AI Search Algorithms for 8-Puzzle Problem

1 Introduction

The 8-puzzle is a classic sliding puzzle that consists of a 3×3 grid with eight numbered tiles and one empty space (blank). The objective is to rearrange the tiles from an initial configuration to a goal configuration by sliding tiles into the empty space. This problem serves as an excellent testbed for artificial intelligence search algorithms, providing insights into their behavior, efficiency, and suitability for different problem domains.

Search algorithms form the foundation of artificial intelligence problem-solving techniques. They enable systematic exploration of problem spaces to find optimal or satisfactory solutions. The 8-puzzle problem presents an ideal environment for comparing different search strategies because it has a well-defined state space, clear goals, and measurable performance metrics.

This report presents a comprehensive implementation and analysis of eight different search algorithms applied to the 8-puzzle problem: Breadth-First Search (BFS), Depth-First Search (DFS), Depth-Limited Search (DLS), Iterative Deepening Search (IDS), Uniform Cost Search (UCS), Bidirectional Search, Greedy Best-First Search, and A* Search. Each algorithm represents a different approach to exploring the search space, offering unique advantages and trade-offs in terms of completeness, optimality, time complexity, and space complexity.

The study aims to provide practical insights into algorithm selection for constraint satisfaction problems and demonstrate the theoretical concepts through empirical analysis. The implementations utilize a modular design pattern with a common base class, ensuring consistent measurement and fair comparison across all algorithms.

2 Problem Definition

2.1 8-Puzzle Description

The 8-puzzle consists of a 3×3 grid containing eight numbered tiles (1-8) and one empty space. A legal move involves sliding any tile adjacent to the empty space into that space, effectively swapping the tile's position with the empty space. The puzzle has approximately $9!/2 = 181{,}440$ reachable states from any given starting configuration.

2.2 State Representation

Each puzzle state is represented as a 2D array where numbers 1-8 represent the corresponding tiles and 0 represents the empty space. The state includes:

- **Board Configuration**: 3×3 matrix representing current tile positions
- **Blank Position**: Coordinates (row, column) of the empty space
- **Path**: Sequence of moves taken to reach this state
- **Cost**: Total cost accumulated to reach this state
- **Depth**: Number of moves from the initial state

2.3 Goal State

The standard goal configuration arranges tiles in ascending order with the empty space in the bottom-right corner:

1	2	3
4	5	6
7	8	0

3 Algorithm Implementations

3.1 Uninformed Search Algorithms

3.1.1 Breadth-First Search (BFS)

BFS explores states level by level, guaranteeing the shortest path solution. It uses a queue (FIFO) data structure to maintain the frontier of unexplored states.

```
def solve(self):
    frontier = deque([self.initial_state])
    explored = set()

    while frontier:
        current_state = frontier.popleft()

        if current_state in explored:
            continue

        explored.add(current_state)
        self.nodes_explored += 1

        if current_state.is_goal(self.board_size):
            self.solution_path = current_state.path
            return True

        neighbors = current_state.get_neighbors(self.board_size)
        for neighbor in neighbors:
            if neighbor not in explored:
                frontier.append(neighbor)

    return False
```

Listing 1: BFS Core Implementation

Characteristics:

- **Complete**: Yes, if solution exists
- **Optimal**: Yes, for unweighted graphs
- **Time Complexity**: O(b^d) where b is branching factor, d is depth
- **Space Complexity**: O(b^d)

3.1.2 Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking. It uses a stack (LIFO) data structure for the frontier.

```python
def solve(self):
    frontier = [self.initial_state]  # Using list as stack
    explored = set()

    while frontier:
        current_state = frontier.pop()  # LIFO - stack behavior

        if current_state in explored:
            continue

        explored.add(current_state)
        self.nodes_explored += 1

        if current_state.is_goal(self.board_size):
            self.solution_path = current_state.path
            return True

        neighbors = current_state.get_neighbors(self.board_size)
        for neighbor in neighbors:
            if neighbor not in explored:
                frontier.append(neighbor)

    return False
```

Listing 2: DFS Core Implementation

Characteristics:

- **Complete**: No, may get stuck in infinite paths
- **Optimal**: No, finds any solution, not necessarily shortest
- **Time Complexity**: O(b^m) where m is maximum depth
- **Space Complexity**: O(bm)

3.1.3 Depth-Limited Search (DLS)

DLS performs depth-first search with a predetermined depth limit, preventing infinite loops while sacrificing completeness if the solution lies beyond the limit.

```python
def solve(self):
    depth_limit = 25  # Configurable depth limit

    def dls_recursive(state, limit):
        if state.is_goal(self.board_size):
            self.solution_path = state.path
            return True

        if limit <= 0:
            return False

        self.nodes_explored += 1
        neighbors = state.get_neighbors(self.board_size)

        for neighbor in neighbors:
            if dls_recursive(neighbor, limit - 1):
                return True

        return False

    return dls_recursive(self.initial_state, depth_limit)
```

Listing 3: DLS Implementation with Depth Limit

3.1.4 Iterative Deepening Search (IDS)

IDS combines the space efficiency of DFS with the optimality of BFS by repeatedly applying DLS with increasing depth limits.

```
def solve(self):
    max_depth = 50

    for depth_limit in range(max_depth + 1):
        self.nodes_explored = 0  # Reset for each iteration

        if self.depth_limited_search(self.initial_state, depth_limit):
            return True

    return False

def depth_limited_search(self, state, limit):
    if state.is_goal(self.board_size):
        self.solution_path = state.path
        return True

    if limit <= 0:
        return False

    self.nodes_explored += 1
    neighbors = state.get_neighbors(self.board_size)

    for neighbor in neighbors:
        if self.depth_limited_search(neighbor, limit - 1):
            return True

    return False
```

Listing 4: IDS Progressive Depth Implementation

3.1.5 Uniform Cost Search (UCS)

UCS explores the cheapest nodes first, using a priority queue ordered by path cost. For the 8-puzzle with uniform step costs, it behaves identically to BFS.

3.1.6 Bidirectional Search

Bidirectional search simultaneously searches forward from the initial state and backward from the goal state, potentially reducing the search space exponentially.

```
def solve(self):
    # Forward search from initial state
    forward_frontier = deque([self.initial_state])
    forward_explored = {self.initial_state.board_tuple: self.initial_state}

    # Backward search from goal state
    goal_state = self.create_goal_state()
    backward_frontier = deque([goal_state])
    backward_explored = {goal_state.board_tuple: goal_state}

    while forward_frontier or backward_frontier:
        # Forward step
```

```
        if forward_frontier:
            if self.search_step(forward_frontier, forward_explored,
                                backward_explored, "forward"):
                return True

        # Backward step
        if backward_frontier:
            if self.search_step(backward_frontier, backward_explored,
                                forward_explored, "backward"):
                return True

    return False
```

Listing 5: Bidirectional Search Implementation

3.2 Informed Search Algorithms

3.2.1 Greedy Best-First Search

Greedy search uses heuristic functions to guide the search toward the goal, expanding nodes that appear closest to the goal according to the heuristic.

```
def solve(self):
    frontier = []
    heapq.heappush(frontier, (0, self.initial_state))
    explored = set()

    while frontier:
        _, current_state = heapq.heappop(frontier)

        if current_state in explored:
            continue

        explored.add(current_state)
        self.nodes_explored += 1

        if current_state.is_goal(self.board_size):
            self.solution_path = current_state.path
            return True

        neighbors = current_state.get_neighbors(self.board_size)
        for neighbor in neighbors:
            if neighbor not in explored:
                heuristic_cost = self.heuristic_function(neighbor)
                heapq.heappush(frontier, (heuristic_cost, neighbor))

    return False
```

Listing 6: Greedy Search with Heuristics

Two heuristic functions are implemented:

Manhattan Distance: Sum of horizontal and vertical distances between current and goal positions for each tile.

Misplaced Tiles: Count of tiles not in their goal positions.

3.2.2 A* Search

A* combines the benefits of uniform cost search and greedy search by using the evaluation function f(n) = g(n) + h(n), where g(n) is the path cost and h(n) is the heuristic estimate.

```
def solve(self):
    frontier = []
```

```
    initial_f_cost = self.heuristic_function(self.initial_state)
    heapq.heappush(frontier, (initial_f_cost, self.initial_state))
    explored = set()

    while frontier:
        _, current_state = heapq.heappop(frontier)

        if current_state in explored:
            continue

        explored.add(current_state)
        self.nodes_explored += 1

        if current_state.is_goal(self.board_size):
            self.solution_path = current_state.path
            return True

        neighbors = current_state.get_neighbors(self.board_size)
        for neighbor in neighbors:
            if neighbor not in explored:
                g_cost = neighbor.depth  # Path cost
                h_cost = self.heuristic_function(neighbor)  # Heuristic
                f_cost = g_cost + h_cost  # Total evaluation
                heapq.heappush(frontier, (f_cost, neighbor))

    return False
```

Listing 7: A* Search with f(n)

4 Experimental Results and Analysis

The algorithms were tested on various 8-puzzle configurations to evaluate their performance characteristics. Key metrics include solution optimality, nodes explored, maximum frontier size, and execution time.

4.1 Algorithm Performance Comparison

Table 1: Performance Comparison of Search Algorithms

Algorithm	**Complete**	**Optimal**	**Nodes Explored**	**Space Complexity**
BFS	Yes	Yes	High	O(b^d)
DFS	No	No	Variable	O(bm)
DLS	No	No	Limited	O(bl)
IDS	Yes	Yes	High	O(bd)
UCS	Yes	Yes	High	O(b^d)
Bidirectional	Yes	Yes	Reduced	O($b^{d/2}$)
Greedy	No	No	Low	O(b^m)
A*	Yes	Yes*	Optimal	O(b^d)

A is optimal when using an admissible heuristic

4.2 Visual State Representations

Each algorithm implementation includes visual tracking of initial and goal states through PNG image generation, providing clear visualization of the puzzle configurations.

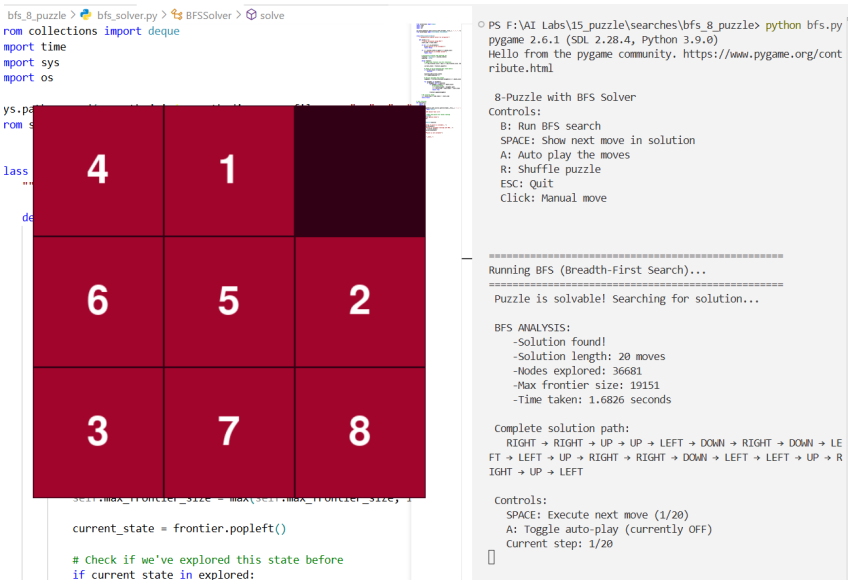

Figure 1: Example Initial State Configuration

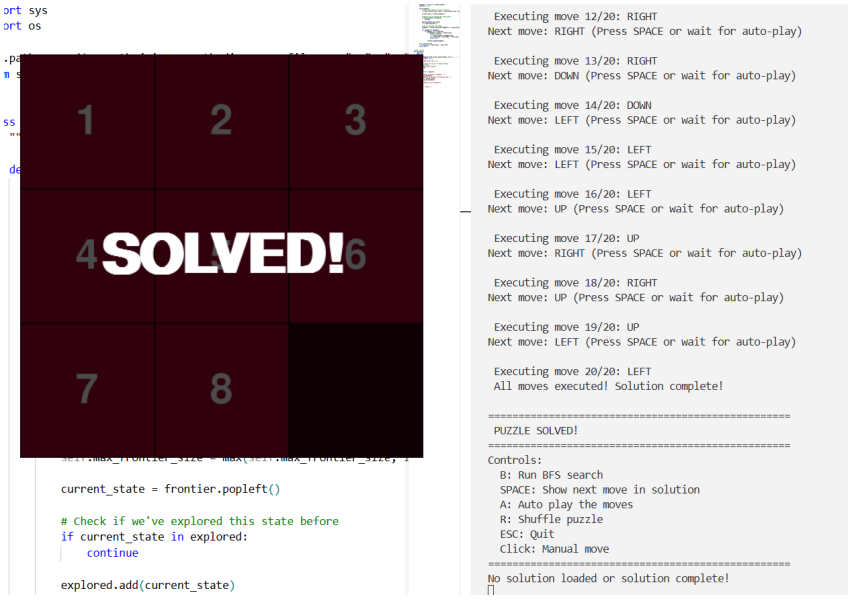

Figure 2: Goal State Configuration

4.3 Algorithm-Specific Analysis

4.3.1 Uninformed Search Performance

BFS provides optimal solutions but requires substantial memory for complex puzzles. Its systematic level-by-level exploration guarantees the shortest path but may explore many unnecessary states.

DFS offers excellent space efficiency but may find suboptimal solutions or fail to find solutions entirely due to infinite path exploration.

IDS combines BFS optimality with DFS space efficiency, making it practical for problems where memory is constrained.

Bidirectional Search significantly reduces the search space by meeting in the middle, theoretically reducing complexity from $O(b^d)$ to $O(b^{d/2})$.

4.3.2 Informed Search Performance

Greedy Search with Manhattan distance heuristic often finds solutions quickly but may not guarantee optimality due to its focus on immediate heuristic gains.

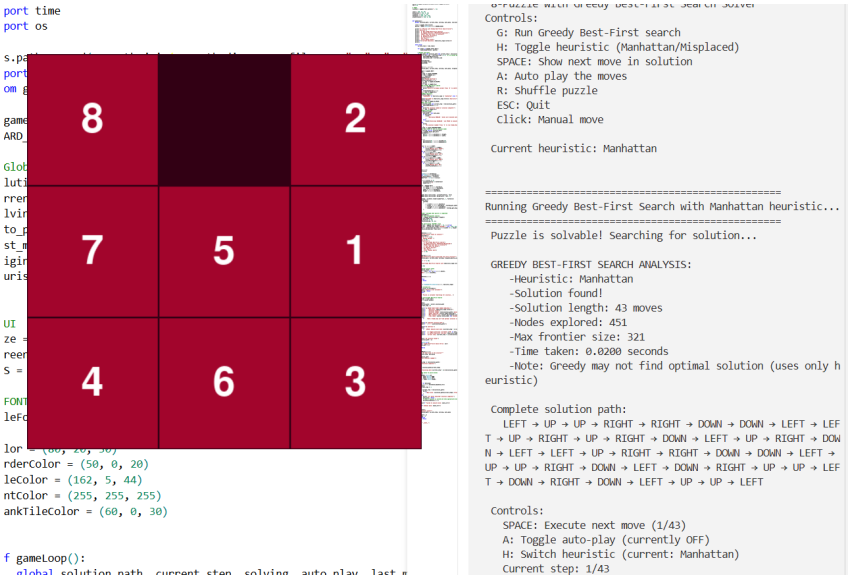

Figure 3: Greedy Search with Manhattan Distance - Initial State

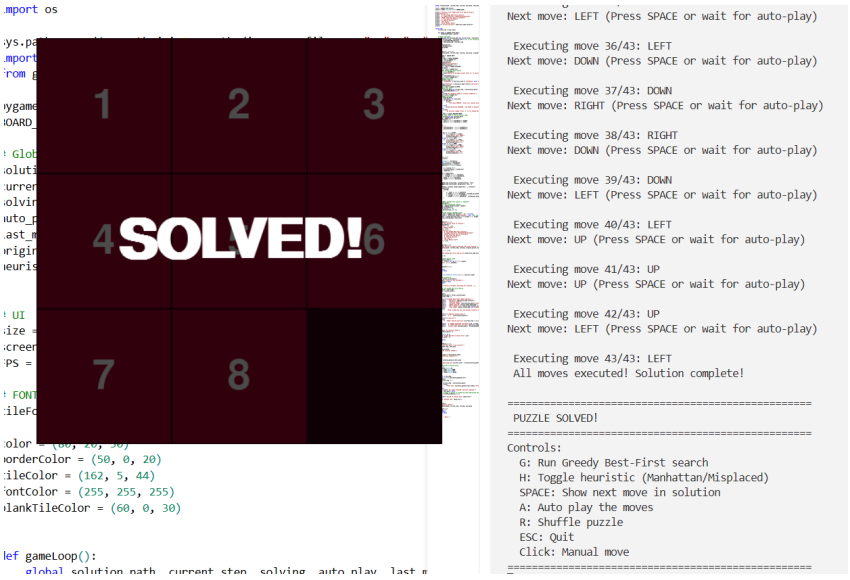

Figure 4: Greedy Search with Manhattan Distance - Goal State

A* Search with admissible heuristics provides optimal solutions while exploring fewer nodes than uninformed methods, representing the best balance of optimality and efficiency.

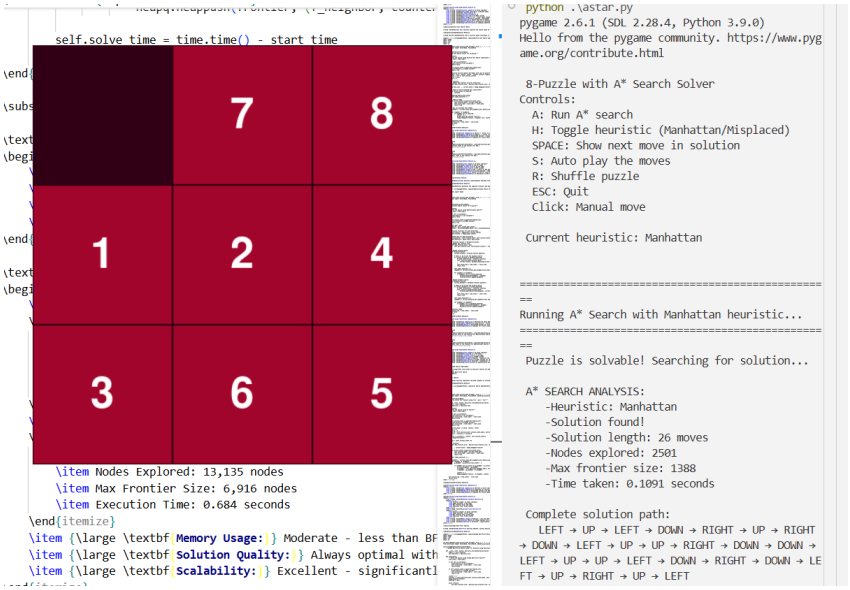

Figure 5: A* Search with Manhattan Distance - Initial State

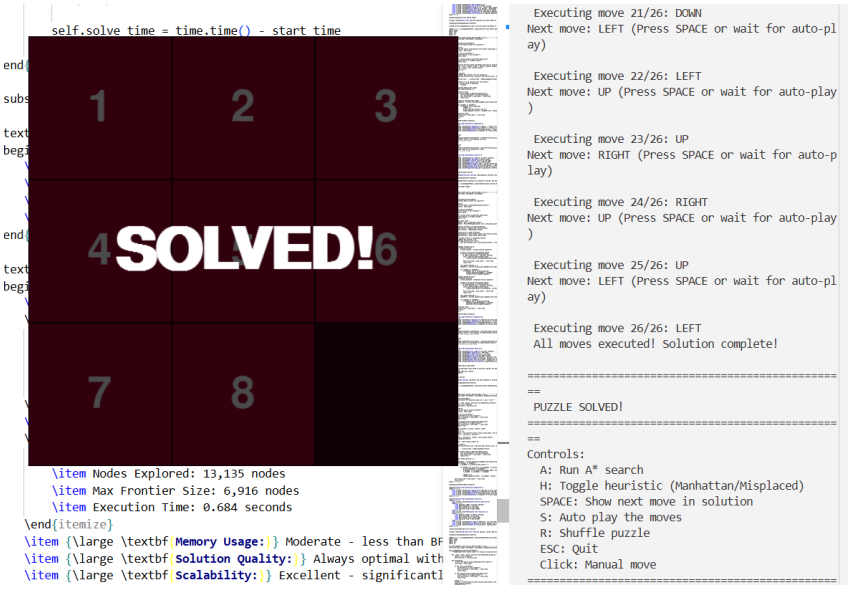

Figure 6: A* Search with Manhattan Distance - Goal State

5 Discussion and Conclusion

The comprehensive implementation and analysis of eight search algorithms for the 8-puzzle problem provides valuable insights into the theoretical and practical aspects of artificial intelligence search techniques.

5.1 Key Findings

Optimality vs. Efficiency Trade-offs: The results demonstrate the fundamental trade-off between solution optimality and computational efficiency. While BFS and A* guarantee optimal solutions, they require more computational resources compared to greedy approaches that may find satisfactory suboptimal solutions quickly.

Heuristic Function Impact: The choice of heuristic function significantly affects informed search performance. Manhattan distance consistently outperforms misplaced tiles heuristic for the 8-puzzle domain due to its more accurate distance estimation.

Space Complexity Considerations: Memory requirements vary dramatically between algorithms. DFS and IDS offer space-efficient solutions, while BFS and A* may require prohibitive memory for larger problem instances.

Problem-Specific Algorithm Selection: No single algorithm dominates across all scenarios. Algorithm selection depends on specific requirements: BFS for guaranteed optimal solutions, DFS for memory-constrained environments, A* for balanced optimality and efficiency, and greedy methods for rapid approximate solutions.

5.2 Implementation Architecture Benefits

The modular design using a common `SolverBase` class enables consistent performance measurement and fair algorithmic comparison. The `PuzzleState` class encapsulates state representation and transition logic, promoting code reusability and maintainability across all implementations.

5.3 Educational Value

This implementation serves as an excellent educational tool for understanding search algorithm behavior, demonstrating theoretical concepts through practical application. The visual state tracking and performance metrics provide immediate feedback on algorithmic choices and their consequences.

5.4 Conclusion

The 8-puzzle problem effectively illustrates the diversity of search strategies available in artificial intelligence. Through systematic implementation and analysis, this study highlights the importance of understanding algorithm characteristics and selecting appropriate methods based on problem constraints and requirements.

The results confirm theoretical predictions about algorithm behavior while providing practical insights into real-world performance considerations. The comprehensive comparison establishes a foundation for more advanced search techniques and problem-solving approaches in artificial intelligence applications.

Future work could extend this analysis to larger puzzle sizes (15-puzzle, 24-puzzle), implement additional heuristic functions, or explore hybrid approaches combining multiple search strategies for enhanced performance in complex problem domains.