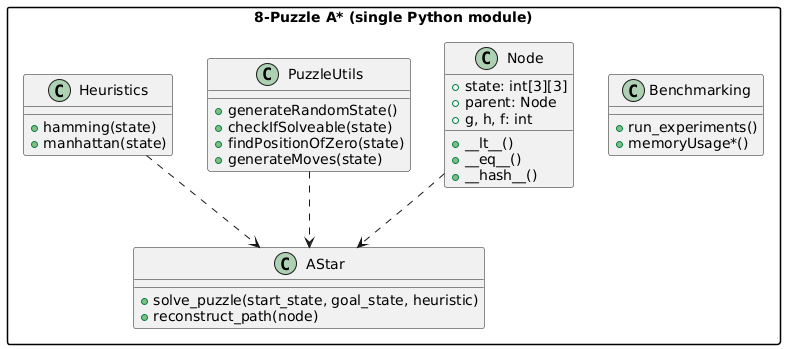
# **8-Puzzle Task Documentation**

**Course:** Introduction to AI and Data Science ILV  
**Group Members:** Matthias Bichler, Roman Chowdhury,   
 Svetlana Mishina, Katja Vymyslicky

**1. Short task description**

The goal of this lab was to solve the 8-Puzzle problem using the A\* search algorithm with two different heuristic functions: Hamming distance and Manhattan distance.  
 The program generates random start states, checks if they are solvable, and then finds the optimal path to the goal state.  
 We measure for each heuristic the run time and the number of expanded nodes for 100 random solvable puzzles.  
 Finally, we compare both heuristics and calculate mean and standard deviation of the results.  
  
**2. Software architecture diagram**

Note: The Diagram was made by using PlantUML. PlantUML is a tool for creating UML diagrams from plain text.

  
  
**3. Short descriptions of modules and interfaces**

Puzzle / Board utilities

start\_state, goal\_state — predefined 3×3 lists.

checkIfSolveable(state) — returns True if the inversion count is even.

findPositionOfZero(state) — finds coordinates of the empty tile (0).

generateMoves(state) — returns all valid next board states by swapping 0 up/down/left/right.

generateRandomState() — creates random 3×3 state with values 0–8.

Heuristics

hamming(state) — counts misplaced tiles (except 0). Complexity O(N).

manhattan(state) — sums |Δrow| + |Δcol| for tiles 1–8. Complexity O(N). Both are admissible heuristics.

A\* Solver

Class Node - stores state, parent, g, h, and f=g+h; implements \_\_lt\_\_, \_\_eq\_\_, and \_\_hash\_\_ for use in heapq and sets.

reconstruct\_path(node) - traces path from goal to start.

solve\_puzzle(start\_state, goal\_state, heuristic) - full A\* implementation using heapq as priority queue and a set for the closed list; returns (path, nodes\_expanded, elapsed\_seconds).

Benchmarking & Memory

run\_experiments() - generates 100 solvable states, runs A\* with both heuristics, records time and expanded nodes, computes mean and stdev.

memoryUsage\*() - simple functions using tracemalloc to print current/peak memory usage for heuristics and move generation.

**4. Explain design decisions**  
4.1 State Representation

State stored as nested list [ [..], [..], [..] ]; converted to tuple‑of‑tuples for hashing. Simpler for indexing and printing.

4.2 A\* Implementation

Priority queue via Python heapq; Node.\_\_lt\_\_ compares f. Updating a node in open list done by linear search + heapify() (sufficient for small puzzles).

4.3 Cost Calculation

Total cost f = g + h. Step cost = 1 per move.

4.4 Heuristics

hamming and manhattan implemented exactly as theoretical definitions. Manhattan expected to expand fewer nodes because it is more informative while remaining admissible.

4.5 Random State Generation

generateRandomState() followed by solvability filtering until 100 solvable boards are collected. No fixed seed → results differ per run.

4.6 Benchmarking

run\_experiments() prints average and standard deviation of runtime and expanded nodes for both heuristics. Solution depth is not recorded (can be added easily using len(path)).

4.7 Memory Measurement

tracemalloc used for approximate measurement of current and peak memory usage during heuristic computations.

**5. Discussion and conclusions**

**5.1 Describe your experience**

Working on Task 1 with our own implementation of the 8-Puzzle A\* algorithmprovided hands-on experience with:

designing and manipulating 2-D list data structures for representing puzzle states;

implementing search nodes (Node class) with cost values g, h, f = g + h;

learning how the A\* algorithm expands states based on the sum of actual and estimated costs;

coding two admissible heuristics (hamming() and manhattan()) and verifying their effect on search efficiency;

using Python modules such as heapq for the priority queue, copy for deep copying states, and tracemalloc for measuring memory usage;

performing automated experiments (run\_experiments()) to gather statistics over 100 random solvable puzzles.

This task helped us clearly understand how admissible heuristics guide A\*, and how data-structure choices (for example, using tuples for hashing and sets for the closed list) influence performance.  
 Compared with the reference version written in separate notebooks, our implementation uses one unified Python file. It required careful function design and testing to keep the code modular and readable.  
 Challenges included verifying solvability using inversion counts, handling deep copies of nested lists, and debugging the open-list update logic in the heap.Through these, we gained deeper insight into how search algorithms work in practice.

5.2 Table with Complexity Comparison of Heuristics

Both heuristics operate in linear time O(N), where N = 9 tiles in the 8-Puzzle. Each function examines every tile once to compute the heuristic cost.

|  |  |  |
| --- | --- | --- |
| **Heuristic** | **Time Complexity** | **Description** |
| Hamming (hamming) | O(N) | Counts how many tiles are not in their goal positions (ignoring 0). |
| Manhattan (manhattan) | O(N) | Sums the Manhattan distances ((|Δrow| + |Δcol|) of each tile from its goal position. |

After running run\_experiments(), insert the measured values in the table below.

**Interpretation:**  
 In theory and as seen in the reference experiment, both heuristics yield solutions of equal depth because A\* with an admissible heuristic always finds the optimal path.  
 However, Manhattan is expected to be significantly faster and to expand far fewer nodes.  
 This happens because Manhattan provides a more accurate distance estimate - it measures *how far* each tile is from its goal rather than only *whether* it is misplaced - which leads the algorithm to explore fewer unnecessary states.

5.3 Possible improvements in future

Linear Conflict Heuristic  
 An enhancement of Manhattan: besides distances, it also detects pairs of tiles in the same row or column but in reversed order. Adding the cost of these conflicts produces a stronger yet still admissible heuristic, further reducing the number of expanded nodes.

Pattern Databases (PDBs)  
 A pattern database stores precomputed shortest distances for subsets of tiles.  
 During search, the heuristic simply looks up these distances, giving very accurate estimates at the cost of additional memory.  
 Such PDB heuristics are common in high-performance 8-Puzzle solvers.

Bidirectional A\*  
 Two A\* searches — one from the start and one from the goal — run simultaneously and meet in the middle.  
 This often cuts the search space dramatically and can find solutions faster, although the implementation is more complex.