Search Algorithm Evaluation System on 3*3 Eight puzzles

Submitted to: Hang Ma
Submitted by: Arthur Lee, psl10
Yixuan Liu, yla357
Arshdeep Kaur, aka232
Submitted on: December 17th, 2024

Table of Content:

Introduction:

The sliding puzzle problem is a classic problem in the field of artificial intelligence and computer science that has fascinated people for decades. This problem has been widely studied as it serves as an excellent benchmark for evaluating search algorithms, such as Breadth-First Search (BFS), A*, and IDA*. Beyond its theoretical importance, solving the sliding puzzle provides insights into heuristic search strategies, and optimization problems, which are fundamental concepts in AI and combinatorial problem solving.

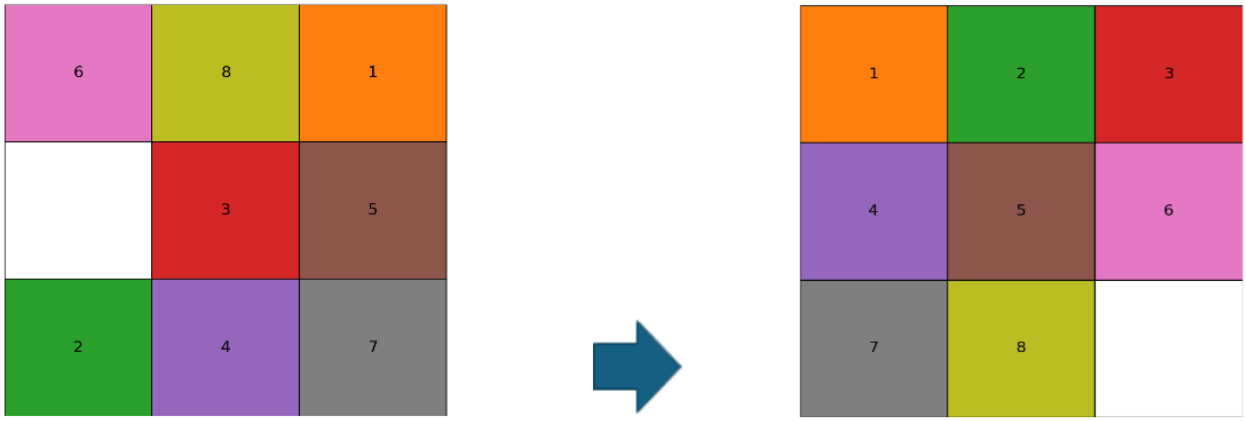


Fig 1: 8 Puzzle problem example Initial State and Goal State

The sliding puzzle problem can be defined as follows:

1. **State Space**: A given puzzle consists of n^2 – 1 numbered tile and 1 empty space, n here represents the dimension of puzzle grid (n * n). The state space of the sliding puzzle problem consists of all possible arrangements of the tiles and the empty space on the n×n grid.
2. **Initial State**: The initial state is a specific configuration of the tiles.
3. **Goal State:** The goal state is tiles ordered in ascending numerical sequence with the empty space in the bottom-right corner.
4. **Transition Function**: A transition T transforms a state S into a new state S' by swapping the empty space at position (i,j) with a valid neighboring tile at (i',j').
5. **Heuristic Functions**: A function that estimates how close a state is to a goal. It is designed for a particular search problem.
6. **Valid Moves:** A valid move consists of sliding one of the tiles adjacent to the empty space into the empty space.

For an 8-puzzle problem, we have 181, 440 reachable states (9! / 2) but as we move on to the 15-puzzle problem the number of reachable states increase to 10,461, 394, 944, 000 (16! / 2). This exponential growth in the number of states, known as state space explosion, becomes more pronounced as we increase n for the puzzle. Such growth poses significant challenges in searching for solutions, requiring more sophisticated algorithms and heuristics to efficiently navigate the exponentially increasing state space. In this report we aim to investigate the performance of various search algorithms for the 3 * 3 Eight-puzzle problem. The central focus is to explore how algorithmic strategies and heuristics influence the efficiency and effectiveness of solving different problem instances.

Implementation:

1. Framework

The project is organized into three main sections: Utilities, Algorithms, and Main. Each section plays a distinct role in solving and analyzing sliding puzzle problems. There will be three main sections: Utilities, Algorithms, main.

In **Utilities**, the puzzle_solver reads and solves puzzles from a file using a specified algorithm. The puzzle_generator creates solvable puzzles of a given size. The visualization module displays the solving process, and the comparison module evaluates algorithm performance and outputs results to a text file.

The **Algorithms** section includes a variety of search methods like A*, BFS, DFS, Bidirectional Search, Greedy Best-First, IDA* and IDDFS. It offers heuristics such as Manhattan distance, misplaced tiles, linear conflict, non-additive PDB, disjoint PDB, symmetry and dual lookup.

The **Main** section is controlled via the main.py script with three command-line options: solving a specific puzzle with a particular algorithm, testing all algorithms on one puzzle, or generating and analyzing a batch of puzzles. This setup allows for flexibility in testing and comparing different search strategies across varying puzzle complexities.

2. Uninformed Search

Uninformed Searches, also known as Blind searches, refer to search algorithms that explore the search space without any prior knowledge about the goal state. These algorithms systematically search through states but do not use heuristics to guide their search. To prevent excessive runtime, the **timeout** for each algorithm is dynamically adjusted based on the size and complexity of the puzzle, ensuring that resources are managed efficiently while still allowing sufficient time for the algorithm to find a solution. The following algorithms were tested by us for comparison.

1) Breadth First Search (BFS):

BFS explores all nodes at current depth before moving on to the next level and it uses a queue (First In First Out structure) to keep track of nodes to be explored. BFS is complete and guarantees optimal solution if one exists for uniform edge cost. BFS is widely used for finding the shortest path, but it requires a large memory for large problem spaces which makes it less practical for problems with large state space. The pseudocode for BFS is shown in figure 2.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Fig 2: Pseudocode for BFS [1]

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

Fig 3: Pseudocode for DFS [3]

2) Depth First Search (DFS):

DFS explores as far as possible along each branch before backtracking and it uses stack (First In Last Out structure) to keep track of nodes to be explored. DFS is not complete and does not guarantee an optimal solution if one exists. DFS uses low memory but the fact that it explores as far as possible along a branch makes it a bad choice for deep or infinite search spaces. The pseudocode for DFS is shown in figure 3.

3) Bidirectional Search:

Bidirectional Search expands from both the start and goal states, meeting at a common state. We used BFS for both forward and backward searches. For final path reconstruction we combine paths from both directions at the meeting point to form the solution. This bidirectional search helps reduce the search space significantly, making the algorithm more efficient as compared to single direction BFS. Bidirectional Search is both complete and optimal if a solution exists for uniform edge cost. The pseudocode for Bidirectional Search is shown in figure 4.

```
BIDIRECTIONAL_SEARCH
 1   Q_I.Insert(x_I) and mark x_I as visited
 2   Q_G.Insert(x_G) and mark x_G as visited
 3   while Q_I not empty and Q_G not empty do
 4       if Q_I not empty
 5           x ← Q_I.GetFirst()
 6           if x = x_G or x ∈ Q_G
 7               return SUCCESS
 8           forall u ∈ U(x)
 9               x' ← f(x, u)
10               if x' not visited
11                   Mark x' as visited
12                   Q_I.Insert(x')
13               else
14                   Resolve duplicate x'
15       if Q_G not empty
16           x' ← Q_G.GetFirst()
17           if x' = x_I or x' ∈ Q_I
18               return SUCCESS
19           forall u^-1 ∈ U^-1(x')
20               x ← f^-1(x', u^-1)
21               if x not visited
22                   Mark x as visited
23                   Q_G.Insert(x)
24               else
25                   Resolve duplicate x
26   return FAILURE
```

Fig 4: Pseudocode for Bidirectional Search [7]

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff_occurred? ← false
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      result ← RECURSIVE-DLS(child, problem, limit − 1)
      if result = cutoff then cutoff_occurred? ← true
      else if result ≠ failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

Fig 5: Pseudocode for ID DFS [1]

4) Iterative Deepening Depth First Search (IDDFS):

IDDFS combines the space efficiency of Depth-First Search (DFS) with the completeness and optimality of Breadth-First Search (BFS). It performs repeated depth-limited DFS with an increasing depth limit until the goal is found. IDDFS is complete and guarantees optimal solution if one exists. IDDFS is useful when the depth of the solution is unknown as it avoids excessive memory consumption compared to BFS. Even though the nodes are visited multiple times this overhead is often not significant because the majority of the computational effort is spent on the deepest level of the search tree. But IDDFS can become inefficient where the solution depth is very large. IDDFS is complete and optimal if the edge costs are uniform. If the costs are non-uniform IDDFS may not be optimal as it does not take costs into account. The pseudocode for IDDFS is shown in figure 5.

3. Informed Search

Also known as heuristic search, Informed search refers to a category of search algorithms that use domain specific knowledge to make the search process more efficient. These algorithms use heuristic function h(n), to estimate the cost or distance from a given state n to the goal state. These algorithms are generally faster than uninformed search algorithms. To prevent excessive runtime, the **timeout** for each algorithm is dynamically adjusted based on the size and complexity of the puzzle, ensuring that resources are managed efficiently while still allowing sufficient time for the algorithm to find a solution. The following algorithms were tested by us for comparison.

1) A Search:*

A * search uses a priority queue to explore states with the lowest estimated total cost f(n)=g(n)+h(n), where g(n) is the actual cost to reach the current node n from start node and h(n) is the heuristic function. A* is complete but is optimal only if h(n) is admissible and consistent. The performance depends on the quality of heuristic. The pseudocode for Bidirectional Search is shown in figure 6.

```
      // A*
1:    initialize the open list
2:    initialize the closed list
3:    put the starting node on the open list (you can leave its f at zero)
-
4:    while the open list is not empty
5:        find the node with the least f on the open list, call it "q"
6:        pop q off the open list
7:        generate q's 8 successors and set their parents to q
8:        for each successor
9:            if successor is the goal, stop the search
10:           successor.g = q.g + distance between successor and q
11:           successor.h = distance from goal to successor
12:           successor.f = successor.g + successor.h
-
13:           if a node with the same position as successor is in the OPEN list \
-                 which has a lower f than successor, skip this successor
14:           if a node with the same position as successor is in the CLOSED list \
-                 which has a lower f than successor, skip this successor
15:           otherwise, add the node to the open list
16:       end
17:       push q on the closed list
18:   end
```

Fig 6: A Search Pseudocode [4]*

```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Fig 7: Greedy Best First Search Pseudocode [5]

2) Greedy Best First Search:

Greedy Best First Search uses a priority queue to explore states with the lowest heuristic cost h(n), it does not consider the g(n) which is the actual cost to reach the current node n from start node. It does not guarantee completeness or optimality. If the h(n) is poorly designed, the algorithm may follow misleading paths and perform poorly. The pseudocode for Bidirectional Search is shown in figure 7.

3) Iterative Deepening A Search:*

Iterative Deepening A * (IDA*) Search is a memory-efficient informed search algorithm that combines the concepts of Iterative Deepening Depth-First Search (IDDFS) and A* Search. During each iteration, the algorithm prunes branches where the total cost, f(n)=g(n)+h(n), exceeds a specified threshold. Initially, the threshold is set to the estimated cost of the start state. For subsequent iterations, the threshold is updated to the smallest f(n) value that exceeded the previous threshold. This process continues until the goal is reached or no solution is found. The pseudocode for Bidirectional Search is shown in figure 8.

```
root=initial node;
Goal=final node;
function IDA*()                          //Driver function
{
    threshold=heuristic(Start);
    while(1)          //run for infinity
    {
        integer temp=search(Start,0,threshold); //function search(node,g score,threshold)
        if(temp==FOUND)                  //if goal found
            return FOUND;
        if(temp== ∞)                     //Threshold larger than maximum possible f value
            return;                      //or set Time limit exceeded
        threshold=temp;
    }
}
```

```
function Search(node, g, threshold)          //recursive function
{
    f=g+heuristic(node);
    if(f>threshold)          //greater f encountered
        return f;
    if(node==Goal)           //Goal node found
        return FOUND;
    integer min=MAX_INT;     //min= Minimum integer
    foreach(tempnode in nextnodes(node))
    {
        //recursive call with next node as current node for depth search
        integer temp=search(tempnode,g+cost(node,tempnode),threshold);
        if(temp==FOUND)          //if goal found
        return FOUND;
        if(temp<min)     //find the minimum of all 'f' greater than threshold encountered
        min=temp;
    }
    return min;  //return the minimum 'f' encountered greater than threshold
}
function nextnodes(node)
{
        return list of all possible next nodes from node;
}
```

Fig 8: IDA Pseudocode [6]*

4. Heuristic Function

A heuristic function has 2 properties:

- Admissibility: Never overestimates the actual cost of reaching the goal [8]
- Consistency (monotonic): Satisfies the triangular inequality h(n) <= c(n, n') + h(n') [8]

We experimented with various heuristic functions to evaluate their effectiveness in guiding the search algorithms toward the goal state for the 8-puzzle problems. Not all the heuristic functions are consistent, which helps us understand that even if h(n) is inconsistent, the algorithm can still find the

solution, but the optimality is not guaranteed. For all the below heuristic the input state is a 2D array representing the puzzle.

1) Manhattan Distance: The Manhattan distance for a tile is the sum of the absolute differences between its current position and its target position in both the row and the column. The pseudocode for Manhattan distance is shown in figure 9.

```
1: function manhattan_distance(state, n):
2:     initialize total_distance = 0

3:     for each row index x from 0 to n-1:
4:         for each column index y from 0 to n-1:
5:             value = state[x][y]
6:             if value ≠ 0:   // Ignore the blank tile
7:                 target_x = (value - 1) // n   // Row index in goal state
8:                 target_y = (value - 1) % n   // Column index in goal state
9:                 distance_x = absolute(x - target_x)   // Row distance
10:                distance_y = absolute(y - target_y)   // Column distance
11:                total_distance += distance_x + distance_y

12:    return total_distance
```

Fig 9: Manhattan Distance Pseudocode

```
1: function misplaced_tiles(state, n):
2:     initialize count = 0
3:
4:     for each row index x from 0 to n-1:
5:         for each column index y from 0 to n-1:
6:             if state[x][y] is not equal to goal[x][y]:
7:                 // Increment count for misplaced tiles

8:     return count
```

Fig 10: Misplaced Tiles Pseudocode

2) Misplaced Tiles: The Misplaced Tiles heuristic counts how many tiles are in the wrong position, compared to the goal state. For each tile in the current state, we check if its position matches the goal configuration. If a tile is not in its target position, it is considered "misplaced." The pseudocode for Misplaced Tiles is shown in figure 10.

3) Linear Conflict: Linear Conflict builds on the Manhattan distance by adding a penalty for tiles that are in the correct row or column but are reversed with respect to their goal positions. This heuristic is designed to provide more precise estimates by accounting for additional constraints imposed by the puzzle's configuration. If a linear conflict is found, the heuristic adds a penalty of 2 for each conflict to the Manhattan distance. The pseudocode for Linear Conflict is shown in figure 11.

```
FUNCTION linear_conflict(state, n):
  INITIALIZE conflict_count = 0
  INITIALIZE manhattan = manhattan_distance(state, n)

  FOR each row FROM 0 TO n-1:
    FOR each column FROM 0 TO n-1:
      value = state[row][column]

      IF value == 0:
        CONTINUE

      target_row = (value - 1) // n
      target_col = (value - 1) % n

      IF target_row == row:
        FOR next_column FROM column+1 TO n-1:
          next_value = state[row][next_column]
          IF next_value != 0 AND (next_value - 1)
            IF next_value < value:
              INCREMENT conflict_count BY 2

      IF target_col == column:
        FOR next_row FROM row+1 TO n-1:
          next_value = state[next_row][column]
          IF next_value != 0 AND (next_value - 1) % n == column:
            IF next_value < value:
              INCREMENT conflict_count BY 2

  RETURN manhattan + conflict_count
```

Fig 11: Linear Conflict Pseudocode

```
FUNCTION inconsistent_heuristic(state, n):
  INITIALIZE distance = 0

  FOR each row index x FROM 0 TO n-1:
    FOR each column index y FROM 0 TO n-1:
      value = state[x][y]

      IF value != 0:
        target_x = (value - 1) // n
        target_y = (value - 1) % n

        IF value is even:
          penalty = 2
        ELSE:
          penalty = 1

        distance += penalty * (ABS(x - target_x) + ABS(y - target_y))

  RETURN distance
```

Fig 12: Inconsistent heuristic Pseudocode

4) Inconsistent Heuristic: The Inconsistent Heuristic modifies the Manhattan distance by doubling the penalty for tiles with even numbers, based on their movement distance. For each tile in the puzzle, the Manhattan distance to its target position is calculated as in the basic Manhattan distance. However, if a tile has an even number, its contribution to the total heuristic is weighted by a factor of 2, meaning its Manhattan distance is doubled. This heuristic is inconsistent as it doubles the penalty for the tiles with even number. This doubling causes the heuristic value to "jump" or "drop" unevenly when moving tiles, violating the consistency condition. The pseudocode for Inconsistent heuristic is shown in figure 12.

5) Pattern Database: The PDB is a precomputed database that stores the minimum number of moves required to solve specific tile patterns in a puzzle. We worked with the following variants of PDBs to optimize the A* algorithm.

Non-Additive Pattern Database

Pattern generation involves identifying subsets of misplaced tiles relative to the goal state. Using Breadth-First Search (BFS), the minimum number of moves required to solve each pattern independently is precomputed and stored in a Pattern Database (PDB). This precomputation allows for efficient heuristic evaluation during the search process. To estimate the heuristic value of a puzzle state, a non-additive approach is used, where the heuristic combines the precomputed distances from multiple PDBs by taking the maximum value among them. This method ensures an admissible heuristic, as it avoids overestimating the true cost while providing a more accurate and informed estimate compared to simpler heuristics.

```
FUNCTION generate_pdb(puzzle, n):

  goal_state = generate_goal_state(n)

  flat_puzzle = Flatten puzzle into a single list
  flat_goal = Flatten goal state into a single list

  misplaced_tiles_list = []
  FOR i FROM 0 TO n*n - 1:
    IF flat_puzzle[i] ≠ flat_goal[i] AND flat_puzzle[i] ≠ 0:
      APPEND flat_puzzle[i] TO misplaced_tiles_list

  FOR tile FROM 0 TO n*n - 1:
    IF tile NOT IN misplaced_tiles_list:
      APPEND tile TO misplaced_tiles_list

  patterns = [tuple(sorted(misplaced_tiles_list))]

  VALIDATE the puzzle input

  WITH multiprocessing pool:
    pdb_values = FOR EACH pattern CALL calculate_pdb_value(pattern, n)

  RETURN a dictionary {pattern: pdb_value}
```

```
FUNCTION calculate_pdb_value(pattern, n):

  INITIALIZE goal_state = List of integers from 1 to n*n-1 followed by 0
  start_state = Convert pattern to a flat list

  IF length of start_state ≠ n*n:
    RAISE ValueError

  Convert start_state to a 2D format
  INITIALIZE a priority queue (heap)
  INSERT (heuristic, 0, start_state) INTO queue

  INITIALIZE visited set
  ADD start_state TO visited

  DEFINE directions = [UP, DOWN, LEFT, RIGHT]

  WHILE queue is not empty:
    POP (heuristic, depth, state) FROM queue

    IF state == goal_state:
      RETURN depth

    FIND position of the blank tile (0)

    FOR each possible move (UP, DOWN, LEFT, RIGHT):
      IF move is valid:
        GENERATE new_state by swapping tiles
        IF new_state is not visited:
          ADD new_state TO visited
          COMPUTE heuristic for new_state using linear_conflict
          INSERT (depth + 1 + heuristic, depth + 1, new_state) INTO queue

  RETURN infinity
```

```
FUNCTION non_additive_pdb(state, n, pdb):
  INITIALIZE distance = 0

  FOR each tile in the state:
    IF tile ≠ 0:
      GENERATE pattern = Tuple of sorted non-zero tiles in the current state
      IF pattern exists in pdb:
        distance = MAX(distance, pdb[pattern])

  RETURN distance
```

Fig 13: Non-Additive Database Heuristic Pseudocode

Disjoint Pattern Databases:

The tiles are divided into disjoint (non-overlapping) groups, and a separate Pattern Database (PDB) is generated for each group. This grouping can be done in multiple ways; for example, in a 3x3 puzzle, we can have disjoint sets such as 3-3-2, 4-4, or 2-2-2-2. We aim to group tiles that are closer to each other in the goal state because these tiles tend to interact the most during the puzzle-solving process. By grouping interacting tiles, the heuristic estimates become more accurate and efficient, as it better reflects the positional constraints of these tiles.

While this heuristic remains admissible (it does not overestimate the true cost), it is not optimal because it ignores interactions between tiles in different groups. Although disjoint PDBs work well for smaller puzzles like the 8-puzzle, they become computationally expensive for larger puzzles, such as the 15-puzzle or beyond. The high number of permutations and cost calculations required to generate these databases demands substantial processing time and memory. On general consumer laptops, this process can be infeasible due to hardware limitations. The pseudocode for disjoint pattern database used by us is shown in figure 16.

```
FUNCTION disjoint_pdb(state, n):
        pdb <- Load precomputed ALL_PDB for a specific disjoint set configuration
        groups <- divide the problem in disjoint groups

        # Calculate heuristic value
        heuristic = 0
        For all group in groups:
                heuristic += lookup(pdb_dict, group_tuple, default=float('inf'))

  RETURN heuristic

FUNCTION create_disjoint_pdb(n):
    ALL_PDB = []  # To store all (state, cost) pairs across groups

    grouped_tiles <- Generate disjoint sets

    For all group in ALL_PDB:
        pdb = precompute_pdb(group, n)  # Reverse BFS to precompute distances
        all_group.extend(pdb)

    #Save the ALL_PDB to file

FUNCTION precompute_pdb(group, n):
    pdb = {}  # Dictionary to store (state, cost) pairs

    # Generate all unique permutations of the group
    all_states <- unique_permutations for group

    For all group in all_states:
        pdb[state] = reverse_bfs(state, n)  # Compute cost using Reverse BFS

    RETURN pdb
```

Fig 16: Disjoint Pattern Database Heuristic Pseudocode

Dual Lookup: The Dual Lookup Heuristic optimizes search performance by combining forward and backward pattern databases (PDBs). Instead of only considering the distance from the current state to the goal state, this heuristic also looks at the reverse distance (from the goal back to the start). This dual direction lookup helps improve accuracy and efficiency.

Forward PDB: Precomputed distances from the goal state to all possible states.

Backward PDB: Precomputed distances from the start state to all possible states.

At runtime, the heuristic combines the forward and backward PDB lookups to estimate the minimum cost.

Algorithm X The Dual Lookup Heuristic for State Space Search

```
function generate_pdb(reference_state, pattern)
    Input: reference_state: the goal or start state as a 2D array
           pattern: list of tiles included in the pattern
    Output: pdb: a pattern database mapping states to costs
    pdb ← ∅
    queue ← deque([(reference_state, 0)])
    reference_tuple ← tuple(map(tuple, reference_state))
    pdb[reference_tuple] ← 0
    while queue ≠ ∅ do
        current_state, cost ← queue.pop_left()
        for all neighbors ∈ get_neighbors(current_state) do
            neighbor_tuple ← tuple(map(tuple, neighbor))
            if neighbor_tuple ∉ pdb then
                pdb[neighbor_tuple] ← cost + 1
                queue.append((neighbor, cost + 1))
            end if
        end for
    end while
    return pdb
function dual_lookup_heuristic(current_state, forward_pdb, backward_pdb)
    Input: current_state: the current puzzle state
           forward_pdb: pattern database from goal to all states
           backward_pdb: pattern database from start to all states
    Output: h_dual: minimum cost estimate from both forward and backward lookups
    current_tuple ← tuple(map(tuple, current_state))
    forward_cost ← forward_pdb.get(current_tuple, ∞)
    backward_cost ← backward_pdb.get(current_tuple, ∞)
    h_dual ← min(forward_cost, backward_cost)
    return h_dual
```

Fig 14: Dual Lookup Heuristic Pseudocode

Algorithm X The Symmetry Heuristic for State Space Search

```
function generate_symmetric_states(state)
    Input: state: the current puzzle state as a 2D array
    Output: symmetric_states: a set of unique symmetric representations
    symmetric_states ← ∅
    state_tuple ← tuple(map(tuple, state)) // Convert the state into a hashable tuple
    Add state_tuple to symmetric_states
    for i = 1 to 4 do
        state ← rotate_90_clockwise(state) // Rotate the state 90° clockwise
        state_tuple ← tuple(map(tuple, state))
        Add state_tuple to symmetric_states
    end for
    horizontal_flip ← flip_horizontally(state)
    vertical_flip ← flip_vertically(state)
    Add tuple(map(tuple, horizontal_flip)) to symmetric_states
    Add tuple(map(tuple, vertical_flip)) to symmetric_states
    return symmetric_states
function symmetry_heuristic(current_state, goal_state)
    Input: current_state: the current puzzle state
           goal_state: the target goal state
    Output: h_symmetry: minimum heuristic cost across symmetric goal states
    symmetric_goals ← generate_symmetric_states(goal_state)
    min_cost ← ∞
    for all symmetric_goal ∈ symmetric_goals do
        cost ← manhattan_distance(current_state, symmetric_goal)
        min_cost ← min(min_cost, cost)
    end for
    return min_cost
```

Fig 15: Symmetric Heuristic Pseudocode

Symmetry: The Symmetry Heuristic is designed to reduce redundant searches by leveraging symmetric properties of the puzzle. Symmetry refers to transformations where certain states can be mirrored or rotated and remain functionally identical for the puzzle-solving process. By recognizing symmetric states as equivalent, this heuristic helps reduce the state space, improving search efficiency. The pseudocode for symmetry heuristic is shown in figure 15.

Methodology:

The following research questions will guide the investigation into the performance of the algorithms on the selected problem instances:

- Which algorithm is the most optimal for solving the 8-Puzzle?
    - This question aims to identify which algorithm consistently provides the best performance in terms of execution time, memory usage, and path length across all puzzle instances.
- *Which heuristic is the best for the A* and IDA** algorithm*?*
    - Since A* and IDA* can be enhanced with different heuristics, we aim to determine which heuristic (among several options such as Manhattan Distance, Misplaced Tiles, Linear Conflict, and others) leads to the best performance in solving the 8-Puzzle in terms of efficiency and optimality.

- *Does there exist a class of instances in which one algorithm always outperforms another algorithm? Is this true for all instances, or is the other algorithm still sometimes a better choice?*
    - Comparing algorithms within each category to identify whether certain algorithms and heuristic dominate in particular scenario types. This examination will also highlight cases where an algorithm's expected dominance breaks down, indicating that no single approach is uniformly superior.

Problem Instances

- The experiment uses a collection of 8-Puzzle instances. These puzzles are represented as 3x3 grids, where the goal is to rearrange the tiles to reach a goal state, typically ordered from 1 to 8 with the empty space (represented by 0) at the bottom-right corner.
- Puzzle Generation:
    - The puzzles can be read from text files or dynamically generated based on input parameters. For this experiment, a set of puzzles was tested to analyze the algorithm performance.

- Algorithms Tested

  The following algorithms were evaluated in this study:

  1. **A* Algorithm with different heuristics**:
     a. **Manhattan Distance**: The sum of the absolute differences between the current and goal tile positions.
     b. **Misplaced Tiles**: The number of tiles that are in the wrong position compared to the goal.
     c. **Linear Conflict**: A more informed heuristic that takes into account conflicts between tiles in the same row or column.
     d. **Inconsistent Heuristic**: A heuristic designed to simulate worst-case behavior.
     e. **Pattern Database Heuristic (PDB)**: Precomputed databases used to optimize the search process. We test non additive, disjoint, symmetry and dual lookup.
  2. **Greedy Algorithm**: This algorithm only optimizes based on the heuristic and does not consider the cost of the path to the current state.
  3. **Baseline Algorithms**: Other algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS) were also considered as baseline methods.

- The best algorithm will be determined based on the following metrics:
    - execution time (s),
    - success rate in solving puzzle (out of all puzzles given, how many of them the algorithm can produce a solution),
    - memory used (kb),
    - steps taken,
    - The optimality rate (for each puzzle, the algorithm is considered optimal if it has an equal to shorter run time than A* misplaced tiles. And after testing on all puzzles, we calculate the optimal rate to see out of n puzzle how many optimal solutions the algorithm can get)

- Since running an increment from 1 to 5 random puzzles takes a long time for testing and taking into consideration that some algorithms like DFS are well known for taking a long time compared to others, we have added a timeout for every algorithm. Each algorithm will have 2 minutes (scaled by 2^n according to puzzle size n) to solve, otherwise they will consider a failed solution.

Experimental Setup:

The experiment was run on Windows 11 using a 4 core i7-1165G7 running at 2.8 GHz, 12 GB RAM was available during runtime. The implementation of the algorithms and heuristics for solving the 8-Puzzle was done in **Python**. The specific version used for the experiments is **Python 3.9.7**. Python was chosen for its flexibility, rich libraries, and ease of implementing algorithms like A* and Greedy. The Python libraries that were utilized are Numpy, time, psutil, sys, os, matplotlib, itertools, heapq, collections, multiprocessing.

Experimental Results:

Below are the best search algorithms determined by testing from 5 randomly generated 3*3 puzzles that were solved by all the above algorithms. These instances were of varying complexity, some puzzles required more moves than others and the best search algorithms for each varied and not all of them obtained the best results with same algorithm and/or heuristics.

Puzzle #	Best Algorithm	Average Execution Time (s)	Average Memory Usage (KB)	Success Rate	Optimality Rate
1	greedy_dual_lookup	0.01925970000002053	11.4375	1.0	1.0
2	ida_star_manhattan	0.00025227049997056383	7.875	1.0	1.0
3	greedy_linear_conflict	0.04276409766650128	12.9453125	1.0	1.0
4	a_star_inconsistent:	0.0022076145000937686	17.61328125	1.0	1.0
5	ida_star_linear_conflict	0.03417550000031042	12.915625	1.0	1.0

We observed that the A* algorithm consistently outperformed IDA* in nearly all test cases. Below, we provide a comparison of the different PDB configurations we tested with the A* algorithm to evaluate their performance and effectiveness.

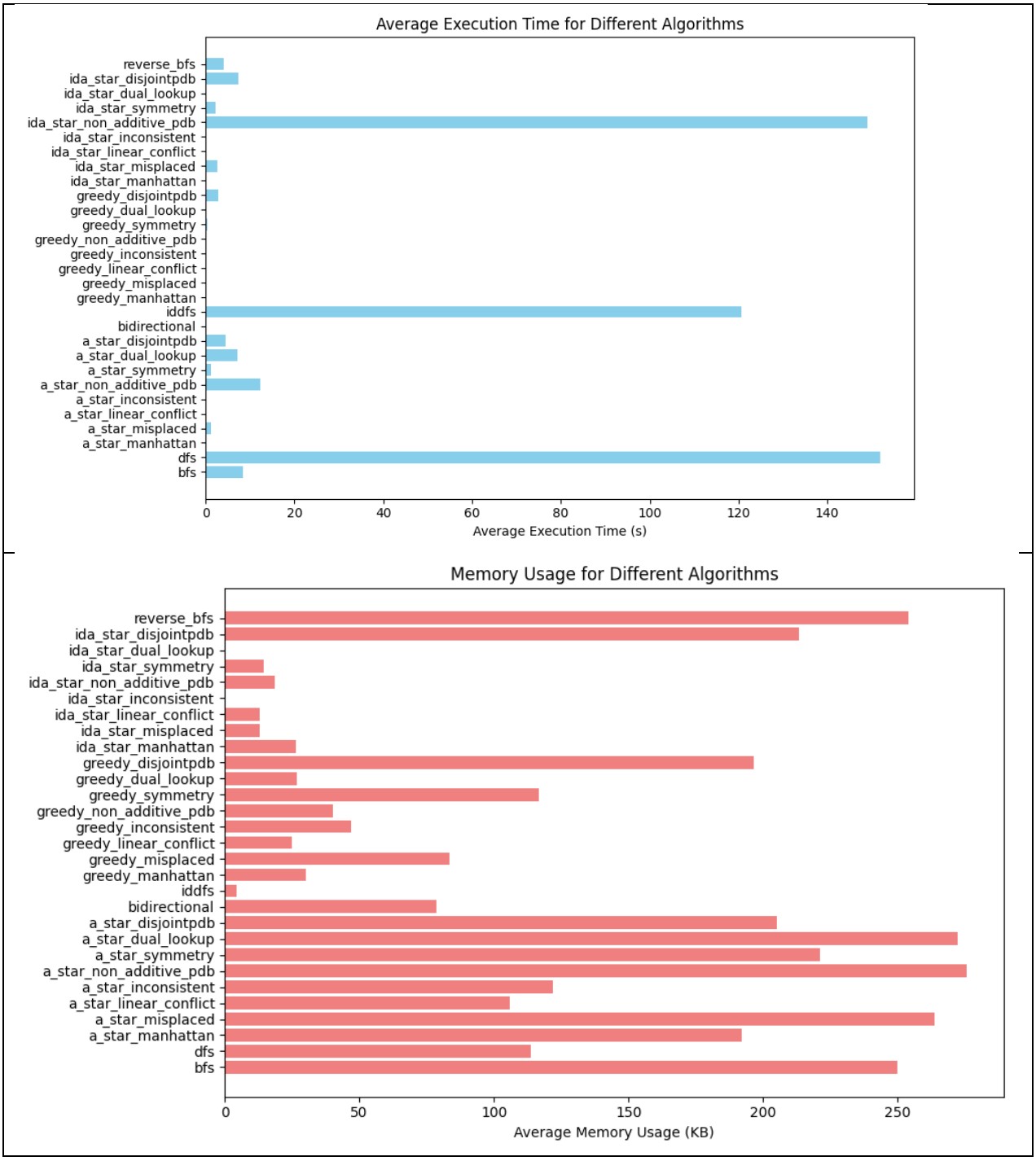
Algorithm	Average Execution Time (s)	Average Memory Usage (KB)	Success Rate	Optimality Rate
a_star_non_additive_pdb	4.545536808400004	258.1390625	1.0	1.0
a_star_dual_lookup	2.363241908400005	258.8640625	1.0	1.0
a_star_disjointpdb	5.972652675400013	206.9404296875	1.0	1.0

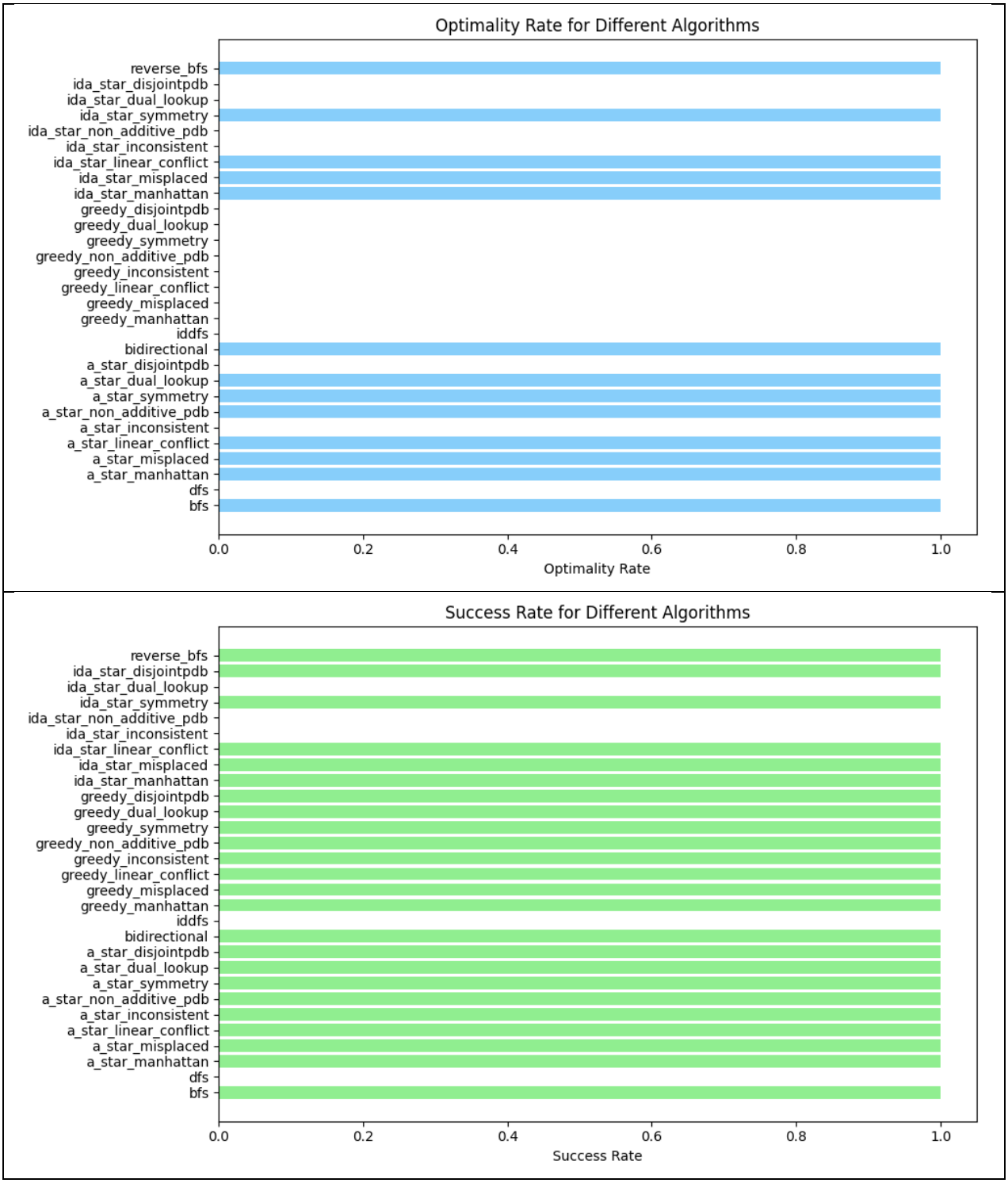
Below, we provide a comparison of the remaining different heuristics we tested with the A* algorithm to evaluate their performance and effectiveness.

Puzzle #	Best Algorithm	Average Execution Time (s)	Average Memory Usage (KB)	Success Rate	Optimality Rate
1	a_star_inconsistent	0.0740778750000004	121.548828125c	1.0	1.0
2	a_star_linear_conflict	0.0872847495000002	122.2265625	1.0	1.0
3	a_star_manhattan	0.027553513666665214	94.03125	1.0	1.0
4	a_star_linear_conflict	0.11151092725000922	138.55859375	1.0	1.0
5	a_star_inconsistent	0.0009281581999971422	7.9671875	1.0	1.0

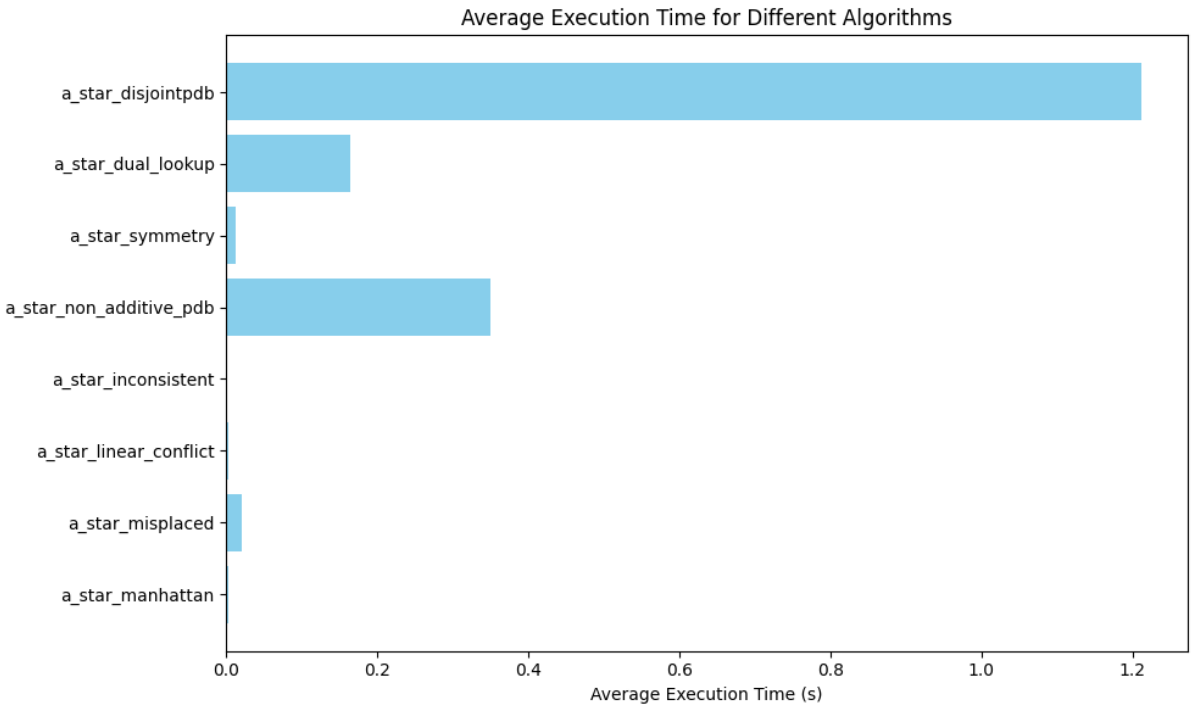
Conclusions:

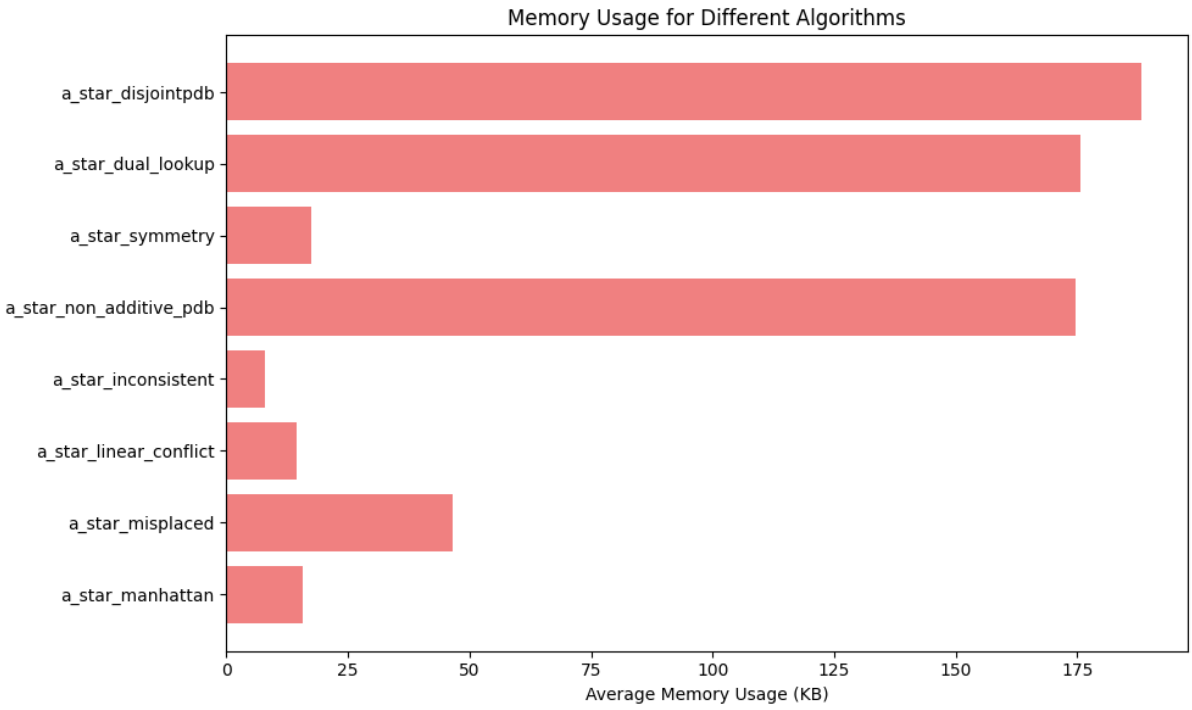
Comparison between all algorithms

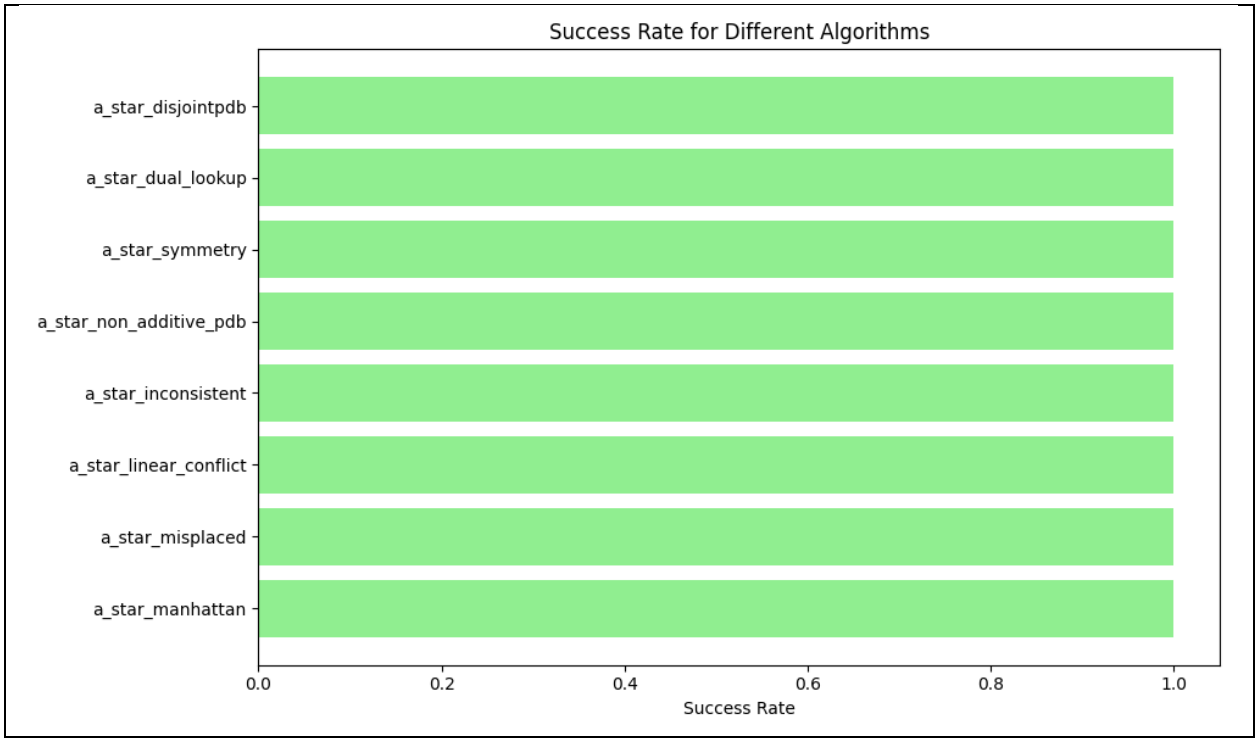

Average Execution Time for Different Algorithms
reverse_bfs
ida_star_disjointpdb
ida_star_dual_lookup
ida_star_symmetry
ida_star_non_additive_pdb
ida_star_inconsistent
ida_star_linear_conflict
ida_star_misplaced
ida_star_manhattan
greedy_disjointpdb
greedy_dual_lookup
greedy_symmetry
greedy_non_additive_pdb
greedy_inconsistent
greedy_linear_conflict
greedy_misplaced
greedy_manhattan
iddfs
bidirectional
a_star_disjointpdb
a_star_dual_lookup
a_star_symmetry
a_star_non_additive_pdb
a_star_inconsistent
a_star_linear_conflict
a_star_misplaced
a_star_manhattan
dfs
bfs
0
20
40
60
80
100
120
140
Average Execution Time (s)
Memory Usage for Different Algorithms
reverse_bfs
ida_star_disjointpdb
ida_star_dual_lookup
ida_star_symmetry
ida_star_non_additive_pdb
ida_star_inconsistent
ida_star_linear_conflict
ida_star_misplaced
ida_star_manhattan
greedy_disjointpdb
greedy_dual_lookup
greedy_symmetry
greedy_non_additive_pdb
greedy_inconsistent
greedy_linear_conflict
greedy_misplaced
greedy_manhattan
iddfs
bidirectional
a_star_disjointpdb
a_star_dual_lookup
a_star_symmetry
a_star_non_additive_pdb
a_star_inconsistent
a_star_linear_conflict
a_star_misplaced
a_star_manhattan
dfs
bfs
0
50
100
150
200
250
Average Memory Usage (KB)



Comparison between heuristics in A*


Average Execution Time for Different Algorithms
a_star_disjointpdb
a_star_dual_lookup
a_star_symmetry
a_star_non_additive_pdb
a_star_inconsistent
a_star_linear_conflict
a_star_misplaced
a_star_manhattan
0.0
0.2
0.4
0.6
0.8
1.0
1.2
Average Execution Time (s)


Memory Usage for Different Algorithms
a_star_disjointpdb
a_star_dual_lookup
a_star_symmetry
a_star_non_additive_pdb
a_star_inconsistent
a_star_linear_conflict
a_star_misplaced
a_star_manhattan
0
25
50
75
100
125
150
175
Average Memory Usage (KB)



According to the result, we can conclude that:

- The overall comparison highlights that *IDA algorithms** such as ida_star_linear_conflict and ida_star_manhattan excel in solving specific puzzles with remarkable execution times and minimal memory usage. These algorithms consistently deliver a success rate and optimality rate of 1.0, making them reliable for solving the 8-puzzle problem. Among these, ida_star_manhattan stands out for Puzzle 2 due to its extremely low execution time (0.00025s) and minimal memory footprint. However, A* algorithms, particularly a_star_inconsistent, also perform efficiently, especially in Puzzle 4, where it demonstrates a balance between execution time and memory usage. While overall trends favor both IDA* and A*, specific algorithms are puzzle-dependent, showcasing varying levels of efficiency.
- The heuristic comparison reveals that inconsistent and linear conflict heuristics are the most effective across different puzzles, often emerging as the best-performing heuristics. For instance, a_star_linear_conflict delivers strong results for Puzzles 2 and 4, while a_star_inconsistent shows outstanding performance for Puzzles 1 and 5, particularly with minimal memory usage (7.97 KB) in Puzzle 5. The Manhattan heuristic, although simpler, performs exceptionally well in Puzzle 3 with the lowest average execution time. Additionally, the comparison proves that the A* algorithm can still work optimally even when using an inconsistent heuristic, achieving both a success rate and optimality rate of 1.0. These findings indicate that while more complex heuristics like linear conflict and inconsistent yield robust and consistent results, simpler heuristics such as Manhattan can also provide significant efficiency under certain conditions.
- DFS and IDDFS always results in timeout even for a 3x3. Despite being known for guaranteeing optimal solution, their run time has put the algorithms at a high disadvantage when compared to others.

- The Second list are the results for the a* pdb algorithm, which has close to double run time and memory compared to other algorithms. That is not counting the time taken to compute the Pattern Database for the puzzle. This although guaranteeing an optimal solution, PDB are more suitable for focusing on one puzzle than solving multiple instances.

References:

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2010.

[2] R. E. Korf and A. Felner, "Disjoint pattern database heuristics," *Artificial Intelligence*, vol. 134, no. 1–2, pp. 9–22, Jan. 2002. [Online]. Available: https://doi.org/10.1016/S0004-3702(01)00092-3

[3] S. Choudhury, "Using uninformed & informed search algorithms to solve 8-puzzle / n-puzzle," *Sandipan's Web Log,* Mar. 16, 2017. [Online]. Available: https://sandipanweb.wordpress.com/2017/03/16/using-uninformed-informed-search-algorithms-to-solve-8-puzzle-n-puzzle/. [Accessed: Dec. 15, 2024].

[4] "Implement A* algorithm with C#: Understand pseudocode," *Stack Overflow*, Feb. 2017. [Online]. Available: https://stackoverflow.com/questions/42893205/implement-a-algorithm-with-c-sharp-understand-pseudoco. [Accessed: Dec. 16, 2024].

[5] A. Ansaf, "AI campus search agents: Informed search algorithms," Columbia University, 2017. [Online]. Available: https://www.cs.columbia.edu/~ansaf/courses/4701/AI_campus_search_agents_informed_new.pdf. [Accessed: Dec. 16, 2024].

[6] "IDA* Algorithm in General," *Algorithms Insight*, [Online]. Available: https://algorithmsinsight.wordpress.com/graph-theory-2/ida-star-algorithm-in-general/. [Accessed: 16-Jun-2024].

[7] "Bidirectional Search – Two End BFS," *Efficient Code Blog*, Dec. 13, 2017. [Online]. Available: https://efficientcodeblog.wordpress.com/2017/12/13/bidirectional-search-two-end-bfs/. [Accessed: 16-Jun-2024].

[8] "Admissible heuristic," Wikipedia, May 28, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Admissible_heuristic. [Accessed: Jun. 16, 2024].