1 Implementation

1. *MazeWorldProblem*

   This is where the original maze problem is implemented, in which it establishes the game rules and tracks the maze object from the pre-loaded maze file. It tracks the start and current states of the robots and uses the *get_successors* function to determine the legal successor states of a given state. It uses the helper function *move_robot* to perform the moves and *is_valid_action* to check if they are possible. It also includes the implementation of *manhatan_heurisitc*, which aggregates the Manhattan distance of each robot from their goal locations.

2. *SensorlessProblem*

   This class is designed to solve a maze where a robot, unaware of its initial position, must navigate to find its location. The maze and potential robot locations are initialized, with each non-wall cell in the maze representing a possible starting point for the robot. Successor states are generated in *get_successors* along with the helper function *move_robots* by attempting to move the robot in all four directions, ensuring that it doesn't move outside the maze's boundaries. The goal is achieved when the set of possible robot locations is narrowed down to one specific cell. A heuristic, returning the count of possible locations, and an animate function for visualizing the solution path, are also included. A test instance of the SensorlessProblem class is created using a specified maze, ready for use with search algorithms to identify the robot's exact location within the maze.

3. *astar_search*

   The AstarNode class encapsulates a state in the search space, its associated heuristic value, parent node, and transition cost. Each node's priority is determined by the sum of its heuristic value and transition cost. The *astar_search* function conducts the search utilizing a priority queue. Nodes are extracted from the priority queue, and their successors are generated and evaluated. The *visited_cost* dictionary keeps track of the cost to reach visited states to avoid re-exploration and ensure optimal paths. When the goal state is reached, the *backchain* function is called to trace back through the parent nodes to construct the solution path. The *SearchSolution* object is then populated with the path and cost information and returned. The code allows for different search problems and heuristic functions to be plugged in as needed.

2 Testing

1. *test_mazeworld*

   I tested the implementation of MazeworldProblem in this file, using several test cases along with animations for the robot paths, with the first three being a single robot, two, then three. I also generated my own mazes using my *build_maze* function to specify maze dimensions, along with valid start and goal positions. The 10x10 maze also doesn't always have a solution, so you can rerun it to explore new options for mazes and initial/goal locations. The larger mazes also take awhile to run, so I commented them out for now.

2. *test_sensorless*

   I tested the implementation of SensorlessProblem in this file, using several test cases with animations for the robot paths, with the first three being a single robot, two, then three. I also generated my own mazes 4 and 8 to test out the rules and implementation.

3 Discussion Questions

1. **If there are k robots, how would you represent the state of the system?**

   - The state of the system could be represented as a tuple, in which the first element is the index of the robot whose turn it is, followed by the (x, y) coordinates of each robot. Thus, for k robots, the state would be $(curr\_robot, x_1, y_1, x_2, y_2, ..., x_k, y_k)$

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

   If we are considering "legal" states in which robots can't be in the same location:

   - There are $n * n$ squares in the maze
   - The number of ways to place the first robot is n^2
   - For the second robot, it's $n^2 - 1$ since one square is already occupied, then $n^2 - 2$ for the third, and so on
   - Thus, for k robots, we have an upper bound of $n^2 * (n^2 - 1) * ... * (n^2 - k + 1)$

   If we are considering all possible states, then there are $n * n$ squares in the maze and k robots, so the upper bound would be n^{2k}.

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**

   - A collision occurs when two or more robots occupy the same square. For any given square, the number of ways we can choose 2 robots from k robots to place on that square (i.e., cause a collision) is given by the binomial coefficient: $\binom{k}{2}$
   - Thus, given there are $n^2 - w$ accessible squares, the number of collision states for any given square is $\binom{k}{2}$
   - The total number of collision states across the entire grid is thus $(n^2 - w) * \binom{k}{2}$

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

- Given this case, it wouldn't be computationally feasible for all start/goal pairs as BFS would have to explore a massive search space. Using our earlier calculation, the number of state would be on the order of $10,000^{20}$.

5. **Describe a useful, monotonic heuristic function for this search space. Show that your heuristic is monotonic**

    - A good heuristic for this problem would be the Manhattan distance, which is the sum of the absolute differences of the x and y coordinates between the robot's current position and the goal. In the context of the maze, since each move has a constant cost of 1/step and the Manhattan distance only decreases or stays the same with each move towards the goal, it ensures our heuristic is monotonic.

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?**

    - The 8-puzzle is a special case because it represents a 3x3 maze with 9 tiles, one of which is blank. The tiles' movements can be equated to the robots' movements in the maze. The Manhattan distance heuristic is suitable for this problem as it gives an estimate of the minimum moves required to reach the goal state by summing up the distances each tile is from its goal position.

7. **The state space of the 8-puzzle is made of two disjoint sets. Describe how you would modify your program to prove this.**

    - The two disjoint sets refer to the solvable and unsolvable configurations of the 8-puzzle problem. In order to prove this, we can simulate the state space of the 8-puzzle by using Mazeworld to determine which states belong in which of the two sets. For instance, if we find a solution for a given start state, then we can conclude that the start state and all the states along the solution path belong within the solvable set. However, if the start state does not result in a solution, then all of those states belong in the unsolvable set. We could perform this for the entire search space to create both the solvable and unsolvable state sets.

8. **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.)**

    - The heuristic I used was the cardinality of the belief state set (i.e., the number of possible states). This function assumes that a robot is able to be in any of those states. The Manhattan distance is the sum of the horizontal and vertical distances between two points on a grid. In this specific implementation, the Manhattan distance is computed element-wise for each component of the state and *goal_locations* and then summed up.
    - Yes, the Manhattan distance is optimistic (or admissible) in grid-based problems. It always underestimates or equals the actual cost to reach the goal because it

does not consider obstacles that might be in the way, meaning it assumes that you can always move directly towards the goal. For a heuristic to be admissible, it must never overestimate the cost to reach the goal.

- Another heuristic that could be used is the Euclidean Distance Heuristic, which uses the straight-line distance between two points. It's more accurate than the Manhattan distance in scenarios where diagonal movement is allowed. However, it is not always admissible in grid environments where only horizontal and vertical movements are allowed.

4 Literature Review

Note: I am not a graduate student

Source: Standley, T. 2010. Finding optimal solutions to cooperative pathfinding problems. In The Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI'10), 173–178.

The study delves into the challenges of cooperative pathfinding, aiming to ensure that multiple agents, such as robots, navigate without mutual interference. Traditional methodologies often struggle with multiple agent scenarios. The researchers introduced two pivotal strategies: operator decomposition for efficient searching and independence detection for individual agent problem-solving. Comparative tests against benchmark methods revealed the superior efficacy of their approach. This research significantly advances the field of multi-agent pathfinding solutions.

5 Bonus Feature

I created a *build_maze* function that constructs a maze of a specified width and height, and then randomly determines wall locations based on a given probability (*wall_prob*). The maze is saved to a file with a naming convention "$maze[width]x[height].maz$". Open spaces within the maze are represented by the character ".", while walls are represented by "#". After generating the maze layout, the function places between 1 to 3 robots in random open locations, indicating their start positions in the file below the maze itself. Additionally, for each robot, a corresponding goal location is randomly selected and stored in a list. The function then closes the maze file and returns the filename, the number of robots, and a tuple containing the goal locations for each robot. Sometimes, the function will generate a maze that is unsolvable, so you can just re-run it to make one that does work.