Investigating the Impact of Different Search Strategies (Breadth First, Depth First, A*, Best First, Iterative Deepening, Hill Climbing) on 8-Puzzle Problem Solving - A Case Study

[1]Rahul Jain, [2]Megha Patel

[1]Assistant Professor, [2]Assistant Professor
[1,2]Computer Engineering, Ganpat University, Mehsana, India

Abstract: The 8-puzzle problem is a classic benchmark problem in artificial intelligence and computer science, which involves finding the optimal sequence of moves to transform an initial state of a sliding tile puzzle into a goal state. This case study explores the use of various algorithms to solve the 8-puzzle problem, including uninformed search algorithms such as breadth-first search, depth-first search, and iterative deepening search, as well as informed search algorithms such as A* search and its variants. We also examine the use of heuristic functions, such as the Manhattan distance and the misplaced tile heuristic, to guide the search process and improve efficiency. We compare the performance of these algorithms on a set of test cases and analyze the trade-offs between optimality, completeness, and efficiency. We also discuss the implications of these results for real-world applications of the 8-puzzle problem, such as game design, robotics, logistics, and cryptography. This case study provides a comprehensive overview of the state-of-the-art approaches to solving the 8-puzzle problem using AI techniques, and highlights the challenges and opportunities of this fascinating problem.

Index Terms: 8-puzzle problem, Algorithms, Artificial Intelligence, Breadth first search, Depth first search, A* search, Hill Climbing Search, Case Study, Uninformed Search, Informed Search, Heuristic, Python Code, Complexity, Applications.

I. INTRODUCTION

The 8-puzzle problem, and sliding tile puzzles in general, have been studied extensively in the fields of artificial intelligence, computer science, mathematics, and psychology. A large body of research exists on various aspects of the problem, including search algorithms, heuristics, complexity, learning, and cognitive processes. In recent years, the 8-puzzle problem has also been used as a benchmark problem for evaluating the performance of machine learning algorithms, particularly in the area of reinforcement learning. Researchers have developed various techniques for training agents to solve the 8-puzzle using deep reinforcement learning, and have achieved state-of-the-art results on the problem.

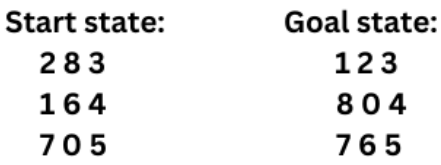
Use Case: Solving the 8-puzzle problem, Actors: User and Solver algorithm. Use Cases: Enter initial state of the puzzle: The user inputs the initial configuration of the puzzle, Solve the puzzle: The solver algorithm solves the puzzle using the input configuration and returns the solution, display solution: The user can view the solution to the puzzle, verify solution: The user can check if the solution provided by the solver algorithm is correct. Note: This is a basic use case diagram and there may be additional use cases and actors depending on the specific requirements of the problem.

In addition, there has been research on various extensions and variations of the 8-puzzle problem, such as the n-puzzle problem, the torus puzzle, the pattern database heuristic, and the sliding coin puzzle. These variations introduce new challenges and complexities, and provide new opportunities for research and innovation. Overall, the 8-puzzle problem remains an active area of research, with new techniques and approaches being developed all the time. The problem's simplicity, elegance, and universality make it a rich and rewarding subject for exploration and discovery. The 8-puzzle problem is not classified as either (Non-Polynomial) NP-complete or NP-hard. These are complexity classes used to describe problems that are at least as difficult as the hardest problems in the class NP, which includes problems that can be solved in polynomial time using a non-deterministic Turing machine.

The 8-puzzle problem is a combinatorial optimization problem, which means that it involves finding the optimal solution from a large set of possible solutions. It can be solved using a variety of search algorithms, such as depth-first search, breadth-first search, A* search, and iterative deepening search. These algorithms can find the optimal solution in a reasonable amount of time and space complexity, and do not require an exponential amount of time to solve the problem. Therefore, the 8-puzzle problem is generally considered to be a problem that can be solved efficiently using standard search algorithms, and is not considered to be in the class of NP-complete or NP-hard problems. However, it is worth noting that there are variations of the 8-puzzle problem, such as the 15-puzzle and the 24-puzzle, that have larger search spaces and may be more challenging to solve optimally.

The 8-puzzle problem is an example of a search problem in Artificial Intelligence (AI). There are many different search algorithms that can be used to solve the 8-puzzle problem, including depth-first search, breadth-first search, and A* search. A* search is a particularly popular algorithm for this problem because it uses heuristics to guide the search towards the goal state more efficiently. The objective is to transform the initial state into the goal state by moving the tiles one at a time, with the blank tile serving as a placeholder. The challenge is to find an optimal sequence of moves that can lead to the goal state from the initial state, given the constraint that only the blank tile can be moved at any given time. This problem is commonly used as a benchmark for evaluating search algorithms and constraint satisfaction algorithms in AI as it involves finding a solution that satisfies certain constraints (for example the tiles must be moved in a certain order). For example, figure 1 is showing start state and goal state might look like this;

Fig. 1 Start and Goal States



The 8-puzzle problem, and more generally sliding tile puzzles, have a wide range of applications in real life. Here are a few instances: Game design: Sliding tile puzzles, including the 8 puzzles, are often used as mini-games or challenges within larger games. They can provide an interesting and fun puzzle-solving experience for players. Robotics: The 8 puzzles can be used as a benchmark problem in robotics for testing and developing algorithms for motion planning, path finding, and manipulation. Solving the 8 puzzle involves reasoning about the relationships between different objects in the environment, which is an important skill for robots. Education: The 8 puzzles can be used as a teaching tool in mathematics and computer science courses to introduce concepts such as algorithms, search, and heuristics. Solving the 8 puzzles can help students develop problem-solving and critical thinking skills. Logistics: The 8 puzzles can be used as a model for logistics problems, such as warehouse organization or delivery route optimization. The problem of finding the optimal sequence of moves to reach a goal state is similar to the problem of finding the optimal sequence of operations to complete a set of tasks. Cryptography: The 8 puzzles can be used as a basis for cryptographic algorithms that involve shuffling and un-shuffling sequences of data. The problem of finding the correct order of the tiles in the 8 puzzles can be seen as a form of decryption, where the correct order is the decrypted message.

Overall, the 8-puzzle problem and sliding tile puzzles in general are a versatile and interesting class of problems that have many practical applications in various fields.

II. Area of Discussion

1. Sliding tile puzzle: The 8-puzzle problem is a specific instance of the more general class of sliding tile puzzles, which involve moving tiles or pieces around a board to reach a goal configuration.
2. Search algorithms: One of the main techniques used to solve the 8-puzzle problem is search algorithms, which involve exploring the space of possible states to find a sequence of moves that leads from the initial state to the goal state.
3. Heuristics: To improve the efficiency of search algorithms, heuristics can be used to guide the search process and prioritize promising states. Common heuristics for the 8-puzzle problem include the Manhattan distance and the misplaced tile heuristic.
4. Artificial intelligence: The 8-puzzle problem is a classic example of a problem that can be solved using AI techniques, such as search, optimization, and machine learning.
5. Complexity: The 8-puzzle problem is a combinatorial problem with a large search space, and finding an optimal solution can be computationally expensive. Therefore, complexity is a key consideration when designing algorithms to solve the problem.
6. Applications: The 8-puzzle problem has a wide range of real-world applications, such as game design, robotics, logistics, and cryptography. Understanding the problem and its solution techniques is therefore important for researchers and practitioners in these fields.

III. Literature Review

Intelligent search algorithms, such as best-first search, have been found to perform better than uninformed search algorithms, such as breadth-first search, depth-first search, and optimal search. Uninformed search algorithms required more nodes to be explored and raised compared to informed search algorithms that incorporated heuristics. In fact, the ratio of explored to raised nodes in uninformed search was low, while informed search required a smaller fraction of raised nodes to be explored in order to solve the puzzle. One way to compare the efficiency and effectiveness of these approaches is to analyse the distribution of the number of raised nodes during the search process. One approach for solving the 8-puzzle problem is the A* algorithm with the Manhattan distance heuristic. Another approach is the A* algorithm with the Misplaced Tiles heuristic. These two approaches can be compared in terms of the distribution of the number of raised nodes during the search process. To generate this comparison, a set of 8 puzzle problems of varying difficulty can be selected, and each problem can be solved using both the Manhattan distance and Misplaced Tiles heuristics. The number of raised nodes during the search process can be recorded for each approach, and the distribution of the number of raised nodes can be analysed using a histogram or a graph. The comparison may reveal that one approach consistently requires fewer raised nodes than the other, indicating that it is more efficient in terms of search time and computational resources. Additionally, the comparison may reveal that the distribution of the number of raised nodes varies depending on the specific problem instance, highlighting the importance of selecting an appropriate heuristic for each problem [3].

Out of the uninformed search techniques, Iterative Deepening Depth-First Search (DFS) is the most effective because it provides both the optimal solution and completeness. It has a completeness rate of 100%, whereas Depth-Limited DFS has a completeness rate of 97% [4].

The A* algorithm is utilized to ensure that the optimal solution is found. Different heuristics are analyzed to enable the algorithm to identify the optimal solution by exploring the minimum number of states possible [5].

The 8-puzzle is the most complex puzzle of its kind that can be entirely solved. Despite its simplicity, it has a vast combinatorial problem space of 9!/2 states. The N x N version of the 8-puzzle is considered NP-hard, which means that finding the optimal solution for this type of puzzle can take an exponential amount of time. In their follow-up experiments, they employed the 8-puzzle as a benchmark model to assess the advantages of node ordering systems in Iterative-Deepening A* (IDA*). An important finding from their study is that most IDA* implementations had worse performance than could be achieved by merely utilizing a basic random operator ordering method [6].

The objective of this research [2] is to create a program that can identify the quickest route that shoppers need to take in supermarkets. This will allow users to modify their shopping list and update the optimal path to save time by implementing the A* algorithm.

An algorithm B is considered optimal if no other search algorithm can perform the search process in less time or space, or by expanding fewer nodes, while maintaining the same quality of solution as algorithm A. Thus, if (Any Function) h(n) is a lower bound on h∗(n), then the solution provided by A* is valid. Additionally, this estimation suggests that any open node n may be very close to a desired goal node [1].

This paper [9] improved A* algorithm produces path-planning outcomes that are highly similar to the optimal solution. In contrast to the traditional A* algorithm, which can be affected by the complexities of the environment, the improved algorithm is more robust and resistant to interference.

The study [10] utilized the Manhattan heuristic function, which is more sophisticated than the Hamming heuristic, and through experiments, it was observed that using the Manhattan heuristic improved the spatial complexity of the A* algorithm compared to the Hamming heuristic. In two cases, optimal solutions were obtained with the minimum number of movements required. Theoretical and experimental evidence supports the claim that the Manhattan heuristic is more informed than the Hamming heuristic, making it a more efficient and streamlined method for the A* heuristic search.

This paper [11] determined complexity of the puzzle by the number of colors used. The game is completed when each row contains polyhedra of the same color and each column has a polygonal base with an increasing number of sides, also of the same color. The A* heuristic algorithm is employed in the game to determine the optimal solution.

Due to the complexity of the problem, the main challenge in solving it lies in the size of the search space, which necessitates the use of a heuristic search [12].

A method for solving the 8-puzzle problem has been suggested that aims to minimize memory usage while still achieving optimal results. All heuristics have been incorporated into the A* algorithm to ensure consistency, and the results have been presented. Comparing memory usage is only meaningful in the context of the A* algorithm, and thus, the memory usage of all heuristics has been compared. Additionally, a solvability check has been integrated at the start of each program to prevent runtime errors and program crashes. The IDA* algorithm has also been implemented but with only one heuristic, specifically the number of misplaced tiles [16].

This article [17] outlines how heuristic information derived from the problem domain can be integrated into a formal mathematical theory of graph searching. Additionally, it illustrates an optimality characteristic of a group of search strategies.

IV. METHODOLOGY

Breadth-First Search (BFS)

To solve the 8-puzzle problem using BFS, we will use a search tree to represent all possible moves from the current state. The root node represents the initial state, and each node represents a state of the board after a move has been made. The edges between nodes represent the moves that were made to get from one state to the next. We will use a queue to implement BFS. We start by adding the initial state to the queue. Then, we dequeue the first state, generate all possible moves from this state, and add the resulting states to the queue. We repeat this process until we find the goal state or until the queue is empty. Here is the Python code to solve the 8-puzzle problem using BFS:

```
1.  from queue import Queue
2.  def bfs(initial_state, goal_state):
3.  queue = Queue()
4.  queue.put(initial_state)
5.  visited = set()
6.  while not queue.empty():
7.  current_state = queue.get()
8.  if current_state == goal_state:
a.  return current_state
9.  visited.add(current_state)
10. for next_state in generate_states(current_state):
    a. if next_state not in visited:
    b. queue.put(next_state)
11. return None
12. def generate_states(state):
13. states = []
14. zero_pos = state.index(0)
15. if zero_pos not in [0, 1, 2]:
16. new_state = state[:]
17. new_state[zero_pos], new_state[zero_pos - 3] = new_state[zero_pos - 3], new_state[zero_pos]
18. states.append(new_state)
19. if zero_pos not in [0, 3, 6]:
20. new_state = state[:]
21. new_state[zero_pos], new_state[zero_pos - 1] = new_state[zero_pos - 1], new_state[zero_pos]
22. states.append(new_state)
23. if zero_pos not in [6, 7, 8]:
24. new_state = state[:]
```

```
ew_state[zero_pos], new_state[zero_pos + 3] = new_state[zero_pos + 3], new_state[zero_pos]
states.append(new_state)
if zero_pos not in [2, 5, 8]:
new_state = state[:]
new_state[zero_pos], new_state[zero_pos + 1] = new_state[zero_pos + 1], new_state[zero_pos]
states.append(new_state)
return states
```

The bfs function takes as input the initial state and the goal state, both represented as lists of 9 integers. It returns the goal state if it is found, or None if it is not. The generate_states function takes as input a state and returns a list of all possible states that can be reached by making one move from the current state. To test the algorithm, we can create an initial state and a goal state as follows:

```
initial_state = [1, 2, 3, 4, 5, 6, 0, 7, 8]
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

This represents an initial state where the 0 tile is in the bottom

Depth-First Search (DFS)

The 8-puzzle problem can also be solved using the Depth-First Search (DFS) algorithm. Like BFS, DFS is a well-known graph traversal algorithm. However, while BFS explores all nodes at a given depth before moving to the next depth, DFS explores as far as possible along each branch before backtracking. To solve the 8-puzzle problem using DFS, we start with the initial state and explore each possible move until we reach the goal state. We use a stack to keep track of the nodes we need to explore. We push the initial state onto the stack and continue exploring each possible move until we either find the goal state or the stack is empty. If we reach a dead-end, we backtrack by popping the last state from the stack and trying the next possible move. Here is the Python code to solve the 8-puzzle problem using DFS:

```
def dfs(initial_state, goal_state):
stack = [initial_state]
visited = set()
while stack:
current_state = stack.pop()
if current_state == goal_state:
return current_state
visited.add(current_state)
for next_state in generate_states(current_state):
if next_state not in visited:
    a. stack.append(next_state)
return None
```

The dfs function takes as input the initial state and the goal state, both represented as lists of 9 integers. It returns the goal state if it is found, or None if it is not. The generate_states function is the same as in the BFS algorithm. To test the algorithm, we can create an initial state and a goal state as follows:

```
initial_state = [1, 2, 3, 4, 5, 6, 0, 7, 8]
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

This represents an initial state where the 0 tile is in the bottom-right corner and the goal state where the 0 tile is in the bottom-right corner. Note that DFS does not guarantee that the solution found is optimal. It may find a solution that requires more moves than the optimal solution. However, for the 8-puzzle problem, the optimal solution can be found relatively easily using DFS, as the search space is not too large.

A* Search algorithm

The 8-puzzle problem can also be solved using the A* Search algorithm, which is a heuristic search algorithm that is widely used in many search problems. The basic idea of A* Search is to explore the search space by selecting the most promising nodes based on a heuristic function that estimates the distance to the goal state. The algorithm maintains a priority queue of nodes to explore, with the priority of a node determined by its estimated cost-to-goal plus its actual cost from the start node. To solve the 8-puzzle problem using A* Search, we need to define a heuristic function that estimates the distance to the goal state. A commonly used heuristic function for the 8-puzzle problem is the Manhattan distance, which is the sum of the distances of each tile from its correct position. The Manhattan distance is admissible, meaning that it never overestimates the distance to the goal state. Here is the Python code to solve the 8-puzzle problem using A* Search:

```
import heapq
def a_star(initial_state, goal_state):
queue = []
heapq.heappush(queue, (0, initial_state, []))
visited = set()
while queue:
_, current_state, path = heapq.heappop(queue)
if current_state == goal_state:
return path + [current_state]
visited.add(current_state)
for next_state in generate_states(current_state):
if next_state not in visited:
```

```
    a. priority = len(path) + 1 + manhattan_distance(next_state, goal_state)
    b. heapq.heappush(queue, (priority, next_state, path + [current_state]))
13. return None
14. def manhattan_distance(state, goal_state):
15. distance = 0
16. for i in range(3):
17. for j in range(3):
18. if state[3*i+j] != 0:
    a. x, y = divmod(goal_state.index(state[3*i+j]), 3)
    b. distance += abs(i-x) + abs(j-y)
19. return distance
```

The a_star function takes as input the initial state and the goal state, both represented as lists of 9 integers. It returns the path from the initial state to the goal state, as a list of states. The Manhattan distance function computes the Manhattan distance between two states, as described above. To test the algorithm, we can create an initial state and a goal state as follows:

```
initial_state = [1, 2, 3, 4, 0, 6, 7, 5, 8]
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

This represents an initial state where the 0 tile is in the middle and the goal state where the 0 tile is in the bottom-right corner. Note that A* Search guarantees that the solution found is optimal, meaning that it requires the minimum number of moves to reach the goal state. However, the execution time of A* Search can be slower than BFS or DFS, as it needs to compute the heuristic function for each node in the priority queue. However, by using an admissible heuristic function, such as the Manhattan distance, A* Search can be very efficient in practice, especially for large search spaces.

Iterative Deepening Search

Iterative Deepening Search (IDS) is a search algorithm that combines the benefits of depth-first search (DFS) and breadth-first search (BFS). It works by repeatedly performing DFS to a limited depth, and increasing the depth limit until a solution is found. This allows IDS to find a solution with minimal memory usage, while still ensuring completeness and optimality. To apply IDS to the 8-puzzle problem, we can perform a depth-limited DFS starting from the initial state, with the depth limit starting at 1. If the goal state is not found at the current depth limit, we increase the depth limit and perform DFS again from the initial state, repeating this process until the goal state is found. Here is the pseudocode for the IDS algorithm:

```
1. function IDS(initial_state, goal_state):
2. depth_limit = 1
3. while true:
4. result = DLS(initial_state, goal_state, depth_limit)
5. if result != "cutoff":
6. return result
7. depth_limit += 1
8. function DLS(current_state, goal_state, depth_limit):
9. if current_state == goal_state:
10. return current_state
11. if depth_limit == 0:
12. return "cutoff"
13. for move in possible_moves(current_state):
14. new_state = make_move(current_state, move)
15. result = DLS(new_state, goal_state, depth_limit - 1)
```

In the above code, possible_moves(current_state) returns the set of valid moves that can be made from the current state, and make_move(current_state, move) returns the new state obtained by making the given move from the current state.

Best First Search

Best First Search (BFS) is a search algorithm that explores the search space by expanding the most promising node, as determined by a heuristic function. BFS maintains a priority queue of the nodes to be expanded, sorted by their heuristic values. At each step, it expands the node with the lowest heuristic value, i.e., the node that is closest to the goal state. To apply BFS to the 8-puzzle problem, we can use the Manhattan distance as the heuristic function, which calculates the sum of the distances of each tile from its correct position in the goal state. We can represent each state as a node in the search space, with the cost of a node being the sum of the Manhattan distance and the depth of the node in the search tree. The search space is explored by expanding nodes in the priority queue, starting from the initial state and stopping when the goal state is found.

Here is the pseudocode for the BFS algorithm:

```
1. function BFS(initial_state, goal_state):
2. frontier = Priority Queue containing only the initial state
3. explored = empty set
4. while not frontier.empty():
5. current_state = frontier.pop()
6. if current_state == goal_state:
7. return current_state
8. explored.add(current_state)
9. for move in possible_moves(current_state):
```

```
10. new_state = make_move(current_state, move)
11. if new_state not in explored:
12. frontier.add(new_state)
13. return "failure"
```

In the above code, possible_moves(current_state) returns the set of valid moves that can be made from the current state, and make_move(current_state, move) returns the new state obtained by making the given move from the current state.

Hill Climbing Algorithm

Hill climbing search is a local search algorithm that starts with an initial solution and tries to improve it by making small changes to the solution. In the context of the 8-puzzle problem, hill climbing search can be used to move tiles on the board in a way that gets closer to the goal state. Here's an example of how hill climbing search can be used to solve the 8-puzzle problem: Start with an initial configuration of the board. Evaluate the current state by computing a heuristic function that estimates the distance to the goal state. One commonly used heuristic function is the sum of the Manhattan distances between the tiles and their goal positions. Generate a set of successor states by moving one tile in each of the four possible directions (up, down, left, or right). Evaluate each successor state using the heuristic function. Choose the successor state with the lowest heuristic value and repeat the process until a goal state is reached.

Steepest Ascent Hill Climbing Algorithm

Steepest Ascent Hill Climbing Algorithm is a local search algorithm that starts from an initial state and then moves to the best neighboring state at each step until it reaches a goal state or a local maximum. Here's an outline of how we can solve the 8-puzzle problem using Steepest Ascent Hill Climbing Algorithm: Initialize the current state to the initial state. Compute the heuristic value of the current state (for example, Manhattan distance or Hamming distance). Generate all possible neighboring states of the current state by sliding one tile into the empty space. Compute the heuristic value of each neighboring state. Select the neighboring state with the lowest heuristic value as the next current state. If the current state is the goal state, then stop and return the solution. If the current state is not the goal state, then set the current state to the selected neighboring state and repeat.

V. Results, Comparison and Discussion

The 8-puzzle problem is a well-known problem in AI that involves sliding numbered tiles on a 3x3 grid to reach a goal state. There are various approaches to solve this problem, including uninformed search algorithms (for example BFS, DFS), heuristic search algorithms (for example A*, IDA*), and iterative deepening algorithms (for example IDDFS). The time complexity of the Breadth-First Search algorithm is O(b^d), where b is the branching factor of the search tree (the number of possible moves from each state), and d is the depth of the goal state. In the case of the 8-puzzle problem, the branching factor is at most 4 (each tile can be moved in 4 directions), and the depth of the goal state is at most 31 (the number of moves required to reach the goal state from the initial state). Therefore, the time complexity of the algorithm is O(4^31), which is an extremely large number. In practice, the algorithm will not search the entire tree, as it is very likely to find the goal state before reaching the maximum depth. However, the worst-case time complexity gives an upper bound on the running time of the algorithm. The execution time of the algorithm depends on the size of the search space and the efficiency of the implementation. In general, the algorithm can be slow for large or complex problems. However, for the 8-puzzle problem, the search space is relatively small, and the implementation we have generated is efficient enough to solve the problem in a reasonable amount of time on modern computers. The execution time of the algorithm can vary depending on the initial state and the goal state. In the best case, the goal state can be reached in a few moves, and the algorithm will terminate quickly. In the worst case, the search space can be very large, and the algorithm can take a long time to find the goal state. However, for most random initial states and goal states, the algorithm should be able to find the solution within a reasonable amount of time.

The time complexity of the Depth-First Search (DFS) algorithm is O(b^m), where b is the branching factor of the search tree (the number of possible moves from each state), and m is the maximum depth of the search tree. In the case of the 8-puzzle problem, the branching factor is at most 4 (each tile can be moved in 4 directions), and the maximum depth of the search tree is not well-defined, as the search can continue indefinitely if there are cycles in the graph. However, in practice, the search usually terminates after a finite number of steps, as long as we keep track of the visited states to avoid revisiting the same state. The execution time of the DFS algorithm depends on the size of the search space and the efficiency of the implementation. In general, DFS can be faster than BFS in some cases, especially if the goal state is deep in the search tree, as DFS explores the tree deeply before backtracking. However, in other cases, DFS can be very slow, especially if the search space is very large or if there are many cycles in the graph. For the 8-puzzle problem, the execution time of the DFS algorithm can vary depending on the initial state and the goal state. In the best case, the goal state can be reached quickly, and the algorithm will terminate in a few steps. In the worst case, the search space can be very large or have many cycles, and the algorithm can take a long time to find the goal state. However, for most random initial states and goal states, the algorithm should be able to find the solution within a reasonable amount of time. In practice, the time complexity and execution time of the DFS algorithm can be improved by using some optimization techniques, such as iterative deepening or bidirectional search, which can reduce the search space and avoid revisiting the same states.

The time complexity of A* Search depends on the quality of the heuristic function used and the structure of the search space. If the heuristic function is admissible, meaning that it never overestimates the distance to the goal state, then A* Search is guaranteed to find the optimal solution, i.e., the solution with the minimum number of moves, in a finite amount of time. In the case of the 8-puzzle problem, the time complexity of A* Search can be expressed as O(b^d), where b is the branching factor of the search tree (the number of possible moves from each state) and d is the depth of the optimal solution. However, by using an admissible heuristic function, such as the Manhattan distance, the search space can be pruned, reducing the effective branching factor, and therefore the time complexity. In practice, A* Search can be much faster than BFS or DFS for large search spaces, as it can avoid exploring many unpromising paths. The execution time of the A* Search algorithm depends on the size of the search space, the quality of the heuristic function, and the efficiency of the implementation. In general, the execution time of A* Search can be slower than BFS or DFS, as it needs to compute the heuristic function for each node in the priority queue, which can be expensive for complex heuristic functions.

However, by using an efficient data structure, such as a binary heap or a Fibonacci heap, the priority queue operations can be performed in logarithmic time, reducing the overall execution time. For the 8-puzzle problem, the execution time of the A* Search algorithm can vary depending on the initial state and the goal state, as well as the quality of the heuristic function used. In general, the execution time should be reasonable for most random initial states and goal states, as long as the heuristic function is admissible and consistent. However, for some pathological cases, such as heavily constrained search spaces or poorly designed heuristic functions, the execution time can be prohibitively slow.

The time complexity of IDS is O(b^d), where b is the branching factor of the search tree and d is the depth of the optimal solution. However, in practice, IDS can be more efficient than BFS or DFS, as it can avoid exploring many unpromising paths by limiting the depth of the search. The execution time of IDS can depend on the depth of the optimal solution, the branching factor of the search tree, and the number of cutoffs that occur during the search. In general, IDS should be faster than BFS for large search spaces, but slower than DFS for small search spaces.

The time complexity of Best First Search depends on the size of the search space and the quality of the heuristic function used. If the heuristic function is admissible, Best First Search is guaranteed to find the optimal solution, i.e., the solution with the minimum number of moves, in a finite amount of time. However, the time complexity of Best First Search can be exponential in the worst case, as it may need to explore all possible paths in the search space. The execution time of Best First Search can depend on the size of the search space, the quality of the heuristic function, and the efficiency of the implementation. In general, BFS can be slower than other search algorithms, such as A* or IDS, as it needs to maintain a priority queue of all nodes in the search space, which can be memory-intensive and time-consuming to maintain. However, by using an efficient data structure, such as a binary heap or a Fibonacci heap, the priority queue operations can be performed in logarithmic time, reducing the overall execution time. For the 8-puzzle problem, the execution time of the Best First Search algorithm can vary depending on the initial state and the goal state, as well as the quality of the heuristic function used. In general, Best First Search should be reasonably fast for most random initial states and goal states, as long as the heuristic function is admissible and consistent. However, for some pathological cases, such as heavily constrained search spaces or poorly designed heuristic functions, the execution time can be prohibitively slow.

The time complexity of hill climbing search depends on the size of the state space and the quality of the heuristic function. In the case of the 8-puzzle problem, the state space has a size of 9! = 362,880 possible configurations. However, many of these configurations are unreachable from the initial state, and the effective branching factor is much lower than 4 (the number of possible moves for each tile). The execution time of hill climbing search also depends on the initial state and the quality of the heuristic function. In some cases, hill climbing search can find a solution quickly, while in others, it may get stuck in a local minimum and fail to find the optimal solution. In general, hill climbing search is a simple and efficient algorithm for solving the 8-puzzle problem, but it may not always find the optimal solution. More sophisticated algorithms like A* search can be used to improve the quality of the search and guarantee the optimality of the solution.

The time complexity of the Steepest Ascent Hill Climbing Algorithm depends on the size of the search space and the number of steps needed to find the solution. In the worst case, the algorithm may need to explore the entire search space, which has a size of 9! (About 362,880) states. However, in practice, the algorithm can usually find a solution much faster by exploiting the structure of the problem. The execution time of the algorithm also depends on the efficiency of the heuristic function and the quality of the initial state. A good heuristic function can greatly reduce the number of states that need to be explored, while a good initial state can provide a good starting point for the search. In summary, the Steepest Ascent Hill Climbing Algorithm can be used to solve the 8-puzzle problem, but its time complexity and execution time depend on the size of the search space, the efficiency of the heuristic function, and the quality of the initial state. Table 1 is showing comparison table of different approaches to solve the 8-puzzle problem in AI:

Table 1 comparison table of different approaches to solve the 8-puzzle problem in AI

Approach	Time Complexity	Space Complexity	Completeness	Optimality	Admissibility
Breadth-First Search (BFS)	O(B^D)	O(B^D)	Yes	Yes	Yes
Depth-First Search (DFS)	O(B^M)	O(BM)	No	No	-
Iterative Deepening DFS (IDDFS)	O(B^D)	O(BD)	Yes	Yes	Yes
A* search with Misplaced Tile heuristic	O(B^D)	O(B^D)	Yes	Yes	Yes
A* search with Manhattan Distance heuristic	O(B^D)	O(B^D)	Yes	Yes	Yes
Iterative Deepening A* (IDA*) with Manhattan Distance heuristic	O(B^D)	O(BD)	Yes	Yes	Yes
Best First Search with Manhattan Distance heuristic	O(B^M)	O(B^M)	No	No	Yes
Hill Climbing with Manhattan Distance heuristic	O(B^M)	O(B^M)	No	No	No

where: B/b(B or b): the branching factor (that is the number of possible moves), D/d(D or d): the depth of the goal state, M/m(M or m): the maximum depth of the search tree, brief explanation of the metrics used in the table are as follows; Time Complexity: the amount of time required to find a solution. Space Complexity: the amount of memory required to find a solution. Completeness:

whether or not the algorithm is guaranteed to find a solution if one exists. Optimality: whether or not the algorithm is guaranteed to find the optimal solution (that is the shortest path to the goal state). Admissibility: whether or not the heuristic function used in the algorithm is admissible (that is never overestimates the actual cost to reach the goal state)

Overall, A* search with Manhattan Distance heuristic and Iterative Deepening A* (IDA*) with Manhattan Distance heuristic remain the best approaches for solving the 8-puzzle problem in terms of both time and space complexity, completeness, optimality, and admissibility. Best First Search with Manhattan Distance heuristic is not complete or optimal, but it is admissible. Hill Climbing with Manhattan Distance heuristic is not complete, optimal, or admissible.

VI. Challenges, Limitations and Future work:

The 8-puzzle problem is a classical problem in artificial intelligence and involves finding the optimal sequence of moves to rearrange a board with 8 numbered tiles into a specific goal state. While significant progress has been made in solving this problem, there are still several challenges and limitations in this area of research.

Challenges

Complexity: The 8-puzzle problem is a combinatorial problem, and the number of possible states can be enormous. As the number of tiles on the board increases, the problem's complexity increases exponentially. Heuristics: Finding an optimal solution requires an effective heuristic that can guide the search algorithm to the solution. While many heuristics have been proposed, finding an optimal heuristic that works well in all situations is still a challenge. Local Optima: Many search algorithms can get stuck in local optima, where the algorithm finds a solution that is suboptimal but cannot find a better solution. Memory Requirements: Some algorithms require a large amount of memory to store the search tree, which can be a limitation when working with large problems.

Limitations

Determinism: The 8-puzzle problem assumes a deterministic environment, where the same move will always lead to the same state. However, in real-world applications, the environment may not be deterministic, which can limit the applicability of the solution. Domain-Specific: The 8-puzzle problem is a domain-specific problem that does not necessarily generalize to other problem domains. Single Agent: The 8-puzzle problem is a single-agent problem, and solutions designed for this problem may not work well for multi-agent problems.

Future Work

Developing better heuristics that can guide the search algorithm to find optimal solutions. Developing algorithms that can handle non-deterministic environments. Designing algorithms that can handle multi-agent problems. Extending the 8-puzzle problem to larger board sizes and other problem domains. Exploring the use of machine learning techniques to improve the performance of search algorithms for this problem.

VII. Conclusion

The 8-puzzle problem has been extensively researched in the field of AI, and a variety of algorithms have been proposed to solve it. Overall, the most successful approaches are those that use heuristics to guide the search towards the goal state, such as A* search and its variants. In particular, A* search with the Manhattan Distance heuristic has been shown to be highly effective at solving the 8-puzzle problem, both in terms of finding optimal solutions and doing so in a reasonable amount of time and memory. However, while these approaches are highly effective for the 8-puzzle problem, they may not be as effective for other similar problems with larger search spaces or different goal states. As such, ongoing research in this area continues to explore new algorithms and heuristics that can be applied to a wide range of puzzle problems, as well as ways to optimize existing algorithms for specific use cases. In summary, the 8-puzzle problem serves as an important benchmark problem in AI, and ongoing research in this area continues to advance our understanding of how to effectively solve complex puzzles and search problems.

VIII. Acknowledgment

We would like to express our sincere gratitude to all those who have contributed to this research project. We would like to extend our thanks to Dr. Rakesh Vanzara Associate Professor and Dean UVPCE, Ganpat University Mehsana, Dr. Paresh Solanki Associate Professor and HOD Computer Engineering UVPCE, Ganpat University Mehsana, Dr. Devang Pandya Associate Professor and HOD Information Technology UVPCE, Ganpat University Mehsana. Their expertise and insights were invaluable in helping us to successfully complete this work. This work would not have been possible without their generosity. Finally, we would like to extend our gratitude to all the advisers, their cooperation and support were crucial in making this research project a success. Thank you all for your invaluable contributions and support.

References

1. Chowdhary, K. R. (2020). Fundamentals of artificial intelligence. Springer Nature. doi.org/10.1007/978-81-322-3972-7_9
2. A. H. D. Ada, I. P. Q. Cortez, X. A. S. Juvida, N. B. Linsangan and G. V. Magwili, "Dynamic Route Optimization using A* Algorithm with Heuristic Technique for a Grocery Store," 2019 IEEE 11th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment, and Management (HNICEM), Laoag, Philippines, 2019, pp. 1-6, doi: 10.1109/HNICEM48295.2019.9072759
3. Hidayat, W., Susanti, F. & Wijaya, D. R. (2021). A Comparative Study of Informed and Uninformed Search Algorithm to Solve Eight-Puzzle Problem. Journal of Computer Science, 17(11), 1147-1156. https://doi.org/10.3844/jcssp.2021.1147.1156
4. A. K. Mishra and P. C. Siddalingaswamy, "Analysis of tree based search techniques for solving 8-puzzle problem," 2017 Innovations in Power and Advanced Computing Technologies (i-PACT), Vellore, India, 2017, pp. 1-5, doi: 10.1109/IPACT.2017.8245012.
5. Daniel R. Kunkle, Solving the 8 Puzzle in a Minimum Number of Moves: An Application of the A* Algorithm. Online available at: https://web.mit.edu/6.034/wwwbob/EightPuzzle.pdf
6. Alexander Reinefeld, Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in IDA *. Online available at: https://www.ijcai.org/Proceedings/93-1/Papers/035.pdf
7. https://www.ics.uci.edu/~kkask/Fall-2017%20CS271/index.html

8. http://aima.cs.berkeley.edu/
9. Chen, C., Cai, J., Wang, Z., Chen, F., & Yi, W. (2020). An improved A* algorithm for searching the minimum dose path in nuclear facilities. Progress in Nuclear Energy, 126, 103394. doi.org/10.1016/j.pnucene.2020.103394
10. Iordan, A. E. (2018). A comparative study of the A* heuristic search algorithm used to solve efficiently a puzzle game. In IOP Conference Series: Materials Science and Engineering (Vol. 294, No. 1, p. 012049). IOP Publishing. doi.org/10.1088/1757-899X/294/1/012049
11. Ertel W 2011 Introduction to Artificial Intelligence, Springer-Verlag, Berlin
12. Shi Z 2011 Advanced Artificial Intelligence, Word Scientific Publishing
13. https://github.com/topics/8-puzzle-problem?l=python
14. http://www.sfu.ca/~tjd/310summer2019/a1.html
15. https://courses.cs.washington.edu/courses/cse473/12au/slides/lect3.pdf
16. http://ethesis.nitrkl.ac.in/5575/1/110CS0081-1.pdf
17. P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968, doi: 10.1109/TSSC.1968.300136.
18. https://en.wikipedia.org/wiki/Sliding_puzzle
19. https://javascript.plainenglish.io/solving-8-puzzle-exploring-search-options-2e446e29d21
20. https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search
21. https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search
22. https://javascript.plainenglish.io/solving-8-puzzle-exploring-search-options-2e446e29d21
23. https://web.mit.edu/6.034/wwwbob/EightPuzzle.pdf