COSC 76: Artificial Intelligence, 25F, PA2: Mazeworld, Minh Nguyen

REPORT

Description

This assignment implements **A-star** search for two planning problems in grids:

1. **Multi/Single-Robot Coordination (deterministic)**
    - Robot(s) move one at a time on a known maze.
    - Robots cannot overlap or pass through walls.
    - Cost = total fuel = number of single-robot moves (waiting is allowed and free).
    - A* uses an **admissible Manhattan heuristic (sum across robots)**.
2. **Sensorless (Blind) Robot**
    - One robot starts with **unknown position** (belief = set of all floor cells).
    - Actions are global moves (N/E/S/W). If a move hits a wall, the robot stays put.
    - State in A* is the **set of all possible positions** (a belief state).
    - Heuristic = **max pairwise Manhattan distance** over the belief; admissible.

The codebase is modular and problem-agnostic: A* only needs a problem with:

- `start_state`
- `is_goal(state)`
- `get_successors(state) -> List[(next_state, step_cost)]`
- A heuristic function `h(state)`

Requirements

- Python 3.10+ (works with stock library only)
- Terminal/shell
- The `.maz` files in the same directory

> No third-party packages are required.

Files

. ├── `Maze.py` # Maze loader/renderer & collision queries ├── `MazeworldProblem.py` # Multi-robot modeling + heuristic + animation ├── `SensorlessProblem.py` # Sensorless (belief) modeling + heuristic + animation ├── `astar_search.py` # Generic A* (with parent/backchain) ├── `SearchSolution.py` # Printable result object (path, cost, nodes visited) ├── `test_mazeworld.py` # Example runs for multi-robot ├── `test_sensorless.py` # Example runs for sensorless ├── `*.maz` # Map files (ASCII) └── `README.md`

How to run

To run my code, for the mazeworld problem, run test_mazeworld.py.

To run my code, for the sensorless blind robot problem, run test_sensorless.py.

If it is difficult to test because the terminal output shows responses for all 5 tests, feel free to comment out other tests to individually test a specific test.

Design

Maze (Maze.py)

Loads ASCII map (`#` wall, `.` floor).

Optional `\robot x y` lines add robots for display; the problem models don't depend on them.

Methods: `is_floor(x,y)`, `has_robot(x,y)`, `__str__()` renderer.

Multi-Robot model (MazeworldProblem.py)

State: `(turn, x1, y1, x2, y2, ...)` where `turn ∈ {0..k-1}` is the index of the robot allowed to move now.

Actions: the current robot may move N/E/S/W or wait (0,0).

Constraints: must move to a floor cell; cannot move into another robot's cell.

Costs: `move = 1`, `wait = 0` (fuel model).

Goal: all `(xi, yi)` equal their targets.

Heuristic (manhattan_heuristic):

```
h(s) = Σ_i |xi - x_goal_i| + |yi - y_goal_i|
```

Admissible & consistent on 4-connected grids with unit step costs; ignores robot–robot blocking (relaxation), so it never overestimates.

Sensorless model (SensorlessProblem.py)

State (belief): `frozenset{(x,y)}` of all still-possible positions.

Actions: global moves; each position attempts the move, or stays if blocked. Next belief is the union of results.

Goal: `|belief| == 1`.

Heuristic (sensorless_heuristic):

`h(S) = max_{p,q∈S} (|px-qx| + |py-qy|)` — the max Manhattan spread.

This is admissible (optimistic) because collapsing the belief to a singleton can't take fewer steps than the farthest two positions are apart on a grid with unit moves (ignores walls → relaxation).

A* (astar_search.py)

Node priority `f = g + h` where g is the cumulative transition cost, h is heuristic.

Uses a **binary heap**; decrease-key handled by a `visited_cost` dictionary (push if we found a cheaper g).

`backchain(node)` reconstructs the path by parents and one reverse (`O(n)` time, `O(1)` extra space).

Examples

Corridor “wrong order” mazes show how free wait actions let robots pass without collisions.

In large open maps (40×40), A* with Manhattan runs well; ensure your goal cell is a floor, not a wall (see comments in the tests).

Sensorless belief often shrinks fastest with “sweeps” (e.g., repeated north until blocked, then east, etc.).

Correctness and Optimality

Multi-robot: With sum-Manhattan (consistent), A* returns an optimal fuel plan (fewest move actions) under the given costs.

Sensorless: Heuristic is admissible; with uniform step cost = 1, A* returns an optimal action sequence that localizes the robot (smallest number of actions).

Complexity

Let b be branching factor, d solution depth.

A* worst-case node expansions ~ O(b^d).

Multi-robot b: up to 5 (N/E/S/W/Wait); walls/robots reduce it.

Sensorless: State size ≤ number of floor tiles. Belief typically shrinks quickly in practice.

Assumptions/Limitation

Robots move sequentially; no simultaneous swaps.

Waiting costs 0 by design; to penalize waiting, change the cost in `get_successors`.

Rendering uses letters A..Z; very large beliefs will reuse letters (purely cosmetic).

`.maz` files must be rectangular and valid; border walls avoid out-of-bounds checks.

Troubleshooting

No path / failure: Verify goal coordinates are floor cells. In `maze_large.maz`, goals were adjusted from blocked (39,39) to (38,38).

Animation looks wrong: `animate_path` mutates `maze.robot_locations`; create a fresh Maze per test if you chain many runs.

Performance slow: Comment out animation; reduce print frequency; ensure heuristics are used (not the null heuristic).

Discussion Questions

A* with multiple robots

1. If there are k robots, how would you represent the state of the system? Hint -- how many numbers are needed to exactly reconstruct the locations of all the robots, if we somehow forgot where all of the robots were? Further hint. Do you need to know anything else to determine exactly what actions are available from this state?

-> `state = (turn, x1, y1, x2, y2, …, xk, yk)`, where turn is the #robot_turn, (xk, yk) is the location of robot k.

2. Give an upper bound on the number of states in the system, in terms of n and k.

-> `≤ k · (n²)^k`

3. Give a rough estimate on how many of these states represent collisions if the number of wall squares is w, and n is much larger than k.

-> `#collision states ≈ k · F^k · (1 − exp(−k(k−1)/(2F))) ≈ k · F^{k−1} · k(k−1)/2`

4. If there are not many walls, n is large (say 100x100), and several robots (say 10), do you expect a straightforward breadth-first search on the state space to be computationally feasible for all start and goal pairs? Why or why not?

-> No. Even with F ≈ 10,000, the (turn-annotated) state space size is on the order of `k · F^k ≈ 10 · (10^4)^{10} = 10 · 10^40`. BFS would need to store a frontier/explored set that explodes exponentially in depth—completely infeasible in time and memory. You need informed search (A*, good heuristics) and strong pruning.

5. Describe a useful, monotonic heuristic function for this search space. Show that your heuristic is monotonic. See the textbook for a formal definition of monotonic.

-> Heuristic: sum of Manhattan distances from each robot to its own goal, ignoring walls and other robots: `h(s) = Σ_i |xi − x*i| + |yi − y*i|`

Admissible: ignores obstacles and blocking, so it never overestimates the true minimal steps.

Monotonic/consistent: For any legal action a from state s to s':

Only one robot moves (turn model). A single 4-connected move changes that robot's Manhattan distance by at most 1, so h(s) ≤ 1 + h(s') for move cost c(s,a,s') = 1.

For a wait action (cost 0), positions don't change → h(s) = h(s'), so h(s) ≤ 0 + h(s'). Therefore h(s) ≤ c(s,a,s') + h(s') for all actions: consistent.

Robots interfere (can block each other and cannot share a cell). Independent BFS plans can deadlock or require reordering; planning in the joint state space is necessary to coordinate who moves when (my A* over the joint state handles this).

6. Describe why the 8-puzzle in the book is a special case of this problem. Is the heuristic function you chose a good one for the 8-puzzle?

-> The 8-puzzle is a 3×3 grid with 8 labeled tiles and one blank. Exactly one tile moves per step into the blank (others are impassable).

This matches the joint-robot model with:

k = 8 “robots” (tiles), plus one special “empty” cell,

Move legality: only the tile adjacent to the blank may move (i.e., constraints stronger than our generic 4-move options).

The heuristic we chose—sum of Manhattan distances of each robot/tile to its goal—is exactly the classic 8-puzzle heuristic. It is admissible and consistent for the 8-puzzle, so it’s a good choice here too (often improved by adding “linear conflict,” but plain Manhattan is standard and safe).

7. The state space of the 8-puzzle is made of two disjoint sets. Describe how you would modify your program to prove this. (You do not have to implement this.)

-> Theory: the 8-puzzle state space splits by permutation parity (even vs odd inversions in the tile order, ignoring the blank). Legal moves preserve parity, so one component is unreachable from the other.

Programmatic proof sketch:

Implement parity(state): Flatten tiles row-major (omit the blank), Count inversions; parity is inversions % 2.

From any start, run a search (BFS/DFS) and record parity for all reached states.

Show that: All reached states share the same parity as the start, Any hand-constructed state with the opposite parity is never reached.

8. Implement and Test

Optimality: Sum-Manhattan is consistent, so A* returns optimal fuel plans (fewest move actions; waits are free).

Why coordination matters: The wrong-order corridor case shows single-robot planning fails to resolve mutual blocking; joint A* naturally interleaves moves and waits.

Scalability: On 40×40, A* with Manhattan is fast in open spaces, while UCS is much slower; this demonstrates the impact of a good heuristic.

Edge cases: Be explicit that goals are on floor tiles (code already fixes the (39,39) wall case).

A* with blind robot

Describe what heuristic you used for the A* search. Is the heuristic optimistic? Are there other heuristics you might use? (An excellent might compare a few different heuristics for effectiveness and make an argument about optimality.)

1. Heuristic Used The heuristic implemented in `SensorlessProblem.sensorless_heuristic` is:

[h(S) = \max_{p, q \in S} \text{ManhattanDistance}(p, q)]

For the current **belief state** (S) — the set of all possible robot locations — this measures the **largest Manhattan distance** between any two possible positions.

Intuitively, it represents the **maximum spatial uncertainty** of the robot's location.

> "If my possible locations are scattered far apart, it will take at least that many moves to figure out where I actually am."

2. Why This Heuristic Is Optimistic (Admissible)

The heuristic is **optimistic** because it never overestimates the real number of moves required to localize the robot:

- It assumes the robot can move freely between any two points (ignores walls).
- Walls or obstacles in reality can only increase the number of steps needed.
- Therefore, the estimated cost ≤ actual cost.

Hence, A* using this heuristic is **admissible** and always finds the **optimal plan**.

3. Monotonicity (Consistency)

This heuristic is also **monotonic (consistent)** because:

- Each robot move changes every position in the belief by at most **one unit** (cost = 1).
- Therefore, the largest pairwise Manhattan distance can decrease by **at most 1** per move.

Formally:

[h(S) \le c(S, a, S') + h(S')]

for every action a, which satisfies the consistency condition required by A*.

4. How the Successor Function Works

The successor function applies the **same action** to every position in the current belief set.

- If a move is valid (no wall), the robot moves one cell in that direction.
- If it hits a wall, it stays in place.
- The new belief state is the **union** of all resulting positions.

The robot is now **more certain** about its position (fewer possibilities),
even though it has **no sensors**.
This is the principle of *localization through motion*.

5. Alternative Heuristics

a. Bounding Box Heuristic

[h(S) = (\max_x - \min_x) + (\max_y - \min_y)]

A simpler underestimate that ignores exact pairwise distances — cheaper to compute but less accurate.

b. Average Pairwise Distance

[h(S) = \text{mean}_{p,q \in S}(|x_p - x_q| + |y_p - y_q|)]

Smooths out spikes in uncertainty; still admissible but provides weaker guidance.

c. Wall-Aware Heuristic

Precompute the true shortest-path distances between all floor cells (via BFS) and use:

[h(S) = \max_{p,q \in S} \text{ShortestPathDistance}(p,q)]

This is the **most accurate** admissible heuristic (accounts for walls) but is computationally heavy for large mazes.

6. Optimality Argument

Because the heuristic is both:

- **Admissible** (never overestimates), and
- **Monotonic** (consistent with move costs),

A* expands nodes in nondecreasing order of `f = g + h`.

The first time it reaches a **singleton belief** (one remaining position), that plan is **optimal** — the shortest possible sequence of moves that guarantees full localization.

Summary Table

Aspect	Explanation
Heuristic	Maximum pairwise Manhattan distance between possible positions
Optimistic (Admissible)	Yes — ignores walls, never overestimates
Monotonic (Consistent)	Yes — spread decreases by ≤ 1 per move
Successor Behavior	Applies move to all positions; blocked ones stay put
Effect	Belief set shrinks as the robot moves; A* finds minimal plan
Alternative Heuristics	Bounding box, mean distance, wall-aware shortest-path

Note:

- It would ignore the \robot command lines
- For large maps with more than 26 floor spaces, it will produce weird ASCII characters so `animate_path` might look weird at first, but once it narrows down, it will look fine. Since the instructions did not state any specifics regarding this, I might want to suggest changing the alphabetical assignments to just `*`.