Mazeworld

Carter Kruse (September 27, 2023)

Instructions

Here is a maze, drawn in the venerable tradition of ASCII art:

```
.......
.##....
..##...
.......
..##...
#.###..
....##.
```

The periods represent open floor tiles (places that can be walked on or over), and the number signs represent walls, places in the maze where a simple robot cannot go.

There is a robot. It can move in any of four directions: North (towards the top of the screen), East, South, and West. There are coordinates on the maze. `(0, 0)` is the bottom left corner of the maze. `(6, 0)` is the bottom right corner of the maze.

In this particular example, perhaps we'd like to plan a path for the robot to go through the maze from the bottom left corner to the bottom right corner, without going through any walls. There is code provided (`Maze.py`) that will do things like load in mazes, print them out, and ask simple questions, including whether there is a robot or a floor tile at a particular location (there might be both).

As a warm-up, create some example mazes, and use a simple BFS to search for a path from start to goal. You are welcome to use any code from the solution to the first assignment. This is just a warm-up, the real work begins with A*, multi-robot path planning, and the blind robot problem.

Report

The report is written in Markdown. There is no detailed description of what should be in the report, as it's up to the student to describe the implementation and results clearly and convincingly. The Markdown, PDF, and Python source files are submitted in a bundle as a single ZIP. There is a short README file that explains how to run the code.

A-Star Search

The A* search algorithm is implemented in `astar_search.py` (in Python). The description of the algorithm is provided in the textbook.

Initially, uniform-cost search was implemented, which is similar to A*, yet without the heuristic. Figure 3.14 in the book shows the algorithm. As indicated, there is a bit of a problem, specifically with the following step near the end:

```
if child.state is in frontier with higher path-cost
    replace that frontier node with child
```

How do we efficiently check if a node is in a priority queue, or replace it efficiently? It depends on the data structure used to implement the priority queue. A standard choice is a heap, which is an array that maintains items in a particular order based on their priority. (No, it's not an array sorted by priority. That would be hard to insert new items into it in a time-efficient manner. Heaps are more clever than that.)

Section 8.4.2 of the Python `heapq` documentation discusses exactly this issue, though in a slightly different context. The suggested solution is to mark the higher-cost item as removed, and add a new item with a lower cost. It should be pointed out that this operation does incur some computational cost when compared to the approach in the book of removing the higher-cost item. The higher-cost item stays in the heap, and as items are added to the heap, the increased size of the heap will mean increased heapify costs.

The most efficient approach appears to use a Fibonacci heap, but we will not discuss Fibonacci heaps in this class. Here's an interesting discussion at StackOverflow.

An acceptable substitute for the purpose of this class is to do the marking as suggested by the Python documentation. How do we mark things? If a state is being considered for addition to the frontier, check if it has been visited for a cheaper cost already. If not, add it to the frontier, and update the cost in a dictionary of state costs. Otherwise, do not add the node to the frontier.

A-Star Implementation

The A* search algorithm is implemented in `astar_search.py` (in Python). While the file contains quite a few comments, we will quickly review the structure here.

There is a node which contains the state, heuristic, parent, and transition cost, while defining the priority of the given node for the heap operations. Further, a comparison operator is included for the purpose of using the node within the heap.

```
# The AstarNode class is useful to wrap state objects, pointing to parent nodes.
class AstarNode:
```

```
    # Each AstarNode except the root has a parent node and wraps a state object.
    def __init__(self, state, heuristic, parent = None, transition_cost = 0):
        self.state = state
        self.heuristic = heuristic
        self.parent = parent
        self.transition_cost = transition_cost

    def priority(self):
        # The priority is the sum of the heuristic and transition cost.
            # The transition cost represents the actual cost from the start node.
            # The heuristic represents the estimated cost to the goal node.
        return self.heuristic + self.transition_cost

    # Comparison Operator
    # This is needed for 'heappush' and 'heappop' to work with nodes.
    def __lt__(self, other):
        return self.priority() < other.priority()
```

As with BFS and DFS, we have a backchaining algorithm that returns the appropriate path.

```
# Backchaining
# Take the current node and follow its parents back as far as possible.
# Get the states from the nodes, and reverse the resulting list of states.
def construct_solution_path(current_node):
    # Backtrack from the current node (presumably the goal node) to construct the solutic
    path = []

    # Cycle through until the root (start node) is reached.
    while current_node: # OR while current_node is not None:
        # Append the 'state' to the path. The 'append' function is faster than 'insert'.
        path.append(current_node.state)

        # Update the pointer of the current node.
        current_node = current_node.parent

    # We should reverse the path at the end, as we use the 'append' function.
    return path[::-1] # OR path.reverse()
```

The actual algorithm for A* is implemented in a single function, given as follows. This follows clearly from pseudocode for the uniform-cost search, which is essentially A* without including the heuristic. As indicated by the problem set-up, we take care to construct the `visited_cost` dictionary in such a way that we only push nodes to the heap that minimize the cost.

```
# A* Search
def astar_search(search_problem, heuristic_function):
    # Initialize the solution, given the search problem and heuristic.
```

```
    solution = SearchSolution(search_problem, "A*" + " " + "(" + heuristic_function.__nam

    # Determine the initial state of the search problem, using an instance variable.
    initial_state = tuple(search_problem.start_state)

    # Create an AstarNode with the appropriate initial state and heurisitic function.
    initial_node = AstarNode(initial_state, heuristic_function(initial_state))

    # Initialize the priority queue (ordered by cost) and push the initial node.
    priority_queue = []
    heappush(priority_queue, initial_node)

    # Creating the 'visited_cost' dictionary and updating it with the initial state.
    visited_cost = {}
    visited_cost[initial_state] = initial_node.priority()

    # While the priority queue contains values...
    while priority_queue:
        # Pop the node from the priority queue.
        current_node = heappop(priority_queue)

        # Increment the counter.
        solution.nodes_visited += 1

        # If the current state is already within the dictionary and has a higher priority
        if current_node.state in visited_cost and current_node.priority() > visited_cost[
            continue

        # Check if the current state is the goal state.
        if search_problem.is_goal_state(current_node.state):
            # If so, construct the solution path and return the solution.
            solution.path = construct_solution_path(current_node)

            # The solution cost may be considered the "fuel" cost or the transition cost
            solution.cost = len(set(robot_locations[1:] for robot_locations in solution.p
            # solution.cost = current_node.transition_cost

            return solution

        # Generate the successor states and add them to the priority queue.
            # This could be broken down into determining the appropriate actions/transiti
        for next_state in search_problem.get_successors(current_node.state):
            # Create a new AstarNode with the appropriate
            next_node = AstarNode(next_state, heuristic_function(next_state), parent = cu

            # If the next state has not been visited or is in the frontier with a higher
            if next_state not in visited_cost or next_node.priority() < visited_cost[next
                # Update the dictionary with the appropriate priority/cost.
                visited_cost[next_state] = next_node.priority()
```

```
                # Push the next node to the heap.
                heappush(priority_queue, next_node)

    # If the frontier is empty and no solution is found, return the solution.
    return solution
```

Multi-Robot Coordination

$ k $ robots live in an $ n \times n $ rectangular maze. The coordinates of each robot are $ (x_i, y_i) $, and each coordinate is an integer in the range $ 0 \ldots n - 1 $. For example, maybe the maze looks like this:

```
. B . . . . .
. # # . C . .
. . # # . . .
. . . . # . .
. . # # . . .
. . # . . . .
A . . . # # .
```

That's three robots `A`, `B`, `C` in a $ 7 \times 7 $ maze. You'd like to get the robots to another configuration. For example:

```
. . . . . . .
. # # . . . .
. . # # . . .
. . . . # . .
. . # # . . B
. . # . . . A
. . . . # # C
```

There are some rules. The robots only move in four directions, north, south, east, and west. The robots cannot pass through each other, and may not occupy the same square. The robots move one at a time. First robot `A` moves, then robot `B`, then robot `C`, then `D`, `E`, and eventually, `A` gets another turn. Any robot may decide to give up its turn and not move. So there are five possible actions from any state.

Let's make the cost function the total fuel expended by the robots. `A` robot expends one unit of fuel if it moves, and no fuel if it waits a turn. As a potential further problem, we may consider the case where the robots move simultaneously and/or the cost metric is a measure of the shortest total time.

Only one robot may occupy one square at a time. You are given a map ahead of time, and it will not change during the course of the game.

Multi-Robot Coordination Implementation

The Mazeworld problem is implemented in `MazeworldProblem.py` (in Python). While the file contains quite a few comments, we will quickly review the structure here.

The constructor allows us to keep track of the start state and the goal state, according to the problem statement. As indicated, other aspects of the problem are added, including the number of robots.

```
# Constructor
def __init__(self, maze, goal_locations):
    self.maze = maze
    self.goal_locations = goal_locations

    # The initial state is determined according to the maze characteristics.
    self.start_state = (0,) + tuple(maze.robotloc)

    # The number of robots is determined according to the goal locations.
    self.num_robots = len(goal_locations) // 2
```

The `get_successors` method is responsible for determining the appropriate states that follow, given an initial state.

The `for` statement considers all of the possible moves, and then determines if the state/position is safe, according to the constraints of the game. If so, we append the new value to the list.

Further, we introduce randomization of the moves to ensure that the algorithm functions for all possible combinations.

```
# Determine the successor states (as a list) for a given state.
    # Consider the valid moves from the current state and update the robot's position acc
def get_successors(self, state):
    # Initialize the list of successors, which starts out as empty.
    successors_list = []

    # Enumerate the possible "moves" for each of the robots, according to the problem set
    possible_moves = [(0, 0), (0, -1), (0, 1), (-1, 0), (1, 0)]

    # Introduce randomness to allow for consecutive iterations to be independent of a det
    random.shuffle(possible_moves)

    # Cycle through the list of possible moves.
    for move in possible_moves:
```

```
            # Creating the next state, according to the appropriate transition.
            next_state = (int((state[0] + 1) % self.num_robots),)

            # Creating a list to modify the position of a specific robot.
            state_list = list(state[1:])

            # The robot to move is given by 'state[0]', and modifications are given as follow
            state_list[(2 * state[0])] += move[0]
            state_list[(2 * state[0]) + 1] += move[1]

            # Creating the next state based on the robot to move and the locations of the rob
                # The first robot moves, then the second, and so on.
            next_state += tuple(state_list)

            # Check if a given state (determined by an initial state and transition) is safe.
            if self.is_safe(next_state):
                successors_list.append(next_state)

        return successors_list
```

The following helper function is used to determine if a given state is safe, and thus is added to the list. For this problem, there are a couple of situations that are important to check. First, we must consider if the robot is located on a floor spcae. Next, we have to check if the robots are in collision with one another.

```
    # Test if a given state is safe (before adding to successor list).
    def is_safe(self, state):
        robot_locations = set()

        for i in range(1, len(state), 2):
            # Wall/Floor Constraints: Check if the robots are on a floor space.
                # The boundary constraints are checked within the 'is_floor()' method.
            if not self.maze.is_floor(state[i], state[i + 1]):
                return False

            # Collision Constraints: Check if the robots are in collision with each other.
                # The 'has_robot()' method is not used, as this does not allow for no movemer
            if (state[i], state[i + 1]) not in robot_locations:
                robot_locations.add((state[i], state[i + 1]))
            else:
                return False

        return True
```

Further, there is a method used to check if the current state is the goal state, which returns a boolean value.

```
# Test if the state is at the goal.
def is_goal_state(self, state):
    if state[1:] == self.goal_locations:
        return True
    return False
```

The to_str statement was already written, but it is simply responsible for displaying what the problem is, along with the start state. This fully defines the problem, given the problem set-up. We know the goal state, and we do not need to consider the transitions, as these are already considered in previous functions.

```
def __str__(self):
    # You may add further information about the problem state (if necessary).
    string =  "Mazeworld Problem: " + "Start State: " + str(self.start_state) + " -- " \
        + "Goal State: " + str(self.goal_locations)
    return string
```

Similarly, the `animate_path()` method was already written, but is simply responsible for displaying the result/path of the search algorithm for this particular problem.

```
# Given a sequence of states, (including robot turn), modify the maze and print it out.
    # (Be careful, this does modify the maze!)
def animate_path(self, path):
    # Reset the robot locations in the maze.
    self.maze.robotloc = tuple(self.start_state[1:])

    # Cycle through each state within the path.
    for state in path:
        # Print the problem set-up.
        print(str(self))

        # Update the locations of the robot.
        self.maze.robotloc = tuple(state[1:])
        sleep(0.2)

        # Print the updated maze with the robot locations.
        print(str(self.maze))
```

Finally, the Manhattan heuristic provides the heuristic function used for the A* search, which is described later in the report.

```
# Calculate the Manhattan heuristic for the Mazeworld problem.
def manhattan_heuristic(self, state):
```

```
        total_distance = 0

        for i in range(1, len(state), 2):
            # Determine the position of the robot and it's associated goal position.
            robot_position = (state[i], state[i + 1])
            goal_position = (self.goal_locations[i - 1], self.goal_locations[i])

            # Add the Manhattan distance for a given robot to the total distance.
            total_distance += abs(robot_position[0] - goal_position[0]) + abs(robot_position[

        return total_distance
```

Questions

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?*

If there are $ k $ robots, we may represent the state of the system using a tuple $ (k, x_1, y_1, x_2, y_2, \ldots, x_k, y_k) $. That is to reconstruct the locations of all the robots, if we somehow forgot where all of them were, we need to have $ k $ tuples of size $ 2 $ representing the x,y-coordinates of each robot. Further, we need to know additional information to determine exactly what actions are available from this state, specifically the next robot to move, which is given as the first element of the tuple ($ k $ is used as an index here).

Thus, a tuple of size $ 2k + 1 $ is needed to fully represent the state of the system, allowing us to determine exactly what actions are available from the state. The width/height (dimensions) of the maze (alongside the goal locations of the robots) represents additional information that is not part of the state, but reflects the problem set-up.

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

The upper bound on the number of states in the system is given in terms of $ n $ and $ k $ as $ k \left(n^2 \times \left(n^2 - 1 \right) \times \left(n^2 - 2 \right) \times \cdots \times \left(n^2 - (k - 1) \right) \right) $. This is equivalent to $ k \left(\frac{\left(n^2 \right)!}{\left(n^2 - k \right)!} \right) $.

That is, if there are $ k $ robots and the maze is $ n \times n $, the first robot may choose from $ n^2 $ locations in the maze, the second may choose from $ n^2 - 1 $, and so on. This gives the upper limit on the number of states in the system considering that the maze does not contain any walls. We recall that the state includes information on which robot is next to move, so we introduce the factor of $ 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 $.*

An estimate on how many of these states represent collisions if the number of wall squares is $ w $ and $ n $ is much larger than $ k $ is given as follows: $ k \left(n^2 \times \left(n^2 - 1 \right) \times \left(n^2 - 2 \right) \times \cdots \times \left(n^2 - (k - 1) \right) \right) - k \left(n^2 - w \times \left(n^2 - 1 - w \right) \times \left(n^2 - 2 - w\right) \times \cdots \times \left(n^2 - (k - 1) - w \right) \right) $. This is equivalent to $ k \left(\frac{\left(n^2 \right)!}{\left(n^2 - k \right)!} \right) - k \left(\frac{\left(n^2 - w \right)!}{\left(n^2 - k - w \right)!} \right) $.

That is, if there are $ w $ wall squares, the number of possible (legal) states/locations for the robots decrease by $ w $. Thus, the total number of states is $ k \left(n^2 \times \left(n^2 - 1 \right) \times \left(n^2 - 2 \right) \times \cdots \times \left(n^2 - (k - 1) \right) \right) = k \left(\frac{\left(n^2 \right)!}{\left(n^2 - k \right)!} \right) $ and the total number of valid states is $ k \left(n^2 - w \times \left(n^2 - 1 - w \right) \times \left(n^2 - 2 - w\right) \times \cdots \times \left(n^2 - (k - 1) - w \right) \right) = k \left(\frac{\left(n^2 - w \right)!}{\left(n^2 - k - w \right)!} \right) $

The difference represents the total number of states with a collison. Further, we may consider a different approximation method, by which we take the percentage of walls in the maze and multiply that by the total number of states. This yields a result of $ \frac{w}{n^2} k \left(n^2 \times \left(n^2 - 1 \right) \times \left(n^2 - 2 \right) \times \cdots \times \left(n^2 - (k - 1) \right) \right) $, which is equivalent to $ \frac{w}{n^2} k \left(\frac{\left(n^2 \right)!}{\left(n^2 - k \right)!} \right) $.

4. *If there are not many walls, $ n $ is large (say $ 100 |times 100 $), 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?*

If there are not many walls and $ n $ is large (say $ 100 \times 100 $) with several robots (say $ 10 $), a straightforward breadth-first search on the state space will not be computationally feasible for all start and goal pairs. This is a result of the method of BFS, which in the worst case explores all nodes within a graph/tree.

In essence, each robot would explore $ 4 $ nodes (locations on the maze) at a time, adding these points to the explored set (thus adding them to memory). The BFS algorithm would thus produce a signficantly high memory cost, as it stores each of the nodes it visits (as an uninformed search strategy). Further, considering that there are $ 10 $ robots exploring a maze of size $ n \times n $ where $ n $ is large, the number of state spaces is exponentially high.

As indicated, breadth-first search tends to take longer than A* as it does not use the heuristic in the way that A* does. To improve upon BFS, A* considers a heuristic to determine the approximate path cost to from any given node to the goal.

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.*

The Manhattan heuristic is a useful, monotonic heuristic function for the search space, representing the minimum number of "moves" from a location to reach the goal state. The Manhattan distance calculates the distance between the robot's coordinates and the goal coordinates with the assumption that the robot is unable to move along diagonal spaces. In this sense, the Manhattan heuristic never overestimates the actual cost to the goal, and thus is consistent, providing a minimum path to the goal location/state (when used with A*).

As indicated, this heuristic represents the most optimal path to the goal without overestimating the cost, which is a feature that contributes to it being monotonic. By definition, a heuristic is monotonic when it is less than the actual cost of the path to the goal, and when it is less than the paths of the adjacent nodes to the goal plus the heuristic. The Manhattan heuristic is monotonic, as for all actions it is able to take, the path cost from the robot's location to it's neighbors is increased by $ 1 $. In the case where the neighbor is closer to the goal, the Manhattan distance is reduced by $ 1 $. In the case where the neighbor is further from the goal, the Manhattan distance is increased by $ 1 $. Thus, regardless of the location of the neighboring node, the path cost to the neighbor plus the neighbor's heuristic code will always be greater than or equal to the robot's current heuristic cost. (*Triangle Inequality*)

It may seem that you can just plan paths for the robots independently using three different breadth-first-searches, but this approach won't work very well if the robots get close to one another or have to move around one another. Therefore, you should plan paths in the complete state space for the system.

6. *Implement a model of the system and use A* search to find some paths. Test your program on mazes with between one and three robots, of sizes varying from $ 5 |times 5 $ to $ 40 |times 40 $ (You might want to randomly generate the $ 40 |times 40 $ mazes.)*

The following are a few neat examples, along with a description of why they are interesting. The output of the program (as ASCII text) is not given in this Markdown file, though using the testing file will demonstrate that reasonable paths are found by the algorithm.

`maze4.maz`

```
######################
##########.###########
#....................#
##########.###########
######################
\robot 1 2
\robot 2 2
\robot 3 2
```

Let us consider the case where the goal is `(20, 2, 19, 2, 18, 2)`. In this case, the robots are located in a corridor, in the "wrong" order, and must orient themselves differently (do something tricky) to reverse their order.

`maze5.maz`

```
####################
##########.##########
#..................#
####################
####################
\robot 1 2
\robot 2 2
\robot 3 2
\robot 4 2
```

Let us consider the case where the goal is `(20, 2, 19, 2, 18, 2, 17, 2)`. In this case, the robots are located in a corridor, in the "wrong" order, and must orient themselves differently (do something tricky) to reverse their order. This time, however, the solution is not so simple, as the robots are not able to tuck themselves away while letting the others pass. Instead, there must be a constant reordering, and increasing the number of robots makes the problem further computationally difficult.

`maze6.maz`

```
..........##########
##########.##########
##########.##########
#..................#
##########.##########
##########.##########
##########..........
\robot 0 6
\robot 1 6
\robot 2 6
```

Let us consider the case where the goal is `(21, 0, 20, 0, 19, 0)`. In this case, the robots are located within a more complex maze, and are again in the "wrong" order, and must reverse their order. This time, however, there are more options for reversing the order of the robots, and we consider that the robots must follow the vertical corridors to reach the final locations/state.

`maze7.maz`

```
.............................###############
.............................#..............
.............................#..............
.............................#..............
.............#...............#..............
.............#...............#..............
.............#.............................
.............#.............................
.............#.............................
###############.............................
\robot 0 1
\robot 0 9
```

Let us consider the case where the goal is `(46, 8, 46, 0)`. In this case, the robots are located within a large maze with relatively few walls, yet they start in different locations and come close to one another in the center as they navigate the maze. Thus, the robots should not "get in the way" of one another as they cross in the center while navigating the obstacles/walls.

`maze8.py`

```
#.................#.................#
.#...............#..#...............#.
..#.............#....#.............#..
...#...........#......#...........#...
....#.........#........#.........#....
.....#.......#..........#.......#.....
......#.....#............#.....#......
.......#...#..............#...#.......
........#..#................#..#........
.........#..................#.........
\robot 0 8
```

Let us consider the case where the goal is `(38, 9)`. In this case, the robot is located within a large maze with relatively few walls, yet there is an obvious path that is shortest/best to the goal. The robot follows closely along the wall, according to the heuristic, though it is not always the case that the robot is directly next to the wall. This is because the Manhattan heuristic weights locations equally (depending on their x,y (straight-line) distance to the goal). In this sense, "diagonals" are converted into straight lines, so there are some instances in which the robot will move to the right (or up/down) twice before changing course.

7. *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 in the book is a special case (subset) of this problem (the mazeworld), as it represents the case where there is a $ 3 \times 3 $ maze with no walls and $ 8 $ robots (that is, you fill the maze with robots except for a single space). That is, the robots (numbered tiles $ 1 $ through $ 8 $) are all looking to reach a final goal state, represented by a set of coordinate values.

Further, you may consider thinking about this case as the "blank" spot moving around the $ 3 \times 3 $ maze, though this does not entirely solve the problem, and may be harder to implement. Essentially, the $ 8 $-puzzle is a maze that is reconstructed around a moving "blank" space, where each part of the reconstructed maze must end up in the correct location, though this is not necessarily the best way to think about the problem.

In this case, the Manhattan heuristic function is still effective for the $ 8 $-puzzle, as it represents a consistent heuristic, though it is certainly complicated by the fact that robots are blocking the path of others.

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

The state space of the $ 8 $-puzzle is made of two disjoint sets, specifically solveable and unsolveable puzzles. In this sense, the disjoint sets represent two separate (connected) graphs with the states of each. The first set/graph is connected to the goal, while the other is not, thus there are at least two disjoint sets/graphs. In using the A* search algorithm, states will either lead to a proper solution, or they will lead to no solution at all.

To prove this, the aim would be to modify the program to find a starting state that does not yield a proper goal solution using A* search. This would be considered a secondary "goal" state. Now, we could randomize the starting state, and show (using A* search) that the state falls into one of the two disjoint sets. That is, if there is a path to the primary "goal" state (given as the proper goal), then we know it is part of the first set. Otherwise, there must be a path to the secondary "goal" state, which we could verify using the A* algorithm on the $ 8 $-puzzle. In this case, we know that the state is part of the second set.

To clarify, if we have a starting state that is unable to reach the proper goal state in the $ 8 $-puzzle, then we know that every other starting state will fall into one of two categories: either there is a path from it to the goal state or there is not. In the case where there is not, there will be a path from the starting state to the other starting state.

The goal solution is not reachable from the start state, and vice versa. Alternatively, we may think about the number of "inversions" as determined by the number of times that an tile's value is greater than the value of the tile at the next index. By considering all states, we classify the number of "inversions" into even and odd, which ends up yielding the same result.

Sensorless Problem (Blind Robot, Pacman Physics)

Assume that you now only have a single robot in mazeworld, but there's a catch. The robot is blind, and doesn't know where it starts! The robot does know the map of the maze.

The robot has a sensor that can tell it what direction is north (so that it can still move in intended directions). However, the robot has no other sensors. No, it really can't tell when it hits a wall.

Model & Implementation

If you execute the action "west" from a configuration where "west" is blocked, the robot simply doesn't move. In this problem, we will write a planning algorithm that gives a plan that even the blind robot can follow. Write an algorithm that causes the robot to go to the goal no matter where it starts. Specifically, let the 'state' in the search be the current set of possible locations of the robot. For example, in a $ 4 \times 3 $ maze, the first state in the search will be the set of all $ 12 $ possible starting locations: $ \left{ (0, 0), (1, 0), (2, 0), \ldots (3, 2) \right} $. The actions will be {north, south, east, west}. The goal test function tests if the state is a singleton, with just one element in the set. If there is just one element, then the robot knows where it is.

Sometimes people ask questions like "can the robot tell the difference between a wall and the edge of the maze?" No. The robot can't even tell that it hit something. All we know is that if you ask the robot to go west, it will either go west one square (if there is no wall there) or not (if there is a wall there). The robot doesn't know which happened. We know the shape of the maze, however, and this will allow us to set the search up so that the resulting sequence of robot commands is robust to either situation.

Write a model of the search problem in Python and use the model together with the A search algorithm you wrote previously to find motions of the robot. Animate the complete path, showing how the robot belief state changes as the path is executed. (By animate, I mean print out a sequence of text mazes with the robots in the proper locations.)*

Try a few different mazes ranging from small (6 x 6) to somewhat larger.

The Sensorless problem is implemented in `SensorlessProblem.py` (in Python). While the file contains quite a few comments, we will quickly review the structure here.

The constructor allows us to keep track of the start state and the goal state, according to the problem statement.

```
# Constructor
def __init__(self, maze):
    self.maze = maze
    self.start_state = [] # The various possible (valid) robot locations, contained withi

    # Initially, every single legal position is added.
```

```
        for x in range(self.maze.width):
            for y in range(self.maze.height):
                if self.maze.is_floor(x, y):
                    self.start_state.append((x, y))
```

The `get_successors` method is responsible for determining the appropriate states that follow, given an initial state.

The `for` statement considers all of the possible moves, and then determines if the state/position is safe (is not a wall), according to the constraints of the game. If so, we append the new value to the list. If not, we append the original value to the list. A set is used to ensure that multiple instances of the same value are not included, as we are representing a belief state.

Further, we introduce randomization of the moves to ensure that the algorithm functions for all possible combinations.

```
# Determine the successor states (as a list) for a given state.
    # Consider the valid moves from the current state and update the robot's position acc
def get_successors(self, state):
    # Initialize the list/set of successors, which starts out as empty.
        # A set is used to ensure that repeats are not allowed.
    successors_list = set([])

    # Enumerate the possible "moves" for the robot, according to the problem set-up.
    possible_moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]

    # Introduce randomness to allow for consecutive iterations to be independent of a det
    random.shuffle(possible_moves)

    # Cycle through the list of possible moves.
    for move in possible_moves:
        # Creating the next state.
        next_state = set([])

        # Cycle through the list of coordinates in the state.
        for coordinate_pair in state:
            # Unpack the coordinate pair and move values.
            x, y = coordinate_pair
            dx, dy = move

            # Create a new x and y coordinate pair based on the transition.
            new_x, new_y = x + dx, y + dy

            # If the position is not a wall or out of bounds...
            if self.maze.is_floor(new_x, new_y):
                # Add the new position to the set of possible states.
                next_state.add((new_x, new_y))
```

```
            else:
                # Otherwise, do not update the coordinate pair.
                    # Add the same position to the set of possible states.
                next_state.add(coordinate_pair)

        # Add all the possible robot locations given an action.
        successors_list.add(tuple(next_state))

    return tuple(successors_list)
```

Further, there is a method used to check if the current state is the goal state, which returns a boolean value.

```
# Test if the state is at the goal.
def is_goal_state(self, state):
    # If the state space is singular, then the robot has been located.
    return len(state) == 1
```

The to_str statement was already written, but it is simply responsible for displaying what the problem is, along with the start state. This fully defines the problem, given the problem set-up. We know the goal state, and we do not need to consider the transitions, as these are already considered in previous functions.

```
def __str__(self):
    # You may add further information about the problem state (if necessary).
    string = "Blind Robot Problem: " + "Start State: " + str(self.start_state)
    return string
```

Similarly, the `animate_path()` method was already written, but is simply responsible for displaying the result/path of the search algorithm for this particular problem.

```
# Given a sequence of states, (including robot turn), modify the maze and print it out.
    # (Be careful, this does modify the maze!)
def animate_path(self, path):
    # Reset the robot locations in the maze.
    self.maze.robotloc = tuple(item for sublist in self.start_state for item in sublist)

    # Cycle through each state within the path.
    for state in path:
        # Print the problem set-up.
        print(str(self))

        # Update the locations of the robot.
        self.maze.robotloc = tuple(item for sublist in state for item in sublist)
```

```
            sleep(0.2)

            # Print the updated maze with the robot locations.
            print(str(self.maze))
```

Finally, the sensorless heuristic provides the heuristic function used for the A* search, which is described later in the report.

```
# Calculate the sensorless heuristic for the sensorless problem.
def sensorless_heuristic(self, state):
    return len(state) - 1 # math.log2(len(state))
```

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

Hint: How does the successor function work? Take an example. Assume the current state is $ \left{ (3, 0), (3, 1), (3, 2) \right} $, which means the robot knows that it has $ x $ coordinate $ 3 $, but isn't sure about the $ y $ coordinate. Consider the action "north". If the location was initially $ (3, 0) $, then the location is $ (3, 1) $. If the location was initially $ (3, 1) $, then the new location is $ (3, 2) $. What if there is a wall north of $ (3, 2) $? If the old location was $ (3, 2) $, then the new location is $ (3, 2) $, since the robot bumps into the north wall. So the new state is the set $ \left{ (3, 1), (3, 2) \right} $. In other words, the robot is now more sure of its location, even though it has no sensors.

As indicated by the code above, the heuristic for the A* search for the sensorless problem is the length of the state (minus $ 1 $). This heurisitic is optimistic, as the value of the heuristic will never increase for a "move" by the robot in an attempt to determine its state within the maze.

When a robot is unsure of it's location (highlighted by the fact that the length of the state is greater than $ 1 $), the worst case scenario is when it must travel across the entire maze to localize. As the state size reduces, the belief state changes (reflecting that the robot is more confident in it's position, as the number of positions decreases). This indicatese that the heuristic decreases when the belief state size decreases, which is simply represented by the state.

The heurisitic is admissible (as it alwasy underestimates the number of moves required to localize) and is optimistic (as it predicts the cost of localization to be less than or equal to the actual cost).

As a comparison, the null heuristic implements a uniform-cost search, which explores more nodes yet typically still returns a valid result. This is at the cost of memory and time, though when using uniform-cost search, we can ensure that we have considered the problem in greater depth.

Instead of creating screen captures of the generated animation window to show that the algorithm generates effective plans, please run the test code to see the output. In this text, plans are described as a sequence of actions (i.e. "north, west, north, east, west, south, south"), though within the code, they are described using locations/transitions.

Bonus: Polynomial-Time Blind Robot Planning

This next part is a worthy challenge. According to the instructions, attempts are welcome, but you can still score full points for the assignment without answering this question correctly.

The following considering is designed to enhance the problem, thus meriting "extra credit" as defined by the outlines of the assignment. By going above and beyond to consider this problem, we consider a greater level of complexity for the problem.

Prove that a sensorless plan of the type you found above always exists, as long as the size of the maze is finite and the goal is in the same connected component of the maze as the start.

As indicated, the sensorless plan of the type found above performs an A* search where the heuristic is the number of possible locations that the robot could be in. If the goal is not in the same connected component of the maze as the start, this algorithm will not return a result, as there is no path from the start to the goal. Similarly, if the size of the maze is infinite, the goal remains uncertain and require an infinite number of steps from the start to reach the goal, which the A* search algorithm will be unable to accomplish due to time/memory constraints.

However, a sensorless plan of the type described always exists in the case where the size of the maze is finite and the goal is in the same connected component of the maze as the start. In other words, a localizing plan exists, according to the following approach:

1. If we find a path from every possible start location to the goal location using breadth-first search, there must be at least one such path. In practice, this is not done due to time/memory constraints, though we are able to confirm the existence of such a solution using the idea behind BFS.

2. Next, we add all of these paths together into a single long path $ P $ and follow it, keeping track of the set $ S $ of possible current locations. As in the problem, whenever $ S $ is of length $ 1 $ (a singleton), the robot knows exactly where it is and can stop.

3. This plan will eventually localize the robot, considering that following $ P $ will explore the entire reachable space (within the connected component of the graph). The BFS algorithm at worst goes to every node.

In other words, BFS ensures the exsistence of a solution, while A* gives the shortest, considering that the heurisitic is optimistic/admissible. In the case where the size of the maze is finite and the goal is in the same connected component of the maze as the start, a sensorless plan always exists. $ \square $

Now describe (but do not implement) a motion planner than runs in time that is linear or polynomial in the number of cells in the maze. (Notice that the size of the belief space is exponential in the number of cells, so this would be a very interesting result!)

Hint: As the robot moves, the number of robot locations in the belief state tends to decrease. given a map of a maze, can you always find a plan (or a sequence of plans) that decreases the number of robot locations by one?

There exists a motion planner than runs in time that is linear or polynomial in the number of cells in the maze. What follows is a description of such a motion planner. As indicated, such a plan would depend on the particular map, so an algorithm mgiht have to be run in order to generate such a plan.

In a broad-level sense, a "flooding" technique may be used to reduce the belief state of the robot, with the key insight being that we can find a sequence of actiosn that will merge two neighboring belief states into one, for *every* single state. By iteratively doign this, we are able to reduce *any* intial state (representing the belief state) to a single position. The number of "flood/merge" actions needed is polynomial in terms of the number of cells in the maze.A detailed explanation is as follows.

Claim: There exists a plan to reduce *any* initial blind robot belief state to a single position using a polynomial number of actions relative to the number of cells in the maze. *Aside*: This may not produce the fastest path.

Proof: Given any maze, we are able to pre-process it in polynomial time, in order to identify all adjacent cell pairs that share a border, that is, walls are not included. We claaim that for any two belief states $ A $ and $ B $ that contain adjacent cells, there exists a short sequence of actions (at most $ 4 $) moves that merges $ A $ and $ B $ into a single new belief state $ C $.

For example, if $ \left{ (1, 1) \right} \in A $ and $ \left{ (1,2) \right} \in B $, the sequence $ \left['east', 'west' \right] $ will merge them into $ C $ regardless of wall layout.

The idea is that any initial belief state with $ n $ possible positions is able to sequentially be merged down to a single position by greedily selecting adjacent belief "pairings", requiring at most $ n - 1 $ "flood/merge" plans. The algorithm is as follows:

1. Preprocess the maze to find all the adjacent cell pairs with an open border.
2. Initialize the belief state (starting state) to all possible positions.
3. While the belief state is of length greater than $ 1 $, greedily select two "adjacent" belief states $ A $ and $ B $ and determine the appropriate merge plan using preprocessed pairs ('east'/'west' or 'north'/'south').

This algorithm runs in polynomial time relative to the maze size. The main idea is that we are able to "stitch" belief states together through "local" floods/merges, regardless of the layout of the maze.

Further, this is possible because of the finite nature of the maze, along with the placement of the walls. If the maze were infinite, this would not be possible. Essentially, we are able to continuously reduce the belief state by $ 1 $ each time for a maze with $ n \times n $ cells. This is done by carefully selecting the appropriate action to take, which is determined via preprocessing of the maze.

By considering that a robot is not able to move beyond the bounds of the maze, and that it does not move when it hits a wall, we certainly guarantee that a movement (in any direction) will reduce the state space (which is representative of the belief state). This "merging" is only possible knowing that the entire global maze does not shift, that is, the boundary/wall conditions ensure that if we were to drop a robot at every valid location, at least some would stay in the same space, thus reducing the state space in an iterative fashion.

Literature Review - Bonus (Undergraduate)

This part of the assignment is required for graduate students in the class. Undergraduate students are very welcome to do this part (I think it quite interesting), and will receive some additional credit. (Please note in this section of the report whether you are a graduate student or not.)

The multi-robot problem from this assignment is a well-known one, and it has been studied extensively in the AI research community. Professor Kostas Bekris at Rutgers who has done some work in this area, recommended the following five papers.

Graduate students in the course should pick at least one of these papers, and read enough of it to get a sense of the approach. Briefly discuss the paper in your report. (3-5 sentences will be sufficient for this discussion.) The discussion should describe the problem attacked by the paper, give a quick summary of the main result(s), and discuss the basic approach of the paper.

Paper: Auletta, V.; Monti, A.; Persiano, P.; Parente, M.; and Parente, M. 1999. A Linear Time Algorithm for the Feasibility of Pebble Motion on Trees. Algorithmica 23(3):223–245

This paper studies the computational complexity of determining if a sequence of pebble moves on a tree graph is feasible. The pebble motion problem is set up as follows: there is a graph with pebbles placed on vertices and a pebble may be moves to a neighboring (unoccupied) vertex, according to given rules.

The main result of the paper is a linear algorithm that determines if a given pebble motion is feasible. This is done by reducing it to a "reachability" problem on directed acyclic graphs (DAGs). There is a correspondence between pebble motions and vertex orderings (interesting graph theory!), which allows them to construct a directed acyclic graph that encodes the constraints of the problem. As

indicated, by testing reachability, the researchers are able to determine if the pebble motion sequence is valid.

A key idea is using depth-first search of a graph to assign numbers to vertices that refelct the pebble motion constraints. This numbering technique for the vertices is used to build the directed acyclic graph. The linear-time algorithm gives an efficient method for determining the answer to the problem.

Paper: Goraly, G., and Hassin, R. 2010. Multi-color Pebble Motion on Graphs. Algorithmica 58(3):610–636.

This paper studies the multi-color pebble motion problem, where pebbles of different colors are placed on a graph and are moved according to color-specific rules. In this sense, it is quite in line with the previous paper.

The main result is a polynomial-time algorithm that determines if a multi-color pebble motion sequence is feasible (on a graph). The key idea involves constructing a directed graph that encodes the constraints (as before), such that the motion of the pebbles if feasible if and only if there are no cycles in the constraint graph (the graph is acyclic.)

Aside: In taking Math 038 (Graph Theory) here at Dartmouth College, we did not cover this, though it seems like an interesting topic! This indicates the significance of graph theory in the context of computer science and AI, as there a variety of problems that may be constructed as graph theory problems, which we have various theorems/propositions/proofs for.