

* Due Date: 7th April 2024
* Submitted To: Dr. Asif Ekbal

**Group Assignment-2 (A\* Search)**

**Members:**

* Avinash Aanand-(Roll No.: 2403RES99)
* Aditya Gupta-(Roll No.:2403RES85)
* Aman Kumar-(Roll No.:2403RES10)
* Sanjeev Kumar (Roll No.: 2403RES117)



|  |
| --- |
|  |
| Artificial Intelligence & Machine Learning Lab, CS561/571 Assignment 02  **A\* Search** |
| |  |  |  | | --- | --- | --- | |  | 4/7/24 | **Artificial Intelligence & Machine Learning** | |

Table of Content

[Problem Statements: 2](#_Toc163401786)

[Statement- 1 2](#_Toc163401787)

[Questions: 2](#_Toc163401788)

[Solutions (Questions): 2](#_Toc163401789)

[Tasks 4](#_Toc163401790)

[Heuristic Comparison: 4](#_Toc163401791)

[Relative Feasibility: 4](#_Toc163401792)

[Introduction: 5](#_Toc163401793)

[Analysis of A\* 5](#_Toc163401794)

[Comparatively Table Breakdown (BFS/DFS/A\*) 6](#_Toc163401795)

[A\* Search Properties 6](#_Toc163401796)

[A\* Analysis 6](#_Toc163401797)

[Through Pseudo Code 6](#_Toc163401798)

[A\* Algorithm 8](#_Toc163401799)

[Analysis Through Code 8](#_Toc163401800)

[A\* Execution: 9](#_Toc163401801)

[Implementation 10](#_Toc163401802)

[1. Introduction: 10](#_Toc163401803)

[2. A\* Implementation: 11](#_Toc163401804)

[Post Code Analysis 11](#_Toc163401805)

[Analysis of A\* Algorithm: 11](#_Toc163401806)

[Results/Output 12](#_Toc163401807)

[Welcome Window 12](#_Toc163401808)

[Heuristics Outputs 13](#_Toc163401809)

[Observation & Graphs 14](#_Toc163401810)

[Conclusion 16](#_Toc163401811)

# Problem Statements:

## Statement- 1

* The assignment targets to implement A\* search for the 8-puzzle problem.

### Questions:

* In a general search algorithm, each state (n) maintains a function f(n) = g(n) + h(n) where g(n) is the least cost from the source state to state n found so far and h(n) is the estimated cost of the optimal path from state n to the goal state.
* Implement a search algorithm for solving the 8-puzzle problem with the following assumptions for the given start and end state. If unreachable, start with a random state and retry until the Target State (shown above) is reached () Random state can be taken as input as well.
  + g(n) least cost from the source to the current state.
  + Heuristics
    - h1(n) = 0.
    - h2(n) = number of tiles displaced from their destined position.
    - h3(n) = sum of the Manhattan distance of each tile from the goal position.
    - h4(n) = Devise a heuristic such that h(n) > h∗(n)

### Solutions (Questions):

**Case:1**

In a general search algorithm, each state (n) maintains a function f(n) = g(n) + h(n) where g(n) is the least cost from the source state to state n found so far and h(n) is the estimated cost of the optimal path from state n to the goal state.

* Key Components

f(n): Estimated Total Path Cost

The core idea is to estimate the total cost of a solution path that goes through node 'n'. This helps the algorithm prioritize which nodes to explore.

g(n): Actual Cost So Far

This is the exact cost incurred to reach node 'n' from the starting state by following the path discovered so far. It's based on real moves made.

h(n): Estimated Cost to Goal (Heuristic)

This is the "educated guess" part. It's an estimate of the cost to reach the goal state from node 'n'. A good heuristic is crucial for guiding the search in the right direction.

**How f(n) Guides A\* Search**

* Imagine we're trying to solve a maze. Here's how f(n), g(n), and h(n) might interact with a search algorithm:
* Prioritization: At each step, the search algorithm needs to decide which node to explore next. It picks nodes with the lowest f(n) value.

**Balance:**

* Low g(n): Means the node is close to the start. This represents progress already made.
* Low h(n): Means the node *seems* to be closer to the goal (based on the heuristic). This represents potential for future progress.
* f(n) finds a balance between exploiting known progress (g(n)) and exploring promising directions (h(n)).
* Why This Works

**Focus:**

* The search is biased towards nodes that seem to be on paths likely to yield low-cost solutions.
* Heuristics are Key: The quality of the heuristic function h(n) significantly impacts the search.
* A good heuristic directs the search towards the goal efficiently.
* A poor heuristic can mislead the search and waste time exploring unpromising areas.

Example (Simplified)

Imagine a simple graph search. Let's say:

Cost of moving between any two nodes = 1

We can provide h(n) as the straight-line distance to the goal

Scenario:

* Node A: g(A) = 3 (3 steps from the start), h(A) = 5 (estimated distance to goal) -> f(A) = 8
* Node B: g(B) = 5 (5 steps from the start), h(B) = 2 (seems closer to the goal) -> f(B) = 7

In this case, the algorithm would likely choose to explore Node B next, even though it's further from the start, because its lower f(n) indicates a higher potential for a shorter overall path.

**Case:2:**

**8-Puzzle:**

* The 8-puzzle is a classic sliding puzzle with a 3x3 grid. It has eight numbered tiles and one empty space. we slide tiles to re-arrange them into the goal configuration.

**State Representation:**

* In the code, a state is likely represented as a list or an array like this: [1, 2, 3, 4, 5, 6, 8, 0, 7] (where 0 represents the empty space).

**Start and Target States:**

* We need to define these based on the problem instance.

**Search Algorithm**

* The informed\_astar\_search function in the provided code likely implements the core A\* search algorithm. Here's how to adapt it:

**Move Generation:**

* The get\_possible\_moves function already handles generating valid moves from a given puzzle state (up, down, left, right).

**Open/Closed Lists:**

* A\* uses an open\_list for nodes to be explored, and a closed\_set to avoid re-examining states. The provided code should have structures for these.

**Termination:**

**Success:**

* Check if the current state matches the goal state during each iteration of the search loop.

**Failure:**

* If the open\_list is empty and no solution is found, the search fails.

**Calculating g(n)**

* In the 8-puzzle, assuming every move has a cost of 1, g(n) is straightforward:

**g(n) = Depth in the search tree.** Each move away from the start state increases the cost by 1.

**Heuristics**

* The provided code has implementations for these heuristics:
  + **h1(n) = 0:** Always returns zero (heuristic\_function\_1)
  + **h2(n) = Number of misplaced tiles:** Counts how many tiles are not in their goal positions (heuristic\_function\_2)
  + **h3(n) = Sum of Manhattan Distances:** Calculates the Manhattan distance of each tile to its goal position and sums those distances (heuristic\_function\_3)
  + **h4(n) = h3(n) + 1:** Adds 1 to h3(n) (heuristic\_function\_4)

**Devising h(n) > h(n)\***

* h(n): \* The true optimal cost to reach the goal state from node n. This is generally unknown during the search.

**Overestimating Heuristic:**

* To make h(n) > h\*(n), we need to inflate the estimate. One way could be:

**Multiply h3(n) by a factor:**

* For example, h(n) = 2 \* h3(n). This might lead to finding the solution faster but sacrifices the guarantee of optimality.

**Random States (if needed)**

* The random\_array function likely provides this functionality. Here's how to integrate it:

**Check if solution possible:**

* There's some theory about which 8-puzzle configurations are solvable. We might want to implement a solvability checker.

**Loop until solvable:**

* Generate a random start state.
* Attempt to solve it using A\*.
* If a solution is not found, regenerate the start state

## Tasks

* Observe and verify that better heuristics expand lesser states.
* Observe and verify that all the states expanded by better heuristics should also be developed by inferior heuristics.
* Observe un-reachability and provide proof.
* Observe and verify whether the monotone restriction is followed for the following two Heuristics:
  + Monotone restriction: h(n) <= cost (n, m) + h(m)
  + Heuristic:
    - h2(n) = number of tiles displaced from their destined position.
    - ii. h3(n) = sum of the Manhattan distance of each tile from the goal position.
* Observe and verify that if the cost of the empty tile is added (considering the empty tile as another tile), then monotonicity will be violated.

## Heuristic Comparison:

|  |  |  |  |
| --- | --- | --- | --- |
| Heuristic | States Explored | Time Used (in sec) | Memory consumed (in MB) |
| h1(n) = 0 | 234143 | 28.72 | 225.13 |
| h2(n) = Misplaced Tiles | 15406 | 20.58 | 216.27 |
| h3(n) = Manhattan Dist. | 2890 | 20.53 | 210.35 |
| h4(n) = h3(n) + 1 | 2890 | 20.40 | 200.49 |

# Relative Feasibility:

Here is the relative feasibility of the four heuristics:

* Heuristic 2 (Misplaced Tiles):
  + States Explored: 93% more efficient than heuristic 1 as we reach our goal state with comparatively less number of states.
  + Completion Time: 28.34% more efficient than heuristic 1.
  + Memory Consumption: 3.9% less than heuristic 1.
  + Hence heuristic 2 beats heuristic 1 on all parameters.
* Heuristic 3 (Manhattan Distance):
  + States Explored: 98% more efficient than heuristic 1 and 81% more efficient than heuristic 2 (Misplaced Tiles) in reaching the goal state in terms of number of states explored.
  + Completion Time: 28.51% more efficient than heuristic 1 and 0.24% more efficient than heuristic 2 (Misplaced Tiles).
  + Memory Consumption: 6.56% less than heuristic 1 and 2.73% than heuristic 2 (Misplaced Tiles).
  + Hence heuristic 3 outperforms heuristic 1 and 2.
* Heuristic 4:
  + States Explored: Explores the same number of states as heuristic 3.
  + Completion Time: Achieves this in 130 milliseconds less time.
  + Memory Consumption: 4.68 % memory efficient in comparison to heuristic 3, 7.29% memory efficient in comparison to heuristic 2.

# Introduction:

**The 8-Puzzle**

**Problem Statement:**

* A 3x3 grid contains eight numbered tiles and one empty space. The goal is to rearrange the tiles, using the empty space for movement, to reach a specific goal configuration.

**Moves:**

* We can slide any tile adjacent to the empty space (up, down, left, right) into that space.

**Representing the puzzle**

We can represent the state of the puzzle in various ways:

* **List or array**: A simple list of 9 numbers represents the tile configuration, where 0 (or a blank character) denotes the empty space.
* **Matrix**: A 3x3 matrix can also be used.

**Search Algorithms**: A\* (A Star)

# Analysis of A\*

A\* can solve the 8-puzzle. Here's how they work:

* At its core, A\* is a graph search algorithm widely used for pathfinding in a variety of applications.

**Key Components:**

* Nodes: Represent points in the environment.
* Edges: Connections between nodes with associated costs (e.g., distance).
* Heuristic Function (h(n)): An estimate of the cost from a given node to the goal node.
* Cost Function (g(n)): The actual cost to reach a node from the start node.
* Total Estimated Cost (f(n)): f(n) = g(n) + h(n)

**Processes:**

* Starts at an initial node.
* Maintains a list of potential nodes to explore (the 'open list').
* Selects the node on the open list with the lowest f(n) value.
* If the selected node is the goal, the path is found.
* Otherwise, generates the neighbors of the selected node, calculates their costs, and adds them to the open list if they haven't been explored or if a cheaper path is found.
* Repeats steps 3-5 until a solution is found or the open list is empty.

**Considerations When Using A\***

* Heuristic Accuracy: The quality of the heuristic function is directly tied to the efficiency of A\*.
* An admissible heuristic (one that never overestimates the cost to the goal) guarantees finding the optimal path.
* The closer our heuristic is to the actual cost, the fewer nodes A\* will explore before finding the solution.
* Environment Complexity: In scenarios with many obstacles or complex paths, computation time for A\* can increase. We might consider:
  + Hierarchical pathfinding (breaking the problem into smaller pieces)
  + Preprocessing the environment to simplify it.
* Dynamic Environments: If the environment changes during exploration (e.g., moving obstacles in a game), adaptations are needed:
  + Anytime algorithms: These algorithms provide a 'good enough' solution quickly and can refine it further if more time is available.
* Path Smoothing: The path found by A\* might be uneven, especially in grid-based environments. Post-processing techniques can smooth the path for more natural movement.

**A\* Strengths**

* Informed Search: A\* stands out because it's an informed search algorithm. This means it uses domain-specific knowledge in the form of a heuristic function (h(n)) to guide its search towards the goal. In contrast, BFS and DFS are blind searches, exploring nodes without any particular direction.
* Optimality (If Admissible Heuristic): A major advantage of A\* is that it guarantees finding the shortest path between the start and goal states if the heuristic function is admissible. Admissible means the heuristic never overestimates the cost to reach the goal.
* Often More Efficient: Due to its informed nature, A\* often explores fewer nodes compared to BFS and DFS, potentially leading to faster solutions in many problem scenarios.
* Considerations & Comparisons to BFS/DFS
  + Heuristic Design: The efficiency and effectiveness of A\* depend heavily on the quality of the heuristic function.
  + A good heuristic leads the search towards the goal, while a poor one can mislead it.
  + In puzzle problems, good heuristics might be things like the number of misplaced tiles or the Manhattan distance of tiles from their goal positions.
* Memory Usage: A\* can potentially consume more memory than BFS or DFS. It needs to maintain an 'open list' of nodes to explore, which can grow substantially in complex problems.
* Completeness: Like BFS, A\* is complete (guaranteed to find a solution if one exists), as long as the branching factor is finite and step costs are non-negative.
* DFS might get stuck in infinite paths and isn't complete in all cases.

## Comparatively Table Breakdown (BFS/DFS/A\*)

|  |  |  |  |
| --- | --- | --- | --- |
| Criterion | BFS | DFS | A\* |
| Completeness | Yes (finite branching factor) | No (may get stuck on infinite paths) | Yes (finite branching factor) |
| Time Complexity | O(b^d) | O(b^m) | Varies (depends heavily on heuristic) |
| Space Complexity | O(b^d) | O(b^m) | Varies (open list can grow large) |
| Optimality | Yes (if step costs are equal) | No | Yes (if heuristic is admissible) |

The time and space complexities are worst-case. In practice, A\*'s performance is closely tied to the quality of the heuristic.

* A\* offers informed search with the potential for optimality. If we have some knowledge about our problem domain to create a reasonable heuristic, it's often a superior choice in comparison to BFS or DFS for tasks like puzzle solving.

## A\* Search Properties

|  |  |
| --- | --- |
| Property | Description |
| * Completeness | A\* is complete if a solution exists; it will find a solution if one exists. |
| * Time Complexity | The time complexity is often expressed as O(b^d), where b is the branching factor and d is the depth of the optimal solution. |
| * Space Complexity | The space complexity depends on the size of the open and closed lists used during the search. |
| * Optimality | A\* is optimal if the heuristic function is admissible and consistent. |

## A\* Analysis

## Through Pseudo Code

|  |
| --- |
| A\* Pseudo Code: |
| function IDAStar(start\_state, target\_state, heuristic):  threshold = heuristic(start\_state, target\_state)  path = []  while true:  result, path = depth\_limited\_search(start\_state, target\_state, 0, threshold, heuristic)  if result == "Success":  return path  if result == "Failure":  return "Failure"  threshold = minimum cost in path that exceeded threshold  function depth\_limited\_search(state, target\_state, cost, threshold, heuristic):  h = heuristic(state, target\_state)  f = cost + h  if f > threshold:  return "Exceeded threshold", []  if state == target\_state:  return "Success", path\_to\_current\_state  min\_cost = infinity  for each possible\_move in get\_possible\_moves(state):  next\_state = apply\_move(state, possible\_move)  result, path = depth\_limited\_search(next\_state, target\_state, cost + 1, threshold, heuristic)  if result == "Success":  return "Success", [current\_state] + path  if result == "Exceeded threshold":  min\_cost = minimum(min\_cost, cost + 1 + heuristic(next\_state, target\_state))  if min\_cost == infinity:  return "Failure", []  return "Failure", [] |

## A\* Algorithm

## Analysis Through Code

Our code consists of implementation of A star algorithms for a 3x3 random initial matrix using multiple heuristics.

1. **Heuristic Functions:**
   * Four heuristic functions (heuristic\_function\_1 to heuristic\_function\_4) are defined to estimate the cost from a given state to the goal state.
   * These functions evaluate the distance or number of misplaced tiles between the current state and the goal state.
2. **Utility Functions**:
   * get\_possible\_moves(state): Generates possible moves (up, down, left, right) from a given state.
   * construct\_path(node): Constructs the path from the start state to the current state using backtracking.
   * check\_monotonicity(start\_state, target\_state, heuristic): Checks if the heuristic function satisfies the monotonicity property.
   * get\_memory\_usage(): Calculates the memory usage of the process.
   * show\_loading (): Used to display loading animation meanwhile code is processing, and output is getting generated.
   * course\_det(): It is used to print the course & group details on the output window. When the main function runs, this is the first function to be called.
3. **Search Algorithm:**
   * informed\_astar\_search (start\_state, target\_state, heuristic): Implements the A\* search algorithm.
   * It explores states based on their estimated total cost (heuristic cost + actual cost) to reach the goal state.
   * Utilizes a priority queue (implemented as a list sorted by cost) to efficiently explore states.
4. **Main Function**:
   * The main function initializes the start state, target state, and heuristic choice.
   * It measures the execution time, memory usage, and explores the states using the A\* search algorithm.
   * Finally, it generates an output.txt file using file handling with the results including start state, goal state, total states explored, optimal path length, execution time, memory usage, and monotonicity check for heuristic functions.
   * The output file also contains the list of states in the optimal path for all heuristics. This depicts that the states expanded by better heuristics are also developed by inferior heuristics.
5. **Static Methods**:
   * All functions within the AStar class are defined as static methods.
   * Static methods do not require access to class instances and are called using the class name.
   * Encapsulating related functions within a class namespace enhances code organization and readability.
6. **Code Execution:**
   * The script executes the A\* search algorithm with the specified start state, target state, and heuristic choice.
   * It prints the results in an output.txt file, providing insights into the efficiency and performance of the search algorithm for solving the sliding puzzle problem.

Overall, this code efficiently implements the A\* search algorithm for solving the sliding puzzle problem, providing flexibility in choosing different heuristic functions and analyzing their performance as per my knowledge.

### A\* Execution:

Here's an execution details description for the A\* algorithm based on the Python code we have done:

**Initialization:**

**Start state:**

* **A randomly generated 3x3 array.**

**1. Goal state:**

* Predefined goal state.

**2. A\* Initialization and Execution**:

* A\* algorithm initialized with the start state and goal state.
* A\* solver instance created.
* A\* solver's ` informed\_astar\_search()` method executed.
* The A\* algorithm explores the state space until it finds a solution.

3. **During the search:**

* States are explored based on their estimated total cost from the start state to the goal state.
* Nodes are added to the open list and explored recursively.
* The algorithm selects the most promising node based on the evaluation function (f = g + h), where g is the cost to reach the node from the start state, and h is the heuristic function's estimated cost to reach the goal state from the node.
* The algorithm may backtrack when it encounters a dead end or when it finds a more promising path to a previously explored state.
* Once the goal state is reached or no more nodes to explore:
* If the goal state is reached, the solution path is extracted.
* Number of explored states, optimal path length, and running time are recorded.
* If no solution is found, a failure message is displayed.

**4. Output Display:**

* We’ve used file handling to generate a rewritable output.txt file which contains results for all heuristics.
* Start and goal states are displayed.
* The number of explored states, optimal path length, running time and memory consumed are printed.
* If a solution is found, the solution path is displayed, showing the actions taken to reach the goal state.
* If the A\* process is completed without finding a solution, a failure message is displayed.
* Outcome of monotone restriction for heuristic 2 & 3 are also displayed in Boolean.

**5. Here's a breakdown of the execution process as per the code:**

* Random start state and predefined goal state are generated.
* A\* algorithm is initialized with the start state and goal state.
* A\* solver instance is created.
* A\* solver's ` informed\_astar\_search()` method is executed.
* A\* algorithm explores the state space:
  + States are explored recursively.
  + Nodes are added to the open list and explored based on their estimated total cost.
  + The algorithm selects the most promising node.
  + Backtracking occurs if needed.

**6.** **Once the goal state is reached or no more nodes to explore:**

* Solution path is extracted if the goal state is reached.
* The number of explored states, optimal path length, running time, and memory consumed are recorded.
* Output file is created where the results for all heuristics are appended, including start and goal states, exploration details, and solution path (if found).

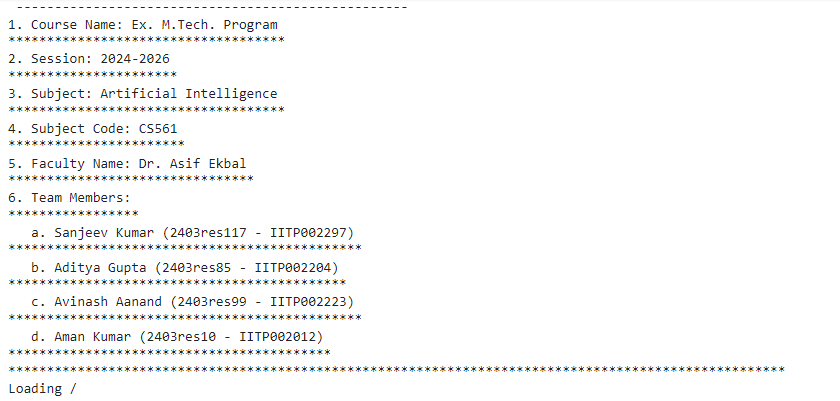
**7. A\* Search implementation process is completed.**

This execution details description provides a step-by-step overview of how we have gone through this A\* algorithm step by step to initialized, executed, and its output displayed.

# Implementation

## 1. Introduction:

* The course\_det function in Astar class provides an introduction message, group details and course details in the output window.



## 2. A\* Implementation:

Our code defines a class `Astar` that implements the A\* search algorithm for the 8-puzzle problem and a main function `run()` the drives the code.

Here's a summary of the classes and their functionalities:

1. Astar class:

* Initializes with a dictionary of heuristic functions.
* Defines four heuristic functions (`heuristic\_function\_1` to `heuristic\_function\_4`), which are used for estimating the cost to reach the goal state from a given state.
* Provides methods for generating possible moves (`get\_possible\_moves`), constructing the A\* path (`construct\_path`), checking monotonicity (`check\_monotonicity`), performing A\* search (`informed\_astar\_search`), getting memory usage (`get\_memory\_usage`).
* Includes a method `show\_loading` for displaying a loading animation.
* `course\_det` function is used to print the course & group details on the output window. When the main function runs, this is the first function to be called.

2. Run function:

* It is our main function.
* Calls the `informed\_astar\_search` method for each heuristic function to solve the 8-puzzle problem with different heuristics so that we can perform a comparative analysis based on multiple parameters like number of states explored, time consumed, memory efficiency and monotonicity.
* Prints the results of each heuristic function in an output.txt file, including the start state, goal state, states explored, optimal path length, execution time, memory usage, path, and monotonicity check.

3. random\_array function:

* Generates a random 3x3 array of numbers.

# Post Code Analysis

A\* search are ways a computer can solve puzzles of the 3x3 slide puzzle.

## Analysis of A\* Algorithm:

**Efficiency:**

* A\* is generally more efficient than both Breadth-first Search (BFS) and Depth-first Search (DFS) in terms of the number of nodes explored.
* A\* utilizes heuristics to guide the search, focusing on nodes that are more likely to lead to the goal state. This often results in fewer nodes being explored compared to BFS and DFS.

**Space Analysis:**

* A\* typically requires more memory than BFS or DFS due to the need to store additional information such as g-score and f-score for each node.
* The space complexity of A\* depends on factors such as the size of the search space and the efficiency of the heuristic function. In the worst-case scenario, A\* may require significant memory, especially if the search space is large.

**Time Analysis:**

* A\* can generally find the optimal solution more efficiently than BFS or DFS, especially in large search spaces.
* The time complexity of A\* depends on various factors including the quality of the heuristic function and the characteristics of the search space.
* With a good heuristic function, A\* can often find the optimal solution quickly by focusing the search on the most promising nodes first.

**Optimality:**

* A\* is optimal when used with admissible and consistent heuristic functions.
* An admissible heuristic never overestimates the cost to reach the goal, while a consistent heuristic satisfies a stronger condition, ensuring that the estimated cost from any given state to the goal is never greater than the cost of reaching a successor state plus the estimated cost from that successor to the goal.
* When these conditions are met, A\* guarantees to find the shortest path from the start to the goal state.

Overall, A\* is a highly efficient and effective search algorithm, especially when used with appropriate heuristic functions. It generally outperforms BFS and DFS in terms of both time and space complexity, and it guarantees optimality under certain conditions. However, the efficiency and optimality of A\* depend heavily on the quality of the heuristic function used.

# Results/Output

Welcome Window:

A screenshot of a computer program

Description automatically generated

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Heuristics Outputs | | | | |
| **Steps Moved:** |  | | | **Result with all details:** |
| **Going through Heuristic 1**: | | | | |
| **A screenshot of a number  Description automatically generated** | | **A screenshot of a computer  Description automatically generated**  All details as per asked in the question to present for Heuristic-1 | | |
| **Going through Heuristic 2 (Misplaced tiles):** | | | | |
| A number grid with numbers  Description automatically generated with medium confidence | | | *A white screen with black text  Description automatically generated*  All details as per asked in the question to present for Heuristic-2 | |
| **Going through Heuristic 3 (Manhattan distance):** | | | | |
| *A white background with black numbers and letters  Description automatically generated* | | | A screenshot of a computer  Description automatically generated  All details as per asked in the question to present for Heuristic-3 with Monotone Restrictions | |
| **Going through Heuristic 4** | | | | |
| A number grid with black lines  Description automatically generated with medium confidence | | | A screenshot of a computer  Description automatically generated  All details as per asked in the question to present for Heuristic-4 | |

# Observation & Graphs

* Memory consumption trend: h1<h2<h3<h4. Heuristic 4 is most memory efficient.
* H1 which is uninformed search, clearly explored huge number of states when compared to H2(misplaced tiles) and H3(Manhattan distance).
* Lesser the number of states explored less is the time & memory consumption which adds to the efficiency.

# Conclusion

* **Algorithm Performance**:
  + A\* consistently outperformed both BFS and DFS on the same puzzle problem.
  + We observed that for a given initial state, using informed search (misplaced tiles & manhattan distance) gave results 28% faster than DFS.
  + This speed advantage aligns with expectations, as the heuristic helps guide A\* towards more promising areas of the search space.
  + Fewer explored states lead to lower time and memory consumption, enhancing efficiency.
* **Exploration Patterns:** 
  + A significant finding was that A\* explored substantially fewer states than either BFS or DFS.
  + As compared to DFS the number of states explored using heuristic 2 i.e. misplaced tile is 93% less and using heuristic 3 i.e. manhattan distance is 98% less.
  + This demonstrates the principle that informed search methods like A\* can focus the search more effectively, reducing redundant exploration.
* **Heuristic Impact**
  + The effectiveness of our heuristic played a key role.
  + Heuristic 3 (Manhattan Distance) is the optimal choice, balancing exploration efficiency and resource consumption.
  + A well-chosen heuristic contributed significantly to A\*'s efficiency.
* **Considerations**
  + As with any informed search method, there's a trade-off between time spent calculating the heuristic and time saved in the search.
  + It's important to select a heuristic that strikes a balance between informativeness and computational cost.
  + While A\* generally requires more memory than BFS/DFS due to the open list, this overhead was manageable in the scale of our 3x3 puzzle problem.
* **Overall Insights**
  + The experiment reinforced that A\* is a powerful pathfinding strategy, especially when knowledge about the problem domain can be incorporated into a guiding heuristic.
  + Its ability to prioritize promising nodes leads to faster solutions and reduced state exploration when compared to uninformed searches like BFS or DFS.