successor states.
    d) Enqueue the successors if not visited.

B. Depth-First Search (DFS)

DFS explores the deepest states first before backtracking. It uses a stack data structure to maintain the frontier of unexplored states.

Approach:

1) Initialize a stack with the start state.
2) While the stack is not empty:
    a) Pop a state from the stack.
    b) If it's the goal state, return the path.
    c) Generate all valid successor states.
    d) Push the successors onto the stack if not visited.

IV. Problem Solving and Analysis

A. Missionaries and Cannibals Problem

1) BFS Solution: BFS guarantees finding the optimal (shortest) solution for this problem. It explores all possible moves systematically, ensuring that the first solution found uses the minimum number of steps.

2) DFS Solution: DFS may find a solution that is not optimal in terms of the number of steps. It could explore deeper, non-optimal paths before finding a valid solution.

3) Comparison:

- Optimality: BFS always finds the optimal solution, while DFS may not.
- Time Complexity:
  - BFS: O(bˆd), where b is the branching factor and d is the solution depth.
  - DFS: O(bˆm), where m is the maximum depth of the search tree.
- Space Complexity:
  - BFS: O(bˆd), as it needs to store all nodes at the current level.
  - DFS: O(bm), as it only needs to store nodes on the current path.

B. Rabbit Leap Problem

1) BFS Solution: BFS finds the optimal solution by exploring all possible moves level by level. It guarantees finding the sequence with the minimal number of rabbit moves.

2) DFS Solution: DFS explores one path deeply before backtracking. It may find a valid solution quickly, but it might not be the optimal (shortest) sequence of moves.

3) Comparison:

- Optimality: BFS guarantees the optimal solution, while DFS may find a longer sequence of moves.
- Time Complexity:
  - BFS: O(bˆd), where b is the average number of valid moves per state.
  - DFS: O(bˆm), where m is the maximum number of moves possible.
- Space Complexity:
  - BFS: O(bˆd), storing all nodes at the deepest level.
  - DFS: O(bm), storing only the current path.

V. Results

A. Missionaries and Cannibals Problem

1) BFS Results:

- Number of steps to reach the solution: 11
- Different states visited: 15
- Maximum queue size: 6

2) DFS Results:

- Number of steps to reach to solution: 11
- Different states visited: 14
- Maximum stack size: 12

B. Rabbit Leap Problem

1) BFS Results:

- Total Number of operations: 15
- Length of visited nodes: 136
- Maximum queue size: 56

2) DFS Results:

- Total Number of operations: 21
- Length of visited nodes: 22
- Maximum stack size: 34

VI. Conclusion

This report analyzed two classic AI problems using BFS and DFS algorithms. The results demonstrate that:

1) For the Missionaries and Cannibals problem, both BFS and DFS found the optimal solution with 11 steps. This is because the search space is relatively small and well-constrained.
2) For the Rabbit Leap problem, BFS found the optimal solution with 15 steps, while DFS found a longer solution with 31 steps. This highlights BFS's ability to find the shortest path at the cost of higher memory usage, while DFS may find a suboptimal solution but uses less memory.
3) BFS guarantees optimality but may be more memory-intensive, especially for large search spaces. DFS is more memory-efficient but may not find the optimal solution.
4) The choice between BFS and DFS depends on the specific problem characteristics, available computational resources, and whether finding the optimal solution is crucial.

These findings underscore the importance of algorithm selection in AI problem-solving and demonstrate how different search strategies can lead to varying results in terms of solution optimality and computational efficiency.

References

[1] Github Source Code.
[2] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," 4th ed. Pearson, 2020.
[3] D. Khemani, "A First Course in Artificial Intelligence," McGraw Hill Education, 2013.

Lab Assignment 02 :
Implementation of A* Algorithm for Puzzle-8 Problem

***Abstract*—This report presents the implementation of the A* algorithm to solve the Puzzle-8 problem. The puzzle involves moving tiles on a 3x3 grid to achieve a specified goal state. The solution is achieved using a priority queue, Manhattan distance as a heuristic, and various utility functions to explore possible moves.**

By aligning sentences from two papers according to how similar they are, the A* algorithm is used in the plagiarism detection system to detect possible plagiarism in text alignment. The system compares phrases using edit distance as the cost function and estimates the remaining alignment cost using a heuristic. When similar or identical text sequences are found between the papers, the alignment procedure effectively identifies them and marks them as possible instances of plagiarism. These two examples show how adaptable and effective A* is at handling a range of search and optimization issues.

I. Introduction

Within the field of heuristic search problems and artificial intelligence, the Puzzle-8 problem is a well-known challenge. The goal of the problem is to move numbered tiles in a grid in order to make a sequence of legitimate tile moves that will lead to a predefined target state. The A* method is utilized here because it can balance path cost and heuristic estimations to minimize exploration time and resources. It is known for its optimality and efficiency in search issues.

Using both actual costs (from the start state) and an expected cost (heuristic) to reach the end state is the fundamental principle of A*. Because of this, it's a strong option for tackling puzzles like Puzzle-8, where the objective is to explore as much as possible in the fewest steps possible.

Plagiarism Detection System using A* Search aligns sentences between two texts using the A* algorithm to identify plagiarism. It finds the optimal match by combining real and heuristic costs and focuses on semantic similarity to effectively identify potentially stolen content.

The system preprocesses the text by removing punctuation and normalizing sentences before applying the search. It calculates sentence similarity and flags pairs that exceed a defined similarity threshold, providing clear output for further evaluation.

II. Problem A - Approach for A* in Puzzle-8

The A* algorithm efficiently solves the Puzzle-8 problem by combining path cost (g) and heuristic estimate (h) to prioritize the most promising paths. It uses a priority queue, where nodes with the lowest total cost ($f = g + h$) are explored first.

A. 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.

Algorithm 1 8 Puzzle Search Algorithm

```
1:  procedure SEARCH(PriorityQueue pq, Array visited)
2:      if pq.isEmpty() then
3:          return false
4:      end if
5:      puz ← pq.extract()    ▷ all possible successors to puz
6:      if search(pq) then
7:          return true
8:      end if
9:      for each suc in successors do
10:         if suc not in visited then
11:             pq.insert(suc)
12:             visited.insert(suc)
13:         end if
14:     end for
15:     if search(pq.visited) then
16:         return true
17:     else
18:         return false
19:     end if
20: end procedure
```

Each puzzle state is represented by a node, storing:

- state: The current configuration of tiles.
- parent: The previous state.
- g: The cost from the start to the current state.
- h: The estimated cost (Manhattan distance) to the goal.
- f: The total cost ($f = g + h$), used for prioritization.

A custom comparison (`__lt__`) ensures nodes are ordered by f in the priority queue.

B. Write a collection of functions imitating the environment for Puzzle-8

- **Manhattan Distance** : Heuristic determines how many total vertical and horizontal steps are needed to move each tile to its target position. This is done using the `manhattan_distance` function. This heuristic gives a precise estimate of the number of moves required, making it a good fit for Puzzle-8.

- **Generating Successors** : Through the use of four different ways to slide the empty tile (0), the `get_successors`

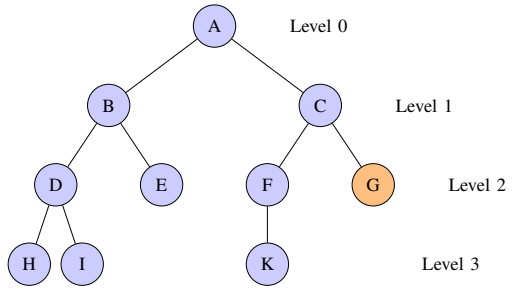
function creates valid next states. It guarantees that boundary restrictions are followed, such as staying on the grid. After the objective is accomplished, successors are returned as new Node objects with updated states, allowing for backtracking across parent nodes.

- **A* Search Algorithm** : The start state is pushed into a priority queue by the `a_star` function, which initiates the search. By creating their successors, updating their costs, and rearranging the queue according to total cost, it investigates nodes (f). The function reconstructs the solution path by going back through parent nodes if the aim is accomplished. Efficiency is increased by not going over previously investigated states by using a visited set.

- **Random Puzzle Generation** : A goal state is generated by randomly selecting successors of the start state for D moves, ensuring the goal is reachable. This approach helps test the algorithm's ability to solve a variety of configurations.

C. Describe what is Iterative Deepening Search.

Iterative deepening depth first search (IDDFS) is a hybrid of BFS and DFS. In IDDFS, we perform DFS up to a certain "limited depth," and keep increasing this "limited depth" after every iteration. Basically it performs DFS at every depth thus reducing the space complexity compared to BFS.



1st Iteration: → A
2nd Iteration: → A, B, C
3rd Iteration: → A, B, D, E, C, F, G
4th Iteration: → A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

D. 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.

Uniform Cost Consideration: The function only has to follow the parent links without taking into account the different edge costs because every move has the same cost. This guarantees the shortest path in terms of moves and streamlines the process of going backward.

E. Generating Puzzle-8 Instances at Depth "d"

Depth Controlled State Generation: Adapt the `generate_start_state` function to take a depth parameter "d" in the `Environment` class. To establish the initial state, the function should execute exactly "d" random valid moves from the goal state.

Move Selection: To produce valid moves at each step, use the `get_next_states` method. To maintain the appropriate depth, pick one of these moves at random, being careful not to undo the preceding step.

Depth Verification: Conduct an examination to make sure that the starting state that is generated is precisely "d" steps away from the desired state. For this to prevent loops or backtracking, the random move selection procedure might need to be modified.

F. Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth "d") using your graph search agent.

TABLE I
AVERAGE TIME AND MEMORY USAGE AT DIFFERENT DEPTHS

Depth	Avg Time (s)	Avg Memory (MB)
0	0.00004	56.00
50	0.03484	17914.40
100	0.18185	55487.04
150	0.46079	102912.32
200	0.39818	96759.04
250	0.40829	109572.96

III. PROBLEM B - PLAGIARISM DETECTION SYSTEM USING A* SEARCH

A. Functions for Plagiarism Detection

- **Text Preprocessing**: The input text is normalized by the `preprocess_text` function by removing punctuation, changing it to lowercase, and dividing it into sentences. By doing this, the text is made clear and prepared for comparison.

- **Semantic Distance**: By comparing the quantity of common terms in two sentences, the `semantic_distance` function calculates how unlike the sentences are from one another. A number between 0 and 1, where 1 denotes the absence of similar terms and 0 denotes identical phrases, is what is returned.

- **Generating Successors**: The `get_successors` function increments sentence indices for each document to provide valid next states. In order to make sure the state adheres to the sentence boundaries of both texts, it returns a list of successor nodes.

- **A* Search Algorithm**: Putting the start state in a priority queue is the first step taken by the `a_star_search` function. It creates successors and investigates the most promising nodes based on their overall cost ($f = g + h$). The function retraces the alignment path across parent nodes after reaching the goal state.

- **Plagiarism Detection**: The alignment from the A* search is used by the `detect_plagiarism` function to detect possible plagiarism. It returns the flagged pairs after determining if the similarity between aligned texts is greater than a predetermined threshold.

IV. Search Algorithm

The search algorithm used for solving the Puzzle-8 problem is outlined below. It defines the process of managing the priority queue and visited nodes to explore successors effectively.

A* is used in the plagiarism search algorithm to align sentences between two documents. By controlling a priority queue, it investigates sentence pairs without going back to previous states and gives preference to paths with the lowest overall edit cost. The search proceeds until both documents are completely aligned, with successors being generated based on sentence indices.

V. Conclusion

Because of its heuristic-based assistance, A* solves Puzzle-8 better than Iterative Deepening Search (IDDFS). A* investigates fewer nodes, particularly at deeper depths, by efficiently constricting the search space. For the search to be efficiently guided, the Manhattan distance heuristic is essential. The heuristic's beneficial effects on performance were confirmed when the algorithm was assessed by counting the nodes searched and the amount of time required.

By reducing the search space and aligning phrases based on semantic similarity, the A* algorithm works well at identifying plagiarism. By combining path costs and heuristic estimations, it is possible to explore sentence pairs optimally and identify potentially plagiarized content with accuracy. This method minimizes pointless comparisons while effectively identifying similar content.

References

[1] Github Source Code.

[2] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," 4th ed. Pearson, 2020.

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

Lab Assignment 03 :

Heuristic Functions and Non-Classical Search Algorithms for Marble Solitaire and SAT Problems

***Abstract*—This study investigates the use of heuristic functions and non-classical search algorithms to effectively reduce search space for tackling large-scale issues. Three different tasks are highlighted: Marble Solitaire, k-SAT, and 3-SAT. Heuristic-guided search techniques, such as Best-First Search and A*, are applied to each problem. The report describes how the challenges were formulated, what heuristics were used, what algorithms were implemented, and provides an analysis of the outcomes.**

I. Introduction

Efficient solutions are necessary for large state-space search problems in order to reduce computing resources such as time and money. Heuristic functions are essential in directing these searches and expediting the procedure by calculating the amount of work required to attain a target state. This paper examines the use of heuristic functions in three particular problems—Marble Solitaire, k-SAT, and 3-SAT—that provide formidable obstacles to search optimization. Non-classical search algorithms like A* and Best-First Search are used to address these problems by attempting to minimize the state space and improve search efficiency overall. Specifically, customized heuristic functions are presented to enhance the accuracy and speed of the search procedure, offering more profound understandings of how these tactics might be applied to address challenging computational problems.

II. Problem 1: Marble Solitaire

A. Problem Description

In the game Marble Solitaire, the object is to clear the board of all the marbles except for one by leaping over nearby stones. The last marble should be placed in the middle of the board. Because there are so many different ways to arrange the boards in Marble Solitaire, the state space is large. Thus, in order to effectively locate the best answer, heuristic functions and search algorithms are crucial.

B. Heuristic Functions

Two heuristic functions were developed for Marble Solitaire:

- **Heuristic 1: Number of Marbles Remaining** - This heuristic counts the number of marbles remaining on the board. The objective is to get to a point where there is only one marble left. Reduced marbles indicate closerness to the solution, offering an easy-to-use but reliable indicator.
- **Heuristic 2: Manhattan Distance** - This heuristic determines how far each surviving marble is from the board's center in Manhattan distance. Assuming that marbles in the center have a higher probability of contributing to the ultimate solution, this heuristic directs the search toward consolidating marbles toward the center.

C. Search Algorithm Implementations

- **Priority Queue-Based Search:** The search through the state space was controlled using a priority queue, where states were ranked based on their combined path cost and heuristic values. This method ensures that states with lower accumulated costs are prioritized for exploration.
- **A* Search Algorithm:** The A* algorithm was constructed using the cost function $f(n) = g(n) + h(n)$, where $g(n)$ represents the path cost from the initial state to the current state, and $h(n)$ is the heuristic estimate of the remaining distance to the goal.

Algorithm 1 A* using Manhattan Distance Heuristic

```
Initialize priority queue Q with initial state
g(n) ← 0 for initial state
h(n) ← heuristic estimate for initial state
f(n) ← g(n) + h(n) for initial state
while Q is not empty do
    n ← node in Q with lowest f(n)
    if n is the goal state then
        return Solution path
    end if
    for each successor m of n do
        g(m) ← g(n)+ cost to move from n to m
        h(m) ← heuristic estimate for state m
        f(m) ← g(m) + h(m)
        Add m to Q
    end for
end while
return No solution found
```

D. Results

Although the Manhattan Distance heuristic consistently produced faster results by directing the search toward the board's center, both algorithms demonstrated good performance. Compared to Best-First Search, A* search using the Manhattan

heuristic required fewer node explorations to get the best solution.

III. Problem 2: k-SAT

A. Problem Description

The k-SAT problem is a well-known NP-complete problem in Boolean logic, which involves determining if a given Boolean formula composed of clauses with exactly k literals can be satisfied. Every clause is made up of a disjunction of k literals, which can be either a variable or its opposite.

B. Heuristic Functions

- **Clause Satisfaction Heuristic:** This heuristic counts the number of clauses that are satisfied under a specific variable assignment in order to evaluate the progress made towards solving the k-SAT problem.

C. Search Algorithm Implementations

- **Random Assignment Generator:** Truth values are assigned to variables at random, and the heuristic counts the number of sentences that are satisfied with the current assignment.
- **Best-First Search:** A priority queue is utilized to investigate variable assignments that optimize the quantity of satisfied clauses.

D. Results

Smaller versions of the k-SAT issue were successfully solved reasonably rapidly using the Best-First Search algorithm. However, frequent backtracking was noted as the problem size rose, especially as the number of clauses increased.

IV. Problem 3: 3-SAT

A. Problem Description

The 3-SAT problem is a specific instance of the k-SAT problem, in which every clause in the Boolean formula is made up of precisely three literals, which may be variables or their negations.

B. Heuristic Functions

- **Clause Satisfaction Heuristic (Adapted for 3-SAT):** This heuristic counts the number of satisfied clauses according to a specified variable assignment, modified to consider the fixed clause size of three literals.

C. Search Algorithm Implementations

- **Hill-Climbing Algorithm:** Starts with a random assignment and iteratively improves it by flipping variable values to increase the number of satisfied clauses.
- **Beam Search Algorithm:** Maintains multiple potential solutions, generating and evaluating neighbors for each candidate, keeping only the most promising ones within a predetermined beam width.

D. Results

Hill-Climbing worked effectively for smaller problems but often became trapped in local optima for larger problems. Beam Search, with a broader beam width, produced better results by simultaneously examining more possible solutions.

V. Results and Analysis

A. A* vs. Best-First Search

The A* algorithm proved most efficient for Marble Solitaire, effectively balancing path cost with heuristic guidance, leading to quicker solutions. In contrast, while Best-First Search effectively explores high-priority nodes, it often requires more backtracking and may miss optimal paths due to its reliance on heuristics.

B. k-SAT and 3-SAT Comparisons

Beam Search excelled, particularly in larger instances, by maintaining multiple candidate solutions that allowed for broader exploration of the search space. The Variable Neighborhood Descent (VND) algorithm, using three neighborhood functions, demonstrated the highest penetrance for 3-SAT problems through its adaptable search strategy. Although Hill-Climbing is quick, it has the lowest penetrance due to frequent entrapment in local optima.

References

[1] Github Source Code.
[2] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," 4th ed. Pearson, 2020.
[3] D. Khemani, "A First Course in Artificial Intelligence," McGraw Hill Education, 2013.

Lab Assignment 04 :

Heuristic Search Solutions: Jigsaw Puzzle Problem and Travelling Salesman Problem

***Abstract*—This report addresses two computational problems solved using heuristic-based algorithms. The first part focuses on the Jigsaw Puzzle Problem, where heuristic search algorithms are applied to arrange scrambled puzzle pieces efficiently. The second part explores the Travelling Salesman Problem (TSP) using Simulated Annealing to find an optimal or near-optimal route that visits each city exactly once. We present the methodologies, algorithms, and results for both problems, demonstrating the effectiveness of heuristic search and Simulated Annealing in solving complex combinatorial optimization problems.**

I. Introduction

Heuristic algorithms provide efficient solutions to problems that are computationally intensive due to their large search space. In this report, we address two such problems: the Jigsaw Puzzle Problem and the Travelling Salesman Problem (TSP). Both problems are NP-hard and require intelligent search strategies to arrive at optimal or near-optimal solutions.

The Jigsaw Puzzle Problem involves arranging scrambled puzzle pieces to form a complete image, and it is solved using heuristic search methods that minimize the number of incorrect piece alignments. The TSP is a well-known combinatorial optimization problem where the task is to minimize the total distance for visiting a series of cities exactly once, for which we employ the Simulated Annealing algorithm.

In the following sections, we will present the methodology and results of solving each problem.

II. Part A: Jigsaw Puzzle Problem Using Heuristic Search

A. Methodology

The solution approach employs a heuristic search algorithm designed to arrange jigsaw puzzle pieces based on edge matching and other heuristic metrics. The key components of the methodology are as follows:

- **Energy Class:** A class Energy is created to compute the cost based on incorrect edge alignments between adjacent puzzle pieces. The class includes methods such as `getLeftRightEnergy()` and `getUpDownEnergy()`, which calculate pixel difference costs along the horizontal and vertical edges of puzzle pieces.
- **Tile Representation:** Each puzzle piece is represented as a grid tile with neighboring tiles in the arrangement. The heuristic search evaluates how well the tiles fit together by comparing their adjacent edge pixel values, with mismatches contributing to the total energy or cost.
- **Energy Evaluation:** The heuristic function `getEnergyAround()` calculates the energy for each tile based on the alignment with its neighboring tiles. The total energy of the current puzzle configuration is evaluated using the `energy()` function, which sums the energies from all tile edges in the puzzle.
- **Heuristic Search Process:** The heuristic search algorithm iteratively minimizes the total energy by rearranging puzzle pieces to reduce the number of mismatched edges. At each step:
  1) **Evaluate Current Configuration:** Calculate the total energy of the current puzzle configuration.
  2) **Apply Move:** The algorithm identifies the move (tile swap or rotation) that most reduces the energy and applies it.
  3) **Convergence:** The process is repeated until no further improvements in energy can be made, which signifies the puzzle is solved.

B. Algorithm Description

The pseudocode for the heuristic-based jigsaw puzzle solver is as follows:

Algorithm 1 Heuristic-based Jigsaw Puzzle Solver

```
procedure SolveJigsaw(puzzle)
    while not puzzle.is_solved() do
        move ← puzzle.find_best_move()
        puzzle.apply_move(move)
    end while
    return puzzle
end procedure
```

III. Discussion

The heuristic-based search algorithm efficiently solved the jigsaw puzzle. The time complexity of the algorithm depends on the puzzle's size and the number of possible configurations. The edge-matching heuristic guided the search, significantly

reducing the number of iterations compared to a brute-force approach.

IV. Part B: Travelling Salesman Problem Using Simulated Annealing

A. Methodology

The Travelling Salesman Problem (TSP) was solved using the Simulated Annealing (SA) algorithm. The steps involved in this method are as follows:

- **City Representation:** The cities in the TSP are represented as 2D points, each with x and y coordinates. The Euclidean distance between two cities is calculated using a distance function, which is later used to compute the total distance of a tour (a path visiting all cities exactly once).
- **Initialization:** A random initial tour (sequence of city visits) is generated, serving as the starting point for the Simulated Annealing process. For instance, the coordinates of cities from the Rajasthan dataset were used to initialize the tour. The total distance of this initial random route is calculated and serves as the initial cost in the optimization process.
- **Simulated Annealing Process:** The Simulated Annealing algorithm aims to minimize the total travel distance by exploring different tours while gradually lowering the "temperature." The key steps include:
  1) **Generate New Solution:** At each iteration, a new solution is generated by randomly swapping two cities in the current tour.
  2) **Evaluate Cost:** The new solution is evaluated by computing the total distance of the tour.
  3) **Acceptance Criterion:** If the new solution is better (lower total distance), it is accepted. Otherwise, it is accepted with a probability that depends on the current temperature and the magnitude of the cost difference.
  4) **Temperature Schedule:** The temperature is gradually decreased over time, reducing the chances of accepting worse solutions as the process converges towards an optimal or near-optimal solution.

V. Results

The Simulated Annealing algorithm was implemented an testedon two datasets: Rajasthan tourist locations and VLSI design points. Results are as follows:

- **Rajasthan Dataset:** An optimal route was found to visit 25 tourist locations in Rajasthan. The simulated annealing algorithm successfully minimized the travel cost.
- **VLSI Dataset:** The algorithm was also applied to VLSI datasets with varying points (131, 237, 343, 379, and 380), achieving significant reductions in travel cost compared to random initial solutions.

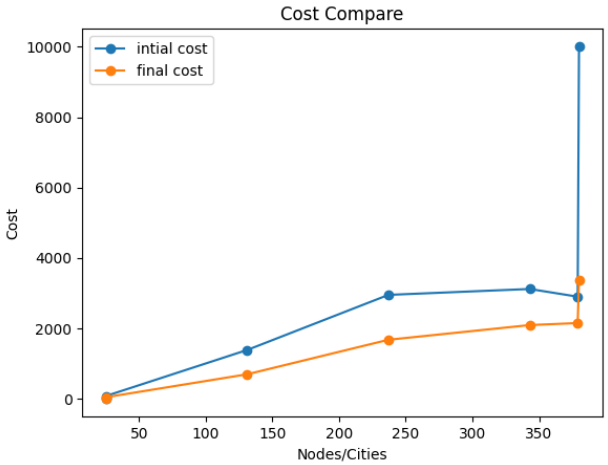


Fig. 1: Cost comparison for 25 locations in Rajasthan

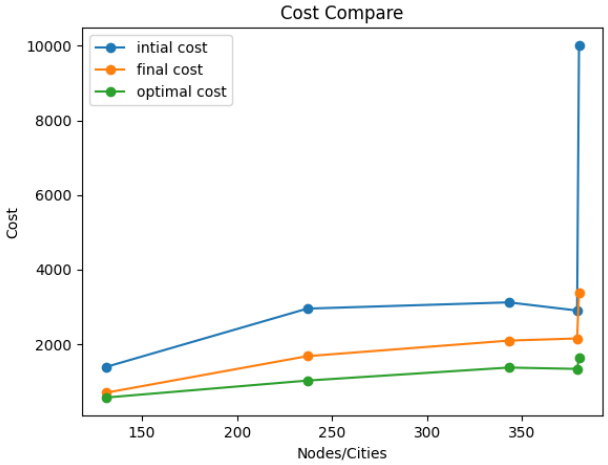


Fig. 2: Cost comparison for VLSI dataset with 237 points

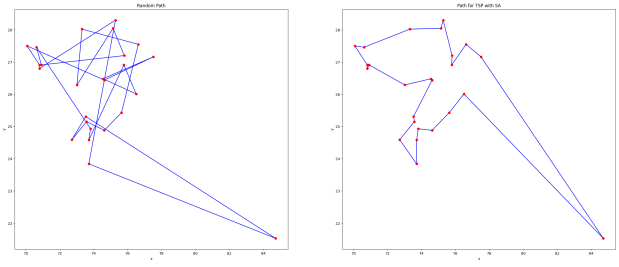


Fig. 3: Final result showing optimal path for the dataset

VI. Discussion

The Simulated Annealing algorithm effectively reduces the total travel cost for the TSP. By gradually reducing the temperature, the algorithm avoids getting stuck in local minima and

explores the solution space thoroughly. The final costs were close to the optimal costs for both the Rajasthan and VLSI datasets.

VII. Conclusion

In this report, we have successfully solved the Jigsaw Puzzle Problem using heuristic search and the Travelling Salesman Problem using Simulated Annealing. Both algorithms produced efficient solutions to their respective problems, demonstrating the value of heuristic-based approaches in solving complex computational challenges.

References

[1] Github Source Code.

[2] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," 4th ed. Pearson, 2020.

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