AI Homework

Antonino Prinzivalli
2058753
prinzivalli.2058753@studenti.uniroma1.it

January 12, 2026

1 Introduction

In this report, I present the implementation and analysis of an optimal solver for the generalized Sliding Tile Puzzle (e.g., 15-puzzle and 24-puzzle). The core of the solution is based on the A* search algorithm, designed to strictly satisfy the "duplicate elimination" and "no reopening" constraints required by the assignment.
To address the problem effectively, I modeled the puzzle as a state-space search where the goal is to reach a canonical ordered configuration. A critical component of the implementation is the instance generation strategy: I developed a robust mechanism to generate guaranteed solvable instances via random walks (scrambling) starting from the goal state. This approach ensures mathematical solvability by construction, avoiding a generate-and-test approach based on sampling random permutations and filtering them via parity checks.
The search algorithm utilizes a priority queue-based frontier and a closed set to manage state exploration efficiently. I implemented and compared two admissible heuristics to guide the search:

- **Misplaced Tiles:** A coarse heuristic that counts the number of incorrect tiles.
- **Manhattan Distance:** A more informed heuristic that sums the grid distances of tiles from their target positions.

The experimental evaluation focuses on two main aspects: the scalability of the algorithm across different board sizes (4×4 and 5×5) and a direct efficiency comparison between the two heuristics.
The results demonstrate that while both heuristics guarantee optimality, the Manhattan Distance heuristic provides a dramatic performance improvement. Specifically, on 5×5 instances, Manhattan reduces the search effort by approximately $52\times$ in terms of node expansions and $39\times$ in runtime compared to the Misplaced Tiles heuristic, confirming the critical importance of heuristic quality in combinatorial search.
In this report, I also implement a second optimal solving approach for the generalized Sliding Tile Puzzle by reducing it to a *Constraint Satisfaction Problem (CSP)* and solving it with the Z3 SMT solver. Instead of exploring states incrementally as in A*, I encode the entire evolution of the board over a finite time horizon T and ask Z3 whether there exists a sequence of legal blank moves that transforms the initial configuration into the goal.
To obtain optimal solutions under unit step cost, I use *iterative deepening on the horizon*: I solve a sequence of CSP instances for $T = 0, 1, 2, \ldots,$ `max_steps` and stop at the first satisfiable horizon. This guarantees that the returned plan has minimum length, while keeping the solver integration fully automatic: the code generates constraints, calls Z3, parses the model, reconstructs the action sequence (`U/D/L/R`), and validates it by simulation.
The experimental evaluation reports CSP-specific metrics that reflect the cost of repeated satisfiability checks (e.g., `solver_calls` and `horizons_tried`) together with runtime and solution cost. On 4×4 instances generated with 30 scramble moves, CSP/Z3 finds an optimal solution of cost 26 (matching A* with Manhattan), but with much higher runtime (minutes) due to solving multiple unsatisfiable horizons before reaching the first satisfiable one.
This highlights the expected trade-off: CSP/Z3 provides a general, declarative, and optimal planning-style solver, while A* with a strong heuristic is typically far more efficient on sliding puzzles.

2 Implementation of A*

I implemented the A* algorithm within the function `astar_duplicate_elimination_no_reopening`, designing the data structures to strictly adhere to the "duplicate elimination" and "no reopening" constraints.
To encapsulate the search state, I defined a `Node` class. Each `Node` instance stores the current configuration (state), a reference to the parent node, and the action applied to reach it, which allows for the reconstruction of the solution path. Additionally, it explicitly stores the path cost $g(n)$ (the cost to reach node n from the start) and the heuristic value $h(n)$ (the estimated cost from n to the goal). The priority for the frontier is determined by the total estimated cost $f(n) = g(n) + h(n)$.
The algorithm initializes by creating the start node with $g(0) = 0$ and $h(0)$ computed via the provided heuristic. To manage the open list, I utilized a priority queue (`frontier_heap`) implemented as a min-heap. This heap stores tuples in the format $(f, \texttt{tie\_counter}, \texttt{state})$, ordered by ascending $f(n)$. Crucially, the heap stores only the lightweight state representation, not the full `Node` object; the corresponding `Node` instance is stored in and retrieved from a parallel dictionary, `frontier_nodes` (mapping `state` $\rightarrow$ `Node`), which always references the best-known version of a node currently in the frontier.
At each iteration, the algorithm extracts the element with the lowest $f(n)$ from the priority queue. To handle the lack of a native "decrease-key" operation in Python's `heapq`, I implemented a "lazy deletion" strategy: I repeatedly pop from the heap until I find a state that is still present in the `frontier_nodes` dictionary. If a popped state is not in the dictionary, it is considered "stale" (invalidated by a previous update) and is ignored.
Once a valid node is extracted, it is removed from the frontier dictionary and checked against the goal test. If it is not the goal, the state is added to the explored set (closed list). This is where the *no reopening* constraint is enforced: during the expansion phase, any successor state (`child_state`) that is already present in `explored` is immediately discarded. This check occurs before checking for duplicates in the frontier, ensuring that closed states are never re-evaluated.
For node expansion, I iterate through all applicable actions, computing the new path cost ($\texttt{child\_g} = \texttt{node}.g + \texttt{step\_cost}$). I then implement duplicate elimination in the frontier:

- If the child state is not in `frontier_nodes`, it is added to both the dictionary and the heap.
- If the child state is already in the frontier, I compare the new $g(n)$ with the existing one. I replace the node only if the new path is strictly better ($\texttt{child}.g < \texttt{existing}.g$). This involves updating the entry in `frontier_nodes` and pushing the new tuple to the heap (leaving the old entry to be treated as stale).

It is worth noting that a strict "no reopening" policy can compromise optimality when using inconsistent heuristics; however, in this domain optimality is preserved because the heuristics used (Manhattan distance and Misplaced Tiles) are consistent (monotone). This property ensures that the first time a node is selected for expansion, the optimal path to that node has been found. Finally, the implementation tracks performance metrics, including expanded/generated nodes and max frontier size, and reconstructs the solution sequence by backtracking through the parent pointers.

2.1 Heuristics Definition

In the A* algorithm, the heuristic function $h(s)$ estimates the minimum remaining cost to reach the goal configuration from a given state s. I implemented and compared two distinct heuristics:

Misplaced Tiles Heuristic ($h_{\text{misplaced}}$): This heuristic calculates the number of numbered tiles (excluding the blank) that are currently not in their goal position:

$$h_{\text{misplaced}}(s) = |\{i \mid s[i] \neq 0 \wedge s[i] \neq \text{goal}[i]\}|$$

Intuitively, this represents a coarse lower bound, based on the logic that any misplaced tile requires at least one move to be corrected. While it is more informative than the zero heuristic, it treats all positional errors equally: a tile offset by one cell contributes the same value as a tile on the opposite side of the grid. Consequently, it is weaker guidance than distance-based metrics.

Manhattan Distance Heuristic ($h_{\text{manhattan}}$): This heuristic computes the sum of the vertical and horizontal distances each tile must travel to reach its target coordinates. For every tile $t \neq 0$, if

(r_t, c_t) is its current position and (r_t^*, c_t^*) is its goal position, the value is:

$$h_{\text{manhattan}}(s) = \sum_{t \neq 0} (|r_t - r_t^*| + |c_t - c_t^*|)$$

This metric provides a much more granular estimation than misplaced, as it quantifies how far each tile is from its destination rather than just if it is wrong. By penalizing states where tiles are geometrically distant from the goal, it directs the search more efficiently. It is an admissible and highly effective heuristic, generally expanding significantly fewer nodes than the other method on complex instances.

3 Implementation of CSP (Z3)

In addition to A*, I implemented an optimal solver by encoding the Sliding Tile Puzzle as a *Constraint Satisfaction Problem (CSP)* and solving it with the Z3 SMT solver. The key idea is to represent the full trajectory of the board from time $t = 0$ to time $t = T$ and constrain each transition to correspond to a legal blank move.

Time-indexed variables. For a board of size $N \times N$ (with $n = N^2$ cells), the encoding introduces:

- Integer variables $B_{t,i}$ for each time $t \in \{0, \ldots, T\}$ and cell index $i \in \{0, \ldots, n-1\}$, where $B_{t,i} \in \{0, \ldots, n-1\}$ denotes which tile occupies cell i at time t.
- An action variable A_t for each step $t \in \{0, \ldots, T-1\}$, with domain $\{U, D, L, R\}$ (encoded as integers $0 \ldots 3$ in the implementation), representing the blank move executed between t and $t+1$.

Core constraints. The CSP enforces the following conditions:

- **Permutation constraints:** for each time t, the board is a permutation of all tiles, implemented by combining domain bounds with `Distinct` over $\{B_{t,0}, \ldots, B_{t,n-1}\}$.
- **Initial and goal constraints:** the time-0 board equals the given initial configuration, and the time-T board equals the goal configuration.
- **Legal moves:** action preconditions ensure that a move does not send the blank out of bounds (e.g., action U is forbidden when the blank is already in the top row).
- **Transition (swap):** between t and $t+1$, the blank swaps with the adjacent tile selected by A_t, while all other cells keep the same value. This is encoded with conditional (`If`) constraints to correctly handle symbolic indexing in Z3.

Optimality via iterative deepening. To guarantee minimum-length solutions under unit costs, I solve a sequence of CSP instances for horizons $T = 0, 1, 2, \ldots,$ `max_steps` and stop at the first satisfiable horizon. The first SAT horizon corresponds to the optimal solution cost.

CSP-specific metrics. Besides total runtime (`elapsed_sec`) and solution cost, I record: (i) `solver_calls`, the number of Z3 checks performed, and (ii) `horizons_tried`, the number of horizons tested (SAT + UNSAT) up to the solution.

4 Problem

For this assignment, I chose to solve the generalized Sliding Tile Puzzle, focusing on the 4×4 (15-puzzle) and 5×5 (24-puzzle) cases in the experimental evaluation. This is a classic combinatorial optimization problem where the objective is to rearrange a grid of numbered tiles into a specific ordered configuration by sliding tiles into an empty space.
I modeled the problem formally as a state-space search. A state is represented as a flattened tuple of integers, `PuzzleState`, where 0 represents the blank tile and the integers $1 \ldots N^2 - 1$ represent the numbered tiles. The goal state is the "canonical" configuration where numbers are ordered sequentially (row-major), with the blank tile positioned at the very end (bottom-right corner).

To encapsulate the problem dynamics, I used a generic `Problem` class. The available actions $\{U, D, L, R\}$ are generated dynamically based on the current position of the blank tile, ensuring that moves respect the grid boundaries. The transition model (result function) returns a new immutable state tuple reflecting the swap between the blank and the target tile. The step cost is uniform, with every move assigned a cost of 1.
A critical aspect of the sliding puzzle is that the state space is partitioned: not all random permutations of the grid are reachable from the goal state. To address this and facilitate robust testing, I implemented a strict mathematical check, `is_solvable`, based on the parity of inversions and the row index of the blank tile. While `is_solvable` is used to validate instances produced by other generation modes (e.g., random permutations) or manual inputs, it is not needed for **Scrambling (Random Walk)** because solvability is guaranteed by construction (it starts from the goal and applies only legal moves). For the reported experiments, I generated instances using **Scrambling (Random Walk)**; the other instance-generation modes are implemented in the code but not used in the experimental section.

Scrambling (Random Walk): The function `generate_scrambled_instance` generates a random solvable instance by starting from the goal configuration and applying a sequence of valid actions $\{U, D, L, R\}$. Since each action is a legal slide of the blank and has an inverse ($U \leftrightarrow D, L \leftrightarrow R$), every generated state is guaranteed solvable by construction. To make shuffling more effective, the generator avoids immediately selecting the inverse of the previous move when alternatives exist. This is the method used in the experiments reported in this report (with `random_moves`=30 and varying seeds).

Default Demo: A fixed, easy-to-solve instance (`_default_demo_initial`) for quick debugging (implemented, not used in the reported experiments).

Manual Input: User-defined configurations passed via command line, which are strictly validated by the `is_solvable` check (implemented, not used in the reported experiments).

Random Permutation: The function `generate_random_permutation_instance` samples a random state and filters it via `is_solvable` (implemented, not used in the reported experiments).

Note on CSP modeling and code reuse. The CSP/Z3 approach uses the same underlying domain model described above (same state representation, same goal definition, same move semantics, and the same scrambling protocol for instance generation). In the implementation, the CSP script (`CSP.py`) explicitly reuses domain utilities from `implementation_A.py` (e.g., `PuzzleState`, `PuzzleAction`, `_goal_state_for_size`, `generate_scrambled_instance`, `is_solvable`, and the board formatter). This ensures that A* and CSP are evaluated on consistent instances and avoids duplicating domain logic across the two solvers.

5 Experiments

In this section, I present the experimental evaluation of the A* algorithm. The analysis is structured into two distinct phases to isolate different performance factors. First, I analyze the scalability of the algorithm: by fixing the heuristic strategy, I observe how the search performance (e.g., node expansions and runtime) changes when increasing the problem dimension from 4×4 to 5×5. This step is designed to understand how the problem difficulty grows with the board size. Second, I perform a direct comparative analysis between the two approaches: Misplaced Tiles and Manhattan Distance. In this phase, I evaluate both heuristics on the same instances to explicitly quantify the efficiency gains provided by the more informed Manhattan heuristic compared to the simpler Misplaced Tiles.

5.1 Experiments A*

In the next section i will explain all the experiment performed in A*, with the final result depicted here Tab 2.

5.1.1 Misplaced heuristic

I evaluated A* on generalized sliding puzzles using the misplaced tiles heuristic, defined as the number of tiles not in their goal position (excluding the blank, i.e., the empty cell represented by 0). For all experiments, I generated solvable instances by scrambling the goal state with `random_moves` = 30. Each

configuration was tested over `runs` = 10 independent instances (same generation protocol). The step cost is uniform (1 per move), and the reported `solution_cost_mean` corresponds to the length of the optimal solution found.

Results for size 5 (24-puzzle). For the 5×5 puzzle, A* with h = misplaced solved all 10 instances. The average solution cost was `solution_cost_mean` = 27.8, consistent with instances generated by 30 scrambling moves. However, the search effort was substantial: `expanded_mean` $\approx 79,967$ nodes and `generated_mean` $\approx 184,374$ nodes. Memory demand was also high, with `max_frontier_mean` $\approx 99,591$. The average runtime was `elapsed_sec_mean` ≈ 1.29 s. These values show that, although the heuristic preserves optimality, it provides limited guidance and A* explores a very large portion of the state space before reaching the goal.

Results for size 4 (15-puzzle). For the 4×4 puzzle, A* with h = misplaced again solved all 10 instances with `solution_cost_mean` = 26.0. In this case, the computational cost was even higher: `expanded_mean` $\approx 283,712$ and `generated_mean` $\approx 574,076$, with `max_frontier_mean` $\approx 261,012$. The average runtime increased to `elapsed_sec_mean` ≈ 5.67 s. Therefore, under the same scrambling depth (30 moves), the misplaced-tiles heuristic led to significantly heavier search on 4×4 than on 5×5 in my sampled instances.

Discussion and interpretation. The misplaced-tiles heuristic is admissible but relatively weak: it only counts how many tiles are incorrect, without measuring how far they are from their goal positions. As a consequence, many different states share the same heuristic value, creating large plateaus in the evaluation function $f(n) = g(n) + h(n)$. When plateaus are large, A* must expand many alternatives with similar f before it can make progress toward the goal, which increases both node expansions and frontier size.

A key point is the scale of the heuristic compared to the problem difficulty. Here the "scale of the problem" can be approximated by the average optimal cost C^* (how many moves are really needed to solve). The misplaced heuristic has a fixed maximum equal to the number of non-blank tiles: in 4×4, $h_{\max} = 15$ while I observed $C^* \approx 26$; in 5×5, $h_{\max} = 24$ while I observed $C^* \approx 27.8$. Thus, in the 5×5 experiments the heuristic values are "closer" to the typical optimal cost, meaning that h can contribute more to $f = g + h$ and can more often separate promising from unpromising states. This tends to reduce the size of f-value plateaus.

More concretely, since a single move swaps the blank with one tile, misplaced frequently changes by 0 (and otherwise by ± 1). This limited granularity causes many ties. For example, at the same depth ($g = 10$), it is common in 4×4 to have misplaced$(A) = 10$ and misplaced$(B) = 10$, yielding $f(A) = f(B) = 20$. In 5×5, it is more likely to observe different counts such as misplaced$(A) = 16$ and misplaced$(B) = 18$, giving $f(A) = 26$ and $f(B) = 28$; A* can then prefer the more promising node and avoid exploring many tied alternatives.

Finally, the fact that the 4×4 case appears "worse" than the 5×5 case for the same `random_moves` does not contradict the general observation that larger puzzles are harder. In my setting, a crucial factor is the scale of the heuristic relative to the typical optimal cost C^*. The misplaced-tiles heuristic has a fixed maximum equal to the number of non-blank tiles: $h_{\max} = 15$ for 4×4 and $h_{\max} = 24$ for 5×5. In my experiments, the average optimal costs were $C^* \approx 26$ (4×4) and $C^* \approx 27.8$ (5×5), meaning that in the larger puzzle the heuristic can reach values that are closer to C^* and therefore contribute more to $f = g + h$. This can reduce the number of f-value ties (plateaus) and make the search order more discriminative, which in turn may reduce node expansions. At the same time, `random_moves` does not guarantee equal difficulty across different board sizes; with different seeds or a different scramble depth, the observed gap could change.

Conclusion. Overall, the experiments confirm that A* with h = misplaced consistently finds optimal solutions, but it can be computationally expensive. The large values of `expanded_mean` and `max_frontier_mean` demonstrate that this heuristic provides limited guidance and may become impractical as instance difficulty increases, motivating the use of more informative heuristics such as Manhattan distance.

5.1.2 Manhattan Heuristic

Experimental setup (recap). I ran A* with the Manhattan-distance heuristic on generalized sliding puzzles, generating solvable instances by scrambling the goal with `random_moves` = 30. Each configuration was evaluated over `runs` = 10 independent instances. The step cost is uniform (1 per move), so the solution cost equals the number of moves in the returned plan. I report averages of: node expansions (`expanded_mean`), generated nodes (`generated_mean`), frontier size (`max_frontier_mean`), and runtime (`elapsed_sec_mean`).

1) What Manhattan measures and why it is informative. The Manhattan heuristic estimates remaining distance as:

$$h_{\text{manhattan}}(s) = \sum_{t \neq 0} (|r_t - r_t^*| + |c_t - c_t^*|)$$

where each tile contributes the number of grid steps needed to reach its goal position (ignoring interactions between tiles). Compared to misplaced, Manhattan has two major advantages:

- **Distance vs. Count:** It measures "how far", not only "how many" tiles are wrong. Two states may have the same number of misplaced tiles but very different true difficulty; Manhattan often distinguishes them.
- **Granularity:** It has finer granularity. A single move can change Manhattan by more than 1 (specifically ± 1 per tile dimension), and the total heuristic changes more consistently across states than misplaced. This typically reduces the number of ties in $f(n) = g(n) + h(n)$, producing fewer large plateaus.

These properties make Manhattan a much stronger guide for A*: it tends to push the search toward states that are geometrically closer to the goal.

2) Results for size 4 (15-puzzle, 4×4). Summary (`runs` = 10, scrambled with 30 moves):

- `solved` = 10/10
- `solution_cost_mean` = 26.0
- `expanded_mean` $\approx 2,834.6$
- `generated_mean` $\approx 5,673.4$
- `max_frontier_mean` $\approx 2,664.9$
- `elapsed_sec_mean` ≈ 0.0485 s

Interpretation: The mean optimal cost ($C^* \approx 26$) is reasonable for instances produced by a 30-step scramble. The search effort is moderate: a few thousand expansions per instance is already a big improvement compared to uninformed search. Memory usage is also controlled, with a peak frontier of $\sim 2.6k$ nodes, indicating that A* is not stuck exploring huge plateaus.

3) Results for size 5 (24-puzzle, 5×5). Summary (`runs` = 10, scrambled with 30 moves):

- `solved` = 10/10
- `solution_cost_mean` = 27.8
- `expanded_mean` $\approx 1,536.5$
- `generated_mean` $\approx 3,478.4$
- `max_frontier_mean` $\approx 1,893.4$
- `elapsed_sec_mean` ≈ 0.0334 s

Interpretation: The mean optimal cost ($C^* \approx 27.8$) is consistent with the scramble length. Despite the problem being "larger" (more tiles, bigger board), Manhattan keeps the expansions around $\sim 1.5k$ on average. Runtime is slightly lower than in the 4×4 case, mainly because the search explores fewer nodes.

4) Comparison between size 4 and size 5. A direct cross-size comparison must be interpreted carefully because `random_moves` = 30 does not guarantee the same difficulty distribution across different sizes. Still, within my sample:

- **Solution cost:** 5×5 is slightly higher (27.8 vs 26.0), meaning the sampled instances were marginally farther from the goal.
- **Search effort:** 5×5 required fewer expansions (1,536 vs 2,835). This suggests that, for the generated instances, Manhattan was very effective at prioritizing the right states early.
- **Frontier/memory:** 5×5 also kept a smaller peak frontier (1,893 vs 2,665).

A plausible explanation is that Manhattan tends to produce a strong and discriminative signal even as the board grows, often reducing tie situations in f-values. However, the most robust conclusion remains that Manhattan is efficient and scales well on these near-to-medium instances.

5) Manhattan vs Misplaced (Key Result). This comparison isolates the effect of the heuristic on the same size (5×5) and scramble regime (30 moves).

- **Optimality preserved:** `solution_cost_mean` = 27.8 for both.
- **Efficiency Gain:** The reduction in expansions is dramatic:

$$\frac{79,967 \text{ (misplaced)}}{1,536 \text{ (manhattan)}} \approx 52\times$$

  The reduction in time is similarly significant:

$$\frac{1.29 \text{ s}}{0.033 \text{ s}} \approx 39\times$$

- **Memory/Frontier:**

$$\frac{99,591}{1,893} \approx 52\times$$

Mechanism: $h_{\text{misplaced}}$ is coarse, creating large plateaus where A* behaves like uniform-cost search. $h_{\text{manhattan}}$ is fine-grained and correlates better with true remaining work, producing fewer ties and a search order that reaches the goal much faster.

6) Conclusion. Manhattan leads to optimal solutions and provides a massive efficiency gain in both time and memory. The improvement is especially evident when compared to misplaced, where expansions and frontier sizes are tens of times larger. The experiments confirm that more informed admissible heuristics dramatically reduce computational work while preserving optimality.

5.2 Experiments CSP/Z3

I evaluated the CSP/Z3 approach on a 4×4 instance generated by scrambling the goal state with `random_moves`=30 and `seed`=1. The solver was run with `max_steps`=40 and iterative deepening on the horizon.

Unlike the A* experiments, where I reported means over 10 runs, for CSP/Z3 I report a **single run** only. The reason is practical: solving even one non-trivial 4×4 instance at this scramble depth can take a very long time, making a 10-run average infeasible within reasonable time limits.

The CSP/Z3 solver found an optimal plan of cost 26 (matching A* with Manhattan on the same instance), but required many satisfiability checks (one per horizon) and therefore a much higher runtime. As shown by `elapsed_sec`, this single run took about 2648.5 seconds, i.e., approximately 44.1417 minutes, which is orders of magnitude slower than A* on instances of comparable depth. Observed metrics for this run:

- `cost` = 26
- `solver_calls` = 27 (horizons $T = 0 \dots 26$)
- `horizons_tried` = 27
- `elapsed_sec` ≈ 2648.5 s

6 Results

The main empirical conclusions are:

- **Heuristic quality dominates A* performance:** Manhattan yields the same optimal costs as misplaced but with dramatically fewer expansions and lower runtime.
- **CSP/Z3 is optimal but much slower on this domain:** on a representative 4×4 scrambled instance, CSP/Z3 returns the same optimal cost as A* Manhattan but requires many UNSAT horizon checks, leading to runtimes in minutes rather than milliseconds.

Table 1: CSP/Z3 (single run) vs A* Manhattan (mean over 10 runs). Both solve 4×4 instances with 30 scramble moves; CSP run uses seed 1.

Metric	**CSP/Z3** (4×4)	**A* Manhattan** (4×4)
Solution cost	26	26.0
Runtime (s)	≈ 2648.5	≈ 0.0485
Expanded nodes	n/a	$\approx 2,834.6$
Solver calls	27	n/a
Horizons tried	27	n/a

7 Instructions to run the code

A* solver (`implementation_A.py`). Single run on a scrambled instance:

```
python3 implementation_A.py --size 4 --heuristic manhattan --random-moves 30 --seed 1
```

Experiments (means over 10 runs, each run generates a new instance by varying the seed):

```
python3 implementation_A.py --size 4 --heuristic misplaced --random-moves 30 --runs 10 --seed 1
python3 implementation_A.py --size 4 --heuristic manhattan --random-moves 30 --runs 10 --seed 1
python3 implementation_A.py --size 5 --heuristic misplaced --random-moves 30 --runs 10 --seed 1
python3 implementation_A.py --size 5 --heuristic manhattan --random-moves 30 --runs 10 --seed 1
```

CSP/Z3 solver (`CSP.py`). The CSP solver depends on the `z3-solver` Python package. If your Linux environment restricts system-wide package installation, create a virtual environment:

```
python3 -m venv .venv
source .venv/bin/activate
pip install z3-solver
```

Run a single CSP instance (`--progress` prints per-horizon SAT/UNSAT attempts, useful for long runs):

```
python3 CSP.py --size 4 --random-moves 30 --seed 1 --max-steps 40 --progress
```

A Appendix - Tables

Table 2: Full performance metrics for A* (Misplaced vs Manhattan). All values are means over 10 runs with 30 random moves.

Heuristic	Size	Expanded	Generated	Max Front.	Max Expl.	Final Front.	Final Expl.	Time (s)	Cost
Misplaced	4×4	283,711.8	574,076.2	261,011.6	283,711.8	261,010.6	283,711.8	5.666	26.0
Misplaced	5×5	79,967.2	184,373.5	99,591.4	79,967.2	99,590.4	79,967.2	1.292	27.8
Manhattan	4×4	2,834.6	5,673.4	2,664.9	2,834.6	2,663.9	2,834.6	0.049	26.0
Manhattan	5×5	1,536.5	3,478.4	1,893.4	1,536.5	1,892.4	1,536.5	0.033	27.8

Metrics (A*):

Expanded: Number of nodes removed from the frontier and expanded (successors generated).

Generated: Total number of successor nodes produced (including those later discarded by duplicate elimination).

Max Front.: Peak size of the frontier (open list / priority queue).

Max Expl.: Peak size of the explored set (closed list).

Final Front.: Frontier size when the search stops (goal found).

Final Expl.: Explored-set size when the search stops.

Time (s): Total runtime in seconds. For a single run, elapsed_sec is the total wall-clock time to find the solution; in the "means over 10 runs" table, "Time (s)" is the average elapsed_sec across the 10 runs (elapsed_sec_mean).

Cost: Solution length (number of moves), with unit cost per move.