COSC 76: PA2 Optional Report

Wesley Tan

October 2024

1 A* Search Implementation

1.1 Description

- **AstarNode Class**: This class represents a node in the search tree, encapsulating essential attributes like the current state of the search, the node's heuristic value h (which is an estimate of the remaining cost to the goal), the parent node for backtracking purposes, and the transition cost g, which is the cost of reaching the current node from the start. Each node stores its cumulative cost, $f = g + h$, where g is the cost from the start to the current node, and h is the estimated cost from the current node to the goal.
- **Astar Search Function**: This function is responsible for carrying out the A* search algorithm, utilizing a priority queue (min-heap) to efficiently select the next node to expand. The priority queue is ordered by the nodes' f-values, ensuring that nodes with the lowest cumulative cost are expanded first. A dictionary named `visited_cost` keeps track of the best-known cost to reach each state, preventing unnecessary re-expansion of nodes if a cheaper path to the state is already known. Once the goal state is reached, the algorithm reconstructs the optimal path using backtracking by following the parent pointers from the goal node back to the start.

Key Design Decisions:

- A major design decision in A* search is selecting a heuristic that is both admissible and efficient. In this implementation, the heuristic is domain-specific and ensures that it never overestimates the cost, making it admissible and ensuring optimality.
- The use of a priority queue (heap) was chosen to ensure that node expansion happens in an order that is guided by the f-value, leading to faster convergence to the goal.
- Once the goal is found, backtracking from the goal to the start is implemented to recover the optimal solution path.

1.2 Evaluation

The A* search algorithm functions as intended, successfully finding the shortest path to the goal in all tested scenarios. Below are some key observations regarding its performance:

- The algorithm consistently finds optimal solutions, as the heuristic is admissible and ensures the shortest path.
- Due to the priority queue, the algorithm efficiently prunes unnecessary paths, expanding only the most promising nodes. The time complexity depends on the branching factor and the quality of the heuristic, but it has shown good performance in typical test cases.
- **Partial Successes**: In some cases with complex graphs, where the heuristic might not be as accurate (yet still admissible), the algorithm takes longer to find the optimal path, as more nodes need to be expanded. This could be improved with a more informed heuristic.

1.3 Discussion

1.3.1 Why Use A* ?

A* search is a more informed algorithm compared to Dijkstra because it incorporates a heuristic function to guide the search towards the goal faster. Dijkstra's algorithm is a special case of A* where the heuristic $h = 0$, which means A* has the potential to be more efficient when an admissible heuristic is available.

1.3.2 Does Your Heuristic Guarantee Optimal Solutions?

Yes, the heuristic used in the implementation is admissible (it never overestimates the true cost to the goal) and consistent (monotonic), which guarantees that A* will find an optimal solution if one exists.

1.3.3 What Are the Trade-offs of Using A*?

A* combines the advantages of both breadth-first search (for completeness) and uniform-cost search (for optimality), but its performance heavily relies on the heuristic. A poor heuristic can lead to the algorithm behaving like an uninformed search, expanding many nodes unnecessarily.

The algorithm works correctly in all cases tested. However, in large graphs with poor heuristic estimates, it may require more computation time due to the expanded search space.

2 Multi-Robot Coordination Problem

The multi-robot coordination problem is modeled in the `MazeworldProblem` class in `MazeworldProblem.py`. The key aspects of the implementation are:

- **State representation**: The state is represented as a tuple containing the current turn and the positions of all robots.
- **Successor generation**: Only the robot whose turn it is can move, considering collisions with walls and other robots.
- **Goal test**: Checks if all robots have reached their goal positions.

2.1 State Representation

To represent the state of the system, we need to track the positions of all robots. The state can be represented as a tuple:

$$(turn, x_1, y_1, x_2, y_2, \ldots, x_k, y_k)$$

where:

- `turn` indicates which robot's turn it is to move.
- (x_i, y_i) are the coordinates of robot i.

Additionally, we need to ensure that no robot occupies the same square as another. This can be achieved by checking the position of each robot before executing a move.

2.2 Upper Bound on Number of States

The upper bound on the number of states can be expressed as:

$$k \cdot n^{2k}$$

Where:

- k is the number of robots.
- n^2 is the number of possible positions for each robot on an $n \times n$ grid.

2.3 Estimating the Number of Collision States with Walls

Each robot can be placed in any of the N accessible squares. Since robots are distinguishable and collisions (robots occupying the same square) are allowed, the total number of possible configurations (states) is:

$$\text{Total States} = N^k$$

A collision occurs when at least two robots occupy the same square. We aim to estimate the number of configurations where this happens.

The number of unique pairs of robots that could collide is:

$$\text{Number of Robot Pairs} = \binom{k}{2} = \frac{k(k-1)}{2}$$

For each pair of robots, collisions can occur at any of the N accessible squares.

Multiplying the number of configurations per pair by the total number of robot pairs:

$$\text{Collision States} \approx \frac{k(k-1)}{2} \times N^{k-1}$$

When n is much larger than k, and considering w wall squares, the approximate number of collision states is:

$$\text{Collision States} \approx \frac{k(k-1)}{2} \times (n^2 - w)^{k-1}$$

This estimation shows that the number of collision states grows proportionally with the number of robot pairs and exponentially with the number of robots minus one, but is mitigated by the large number of accessible squares in the maze.

2.4 Feasibility of Breadth-First Search

For large mazes and several robots, a straightforward breadth-first search (BFS) becomes infeasible due to the size of the state space. For example, with $n = 100$ and $k = 10$, the state space size would be approximately:

$$100^{20} \times 10$$

This is computationally infeasible for BFS, and hence, more informed search methods, like A* with heuristics, are required.

2.5 Monotonic Heuristic Function

A useful heuristic function for the multi-robot coordination problem is the sum of the Manhattan distances from each robot to its goal position. This heuristic is admissible because the Manhattan distance is always less than or equal to the actual path cost, and it is also monotonic. For a heuristic to be monotonic, it must satisfy:

$$h(n) - h(n') \leq c(n, n')$$

where $c(n, n')$ is the cost of moving from state n to state n'. The Manhattan distance heuristic satisfies this condition because moving one robot by one step can only reduce the distance by one unit, which is equal to the cost of the move.

This heuristic is implemented in the `manhattan_heuristic` function:

```
def manhattan_heuristic(state, goal):
    total_distance = 0
    for i in range(1, len(state), 2):
        total_distance += abs(state[i] - goal[i]) +
        abs(state[i+1] - goal[i+1])
    return total_distance
```

This heuristic is monotonic because:

- It never overestimates the true cost to reach the goal.
- The difference in heuristic values between two adjacent states is always less than or equal to the actual cost of moving between those states.

For any two adjacent states s and s',

$$h(s) - h(s') \leq c(s, s')$$

where $c(s, s')$ is the cost of moving from s to s'. This holds because moving one robot by one step can only change the Manhattan distance by at most 1, which is equal to the cost of the move.

2.6 Why the 8-Puzzle is a Special Case of this Problem

The 8-puzzle is a special case of the *multi-robot coordination problem* because it involves moving multiple objects (in this case, tiles) within a constrained space to reach a goal configuration. In the multi-robot coordination problem, multiple robots need to move around a grid to achieve a specific formation or task, typically without collisions or overlapping movements. Similarly, in the 8-puzzle, the eight numbered tiles represent the "robots," and they must be moved to match the desired goal state, using the blank tile as the available movement space.

In both problems, the challenge is to coordinate the movements of several entities in a limited space. In the 8-puzzle, only the blank space can move adjacent tiles, mirroring how, in multi-robot coordination problems, a robot may need to wait for space to be available before moving. The blank tile serves as the "shared resource" that the numbered tiles use to rearrange themselves.

2.7 Is the Heuristic Function (Manhattan Distance) Good for the 8-Puzzle?

Yes, the heuristic function chosen for the multi-robot coordination problem, specifically the *Manhattan distance*, is a good heuristic for the 8-puzzle. The Manhattan distance calculates the sum of the vertical and horizontal distances each tile needs to move to reach its goal position.

For the 8-puzzle, the Manhattan distance is **admissible** because it never overestimates the minimum number of moves required to solve the puzzle. Each move involves shifting a tile either vertically or horizontally into the blank space, so the Manhattan distance directly correlates to the minimum number of moves necessary. Additionally, the Manhattan distance is **consistent (monotonic)**, meaning the estimated cost to reach the goal never decreases as we move from one state to another. Therefore, using this heuristic ensures that the A* search algorithm will always find the optimal solution.

2.8 Modifying the Program to Prove the State Space is Made of Two Disjoint Sets

The state space of the 8-puzzle is known to consist of **two disjoint sets** because not all initial configurations of the puzzle can be transformed into the goal configuration. These two sets are often referred to as *solvable* and *unsolvable configurations*. The solvability of a configuration depends on the number of inversions in the arrangement of the tiles. An inversion occurs when a larger numbered tile precedes a smaller numbered tile, and the sum of these inversions determines whether the puzzle can be solved.

To modify the program to prove this, the following steps could be implemented:

1. **Count Inversions**:
    - Flatten the 3x3 grid into a one-dimensional list, ignoring the blank space.
    - Count how many times a larger number precedes a smaller one. This count gives the number of inversions.
2. **Determine Solvability**:
    - If the inversion count is even, the configuration is solvable.
    - If the inversion count is odd, the configuration is unsolvable and belongs to the disjoint set that cannot reach the goal.
3. **Modify the Search Algorithm**:
    - Before running the A* search, calculate the inversion count for both the start and goal configurations.
    - If one configuration is solvable and the other is not, the program can immediately return that no solution exists, as they belong to different disjoint sets.
    - This can be done as a pre-check to avoid running the A* search unnecessarily for unsolvable configurations.

By adding this inversion check, the program can validate whether the puzzle is solvable or if the start and goal states belong to different disjoint sets. Although this modification doesn't require changing the core A* search logic, it provides an important optimization and theoretical proof of the disjoint state spaces.

3 Blind Robot Problem

The blind robot problem is implemented in the `SensorlessProblem` class in `SensorlessProblem.py`. The approach involves:

- **State representation**: A `frozenset` of all possible robot positions.
- **Successor generation**: Apply moves to all possible positions, keeping only valid moves.
- **Goal test**: Check if the state contains only one possible position.
- **Custom heuristic**: Uses the number of possible positions minus 1 as a heuristic.

The implementation follows these steps:

- Initialize the belief state with all possible locations in the maze.
- For each action, generate the successor belief state by simulating the movement of the robot.
- Use the A* search algorithm with a suitable heuristic to find a sequence of actions that reduces the belief state to a single location.

3.1 State Transition Example

Consider an initial state where the robot knows its x coordinate but not its y coordinate, i.e., its state is $\{(3,0),(3,1),(3,2)\}$. If the robot moves north, the new state depends on whether there is a wall north of these positions:

$$\text{north move:}\{(3,1),(3,2)\}$$

If the robot bumps into a wall at $(3,2)$, its position remains unchanged, and the state becomes:

$$\{(3,1),(3,2)\}$$

Two heuristics are implemented to guide the A* search:

3.2 Manhattan Distance Heuristic

The **Manhattan distance heuristic** is used to estimate the distance between possible robot locations. It calculates the maximum Manhattan distance between any two locations in the belief state. The Manhattan distance between two points (x_1, y_1) and (x_2, y_2) is given by:

$$d_{\text{Manhattan}} = |x_1 - x_2| + |y_1 - y_2|$$

This heuristic captures the spread of possible robot locations, encouraging moves that reduce the uncertainty by collapsing this spread.

Optimism: This heuristic is optimistic (admissible) because it never overestimates the true cost to reduce the set of possible locations to a single state. The actual cost will always be at least as large as the maximum Manhattan distance between the possible locations.

3.3 Custom Heuristic

The **Custom heuristic** used estimates the number of moves required to reduce the belief state to a single location. It is computed as:

$$h_{\text{custom}} = |\text{number of possible locations}| - 1$$

This heuristic counts the number of possible positions and assumes that each move will reduce the number of locations by one.

Optimism: The custom heuristic is also optimistic, as it assumes the robot can always reduce its uncertainty by eliminating one possible location per move. In practice, this might not always be the case, but the heuristic never overestimates the number of moves required.

Heuristic	Maze	Nodes Visited	Solution Length, Cost	Comments
Manhattan	Maze 1 and Maze 2	17	6, 5	Efficient performance with minimal node visits.
	Maze 3	77	9, 8	Increased node visits in a larger maze, but still efficient.
Custom	Maze 1 and Maze 2	19	6, 5	Slightly more nodes than Manhattan, but still efficient.
	Maze 3	25	9, 8	Performs better than Manhattan, visiting fewer nodes in larger maze.
Null	Maze 1 and Maze 2	26	6, 5	Least efficient, with significantly more nodes visited.
	Maze 3	855	9, 8	Extremely inefficient in larger maze, visiting far more nodes than necessary.

Table 1: Performance Comparison of Heuristics

Both the Manhattan and custom heuristics are admissible, ensuring that A* search will find the optimal solution. However, the Manhattan heuristic might lead to fewer nodes being expanded in practice due to its ability to estimate the uncertainty more precisely.

3.4 Alternative Heuristics

- **Maximum Pairwise Distance Heuristic**

  This heuristic calculates the maximum Manhattan distance between all pairs of possible robot locations in the belief state. It provides an estimate of the "spread" of the belief state.

  Mathematical Expression:

  Let B be the belief state, which is a set of possible positions $\{(x_i, y_i)\}$. The heuristic $h(B)$ is defined as:

$$h(B) = \max_{(x_i,y_i),\ (x_j,y_j)\in B} (|x_i - x_j| + |y_i - y_j|)$$

Python Implementation:

```
def max_pairwise_distance_heuristic(state):
    positions = list(state)
    max_distance = 0
    for i in range(len(positions)):
        for j in range(i + 1, len(positions)):
            x1, y1 = positions[i]
            x2, y2 = positions[j]
            distance = abs(x1 - x2) + abs(y1 - y2)
            if distance > max_distance:
                max_distance = distance
    return max_distance
```

- **Perimeter of Area Heuristic**

  This heuristic calculates the perimeter (or area, but perimeter is more appropriate for movement cost estimation) of the smallest axis-aligned rectangle that contains all possible robot locations. It estimates the spread of the belief state.

  Mathematical Expression:

  Let B be the belief state. Define:

$$x_{\min} = \min_{(x,y)\in B} x, \quad x_{\max} = \max_{(x,y)\in B} x$$
$$y_{\min} = \min_{(x,y)\in B} y, \quad y_{\max} = \max_{(x,y)\in B} y$$

  The heuristic $h(B)$ is then:

$$h(B) = (x_{\max} - x_{\min}) + (y_{\max} - y_{\min})$$

  Python Implementation:

```
x_values = [x for x, y in state]
y_values = [y for x, y in state]
x_min, x_max = min(x_values), max(x_values)
y_min, y_max = min(y_values), max(y_values)
return (x_max - x_min) + (y_max - y_min)
```

In terms of effectiveness, the Manhattan distance heuristic tends to guide the robot effectively by focusing on reducing the spatial spread of possible locations, making it highly useful in small to medium-sized mazes. The custom heuristic, on the other hand, is simpler but also effective in reducing the belief state, particularly when there are fewer possible locations.

For larger mazes or more complex belief states, the *maximum pairwise distance* heuristic may provide a more fine-tuned measure of uncertainty, potentially reducing the number of nodes visited during search. However, its higher computational cost might offset this benefit in some cases.

4 Evaluation

4.1 A* Search Performance

The A* search implementation shows good performance, as evidenced by the test results in `test_mazeworld.py` and `test_sensorless.py`. The algorithm successfully solves various maze problems, including multi-robot coordination and the blind robot problem.

4.2 Multi-Robot Coordination Results

The multi-robot coordination problem is tested with different scenarios in `test_mazeworld.py`:

- Single robot navigation.
- Multiple robot coordination.
- Unsolvable problems.

Results show that A* search with the Manhattan distance heuristic performs better than the null heuristic, visiting fewer nodes to find the solution.

4.3 Blind Robot Problem Results

The blind robot problem is tested in `test_sensorless.py` with three different mazes. The results show:

- The custom heuristic (number of possible positions - 1) generally performs better than the null heuristic.
- The algorithm successfully reduces the set of possible positions until a single position is determined.

Performance varies depending on the maze complexity and size.

5 Extensions

5.1 Multirobot coordination

Code found in file `BONUS_simultaneous.py`

5.2 Proof of Existence of a Plan

Problem Setup: Let $M = (V, E)$ be a finite maze represented as a graph, where V is the set of cells (locations) and E is the set of edges connecting adjacent cells (possible movements). The robot has no sensors and does not know its initial position, so its initial belief state is $B_0 = V$. The goal cell $g \in V$ is in the same connected component as the initial belief state.

Definition of Belief State: At any time t, the belief state $B_t \subseteq V$ represents the set of possible locations where the robot could be. An action a_t (moving north, south, east, or west) updates the belief state according to the transition function:

$$B_{t+1} = \{v' \in V \mid \exists v \in B_t \text{ such that } (v, v') \in E_a\},$$

where $E_a \subseteq E$ are the edges corresponding to action a_t.

Claim: A plan $\pi = [a_1, a_2, \ldots, a_k]$ exists such that after executing π, the belief state B_k satisfies $B_k = \{g\}$.

Proof:

Since the maze is finite and connected, we can construct a tree T rooted at the goal g. This tree connects all cells in the connected component containing g. We define a sequence of actions that systematically moves the robot towards the goal along the edges of T.

At each step, the robot performs the following:

1. **Action Selection:** Choose an action a_t that moves towards the goal in T. Since T is a tree, there is a unique simple path from any cell to g.

2. **Belief State Reduction:** After performing a_t, the belief state is updated to $B_{t+1} = \delta(B_t, a_t)$. Cells from which the action would have resulted in an invalid move (e.g., moving into a wall) are eliminated from B_{t+1}.

3. **Convergence:** Since T is finite and connected, and each action moves the robot closer to g along T, the belief state B_t will shrink over time. Invalid positions are eliminated at each step.

Because the maze is finite, and at each step the belief state either remains the same or decreases in size, after a finite number of steps k, the belief state will reduce to a singleton set containing only the goal: $B_k = \{g\}$. Therefore, a plan exists that leads the robot to the goal.

5.3 Polynomial-Time Motion Planner

We aim to design a motion planner that finds a plan π in time polynomial in $|V|$, the number of cells in the maze, despite the exponential size of the belief space (which has $2^{|V|}$ possible states).

Strategy:

1. Represent the belief state B_t as a subset of V. Operations on belief states involve set operations, which can be performed in polynomial time relative to $|V|$.

2. At each step, select an action a_t that maximizes the reduction of the belief state. This can be done by evaluating, for each possible action, the expected size of the updated belief state B_{t+1}.

3. **Belief State Update:** For each action a_t, compute the new belief state:

$$B_{t+1} = \{v' \in V \mid \exists v \in B_t \text{ such that } (v, v') \in E_a\}.$$

This computation involves checking the adjacency of cells and can be done in $O(|E|)$ time.

4. Repeat steps 2 and 3 until B_t is reduced to $\{g\}$.

Time Complexity Analysis:

At each step, the most time-consuming operation is updating the belief state, which takes $O(|E|)$ time. The maximum number of steps required is $|V|$, as the belief state cannot be reduced more than $|V|$ times. Therefore, the total time complexity is $O(|V| \cdot |E|)$.

Since in a planar maze, $|E| = O(|V|)$, the overall time complexity simplifies to $O(|V|^2)$, which is polynomial in the number of cells. In practice, this could look like the following, where B is the belief state:

```
def select_action(self, B: Set[Tuple[int, int]]) -> str:
    best_action = None
    min_new_belief_size = float('inf')
    for action in ['north', 'south', 'east', 'west']:
        new_B = self.update_belief_state(B, action)
        if len(new_B) < min_new_belief_size:
            min_new_belief_size = len(new_B)
            best_action = action
    return best_action

def plan(self) -> List[str]:
    B = self.V.copy()
    plan = []
    while len(B) > 1 or self.goal not in B:
        action = self.select_action(B)
        plan.append(action)
        B = self.update_belief_state(B, action)
    return plan
```