CS520 - Introduction to Artificial Intelligence

Harshankar Reddy Gudur (hsg64)

1

Problem 1

1.1 The Problem

For a fixed, known ship layout as in Project 1, consider the following: you've beaten the aliens and driven them off the ship. You fought bravely, but your sensors were damaged. You cannot tell where you are in the ship. You can tell that your motors are functional enough to move you, but you can't even tell if you're just bumping into a wall when you try to move.

Write an algorithm to take steps and identify where you are in the ship. Do so as efficiently as possible. Be clear about your approach, what you are doing to try to be efficient and intelligent about it, and connect to as many topics from class as you can. (The bot has the map of the grid)

1.2 Initial Solution; Solution 1

Using the same grid as Project 1 with 'D' set to 30 (30x30), my initial idea was to, well, try to go to a corner of the grid. One thing that slipped my mind was the fact that the grid itself is fixed and not variable.

```
    class GridAttrib:
    __slots__ = ('open', 'bot_occupied', 'traversed')

    def __init__(self):
        self.open = False
        self.bot_occupied = True
        self.traversed = False
```

```
    def gen_grid_iterate(self):
        cells_to_open = []
        # Get all blocked cells with one open neighbors
        for j in range(self.D):
            for i in range(self.D):
                if self.grid[j][i].open == True:
                    continue
                neighbors_ind = self.get_neighbors((i, j))
                open_neighbors = []
                for neighbor_ind in neighbors_ind:
                    if self.grid[neighbor_ind[1]][neighbor_ind[0]].
open is True:
                        open_neighbors.append(neighbor_ind)
```

```
                if len(open_neighbors) == 1:
                    cells_to_open.append((i, j))
        # Randomly open one of those cells
        if len(cells_to_open) > 0:
            index = random.choice(cells_to_open)
            self.grid[index[1]][index[0]].open = True
        if self.debug:
            print("After one iteration")
            print(self)
            print(f"Cells to open: {len(cells_to_open)}")
        return len(cells_to_open) != 0

    # Grid generation happens here
    def gen_grid(self):
        for j in range(self.D):
            row = []
            for i in range(self.D):
                row.append(GridAttrib())
            self.grid.append(row)

        # Open Random Cell
        rand_ind = np.random.randint(0, self.D, 2)
        self.grid[rand_ind[1]][rand_ind[0]].open = True
        if self.debug:
            print(self)
        # Iterate on the grid
        while self.gen_grid_iterate():
            pass
        # Randomly open half the dead ends
        cells_to_open = []
        for j in range(self.D):
            for i in range(self.D):
                    all_neighbors = self.get_neighbors((i,j))
                    open_neighbors = [ind for ind in all_neighbors
if self.grid[ind[1]][ind[0]].open]
                    closed_neighbors = [ind for ind in
all_neighbors if not self.grid[ind[1]][ind[0]].open]
                    # randint is used here to maintain a ~50%
chance of any dead end opening
                    if self.grid[j][i].open and random.randint(0,
1) == 1 and len(open_neighbors) == 1:
                        cells_to_open.append(random.choice(
closed_neighbors))
        for ind in cells_to_open:
            self.grid[ind[1]][ind[0]].open = True
        if self.debug:
            print("After dead end opening")
            print(self)
```

Using pickle to save the state of the grid.

```
    def save_grid_state(self, filename):
        file_path = os.ops.join("C:\\Users\\Haru\\Desktop\\",
filename)
        with open(file_path, 'wb') as file:
            pickle.dump(self.grid, file)

```

```
    @classmethod
    def load_grid_state(cls, filename):
        file_path = os.ops.join("C:\\Users\\Haru\\Desktop\\",
    filename)
        with open(file_path, 'rb') as file:
            grid = pickle.load(file)
        obj = cls()
        obj.grid = grid
        return obj
```

The way I approached this was to assign probabilities to the movements of the bot. (We get absolutely no feedback from movement, not even knowing if we bumped into a wall). I picked the top left corner of the grid to be the place I wanted to go to. Below is the initial code.

```
    def move(self):
        # if self.ops is None:
            # self.plan_path()
        # if len(self.ops) == 0:
            # if self.debug:
                # print("No ops found!")
            # return
        # open_neighbors = self.grid.get_open_neighbors(self.ind)

        neighbors = self.grid.get_neighbors(self.ind)
        next_move_prob = random.random()

        if next_move_prob <= 0.7:
            next_dest = random.choice([(self.ind[0], self.ind[1] -
    1), (self.ind[0] - 1, self.ind[1])])
        elif next_move_prob <= 0.85:
            next_dest = (self.ind[0] + 1, self.ind[1])
        else:
            next_dest = (self.ind[0], self.ind[1] + 1)

        if self.grid.valid_index(next_dest) and self.grid.grid[
    next_dest[1]][next_dest[0]].open:
            self.grid.remove_bot(self.ind)
            self.ind = next_dest
            self.grid.place_bot(self.ind)
            self.grid.set_traversed(self.ind)
```

I dispersed the probabilities of the bot movement; 70% chance for it to move either 'UP' or 'LEFT', and 15% chance each for it to move either 'DOWN' or 'RIGHT'. On a random grid configuration I generated and saved, I ran this program and it was surprising performing well. I reached the top left corner in well under a 130 steps most of the time, but again, the end result was for me to find the actual coordinates of the bot at the end of the run. There was no way for me to confidently say that at this timestep, the bot is exactly at (D-1,D-1) since at that particular point, it could've hit the 20% (10% up or 10% left) giving me the wrong coordinates. To this end, I essentially used an aging factor. I looped over the previous program a 1000 times, and it resulted in an average of around 120 steps for that particular

grid. Using this, I essentially reduce the probabilities of the 'DOWN' and 'RIGHT' movements when it reaches that threshold, to a point where I can confidently say that the bot is indeed in those coordinates. This was just the initial idea of getting the problem done, but there's much better ways to go about it. Here, when I changed the grid, and the grid blocked cell placements weren't favourable, the average number of steps increased over 3 fold. So the next idea I wanted to implement was to get that consistency across various grids, and a solution that is much, much more efficient than this rudimentary approach.

1.3 Solution 2

Here, instead of going about it the traditional way, the idea was to first simply assume each open cell in the grid is a '1', denoting that the bot could exist in any of those cells. I then proceed to do a series of move operations on each of these existing 'bots', either 'UP', 'DOWN', 'LEFT', or 'RIGHT'. At some point, we converge at one particular set of coordinates in the grid, and you can reach this regardless of where you spawn. Let's assume we do an 'UP' operation. In this case, if the value of that particular cell above the current cell is 1, we set the current cell to 0, denoting bot movement. If there is a blocked cell, the current cell will stay a '1'. Now, as we do a series of these move operations on the grid, ensuring we stay within the bounds of the indices, we will inevitably end up with just one '1' left, and that's the coordinates for the bot itself.

For example, take the random grid shown below. As mentioned, the 'UP' operation, essentially forces 'all the bots' (Since we assume each cell to be a bot) to move up a position. The 'RIGHT' operation moves them to the right, again, staying as 1 if there's a blocked cell, and making the cell it moves from a 0.

Figure 1.1: Initial Random Grid; After 'UP'; After 'RIGHT'

Similarly, the 'DOWN' and 'LEFT' operations are implemented. After a series of such movements, you'll be left with just one '1', which is the position the bot is in.

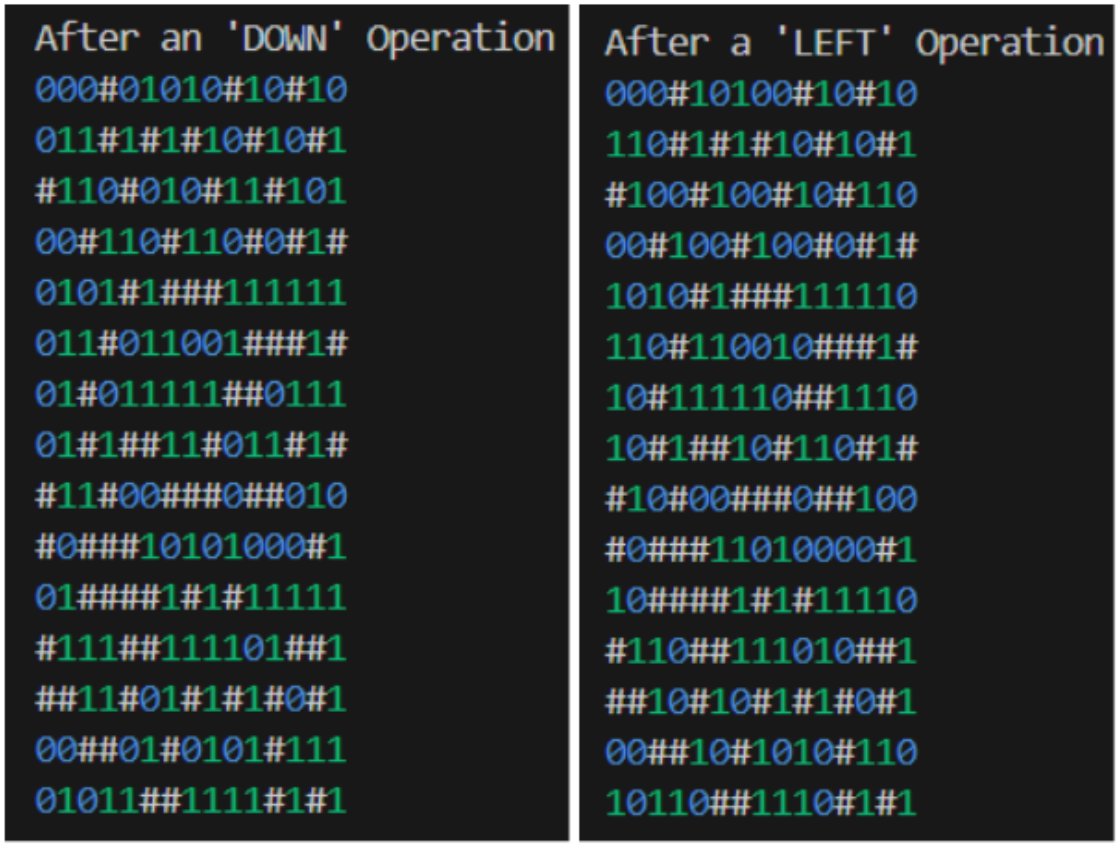

Figure 1.2: After 'DOWN'; After 'LEFT'

This piece of code is responsible for shifting 'all the bots' up a position.

```
    def shift_up(self):
        for j in range(self.D - 1):
            for i in range(self.D):
                if not (0 <= j + 1 < self.D):
                 continue
                if not self.grid[j + 1][i].bot_occupied or not self
    .grid[j][i].open:
                  continue
                self.grid[j][i].bot_occupied = self.grid[j + 1][i].
    bot_occupied
                self.grid[j + 1][i].bot_occupied = False
```

Similarly, this is how I shift the positions right, down, and left.

```
     def shift_right(self):
        for j in range(self.D):
            for i in range(self.D - 1):
                if not (0 <= i + 1 < self.D):
                    continue
```

```
                if not self.grid[j][i].bot_occupied or not self.grid[j][i + 1].open:
                    continue
                self.grid[j][i + 1].bot_occupied = self.grid[j][i].bot_occupied
                self.grid[j][i].bot_occupied = False

    def shift_down(self):
        for j in range(self.D):
            for i in range(self.D):
                if not (0 <= j - 1 < self.D):
                    continue
                if not self.grid[j - 1][i].bot_occupied or not self.grid[j][i].open:
                    continue
                self.grid[j][i].bot_occupied = self.grid[j - 1][i].bot_occupied
                self.grid[j - 1][i].bot_occupied = False

    def shift_left(self):
        for j in range(self.D):
            for i in range(self.D):
                if not (0 <= i - 1 < self.D):
                    continue
                if not self.grid[j][i].bot_occupied or not self.grid[j][i - 1].open:
                    continue
                # if self.grid[j][i].bot_occupied == False:
                #     continue
                self.grid[j][i - 1].bot_occupied = self.grid[j][i].bot_occupied
                self.grid[j][i].bot_occupied = False
```

Through the priority queue, I retrieve the next item, which constitutes the number of zeroes, their coordinates, and the sequence of actions (moving up right down or left) to reach that particular state. I then reset the grid and iterate over each of the actions, getting the new number of zeroes after the action and their coordinates, and append it to the visited list in order to not repeat that configuration. I then check if there's one '1' left in the grid, which would mean that I've found my solution, and I break out of the loop if that's the case.

```
while not pQ.empty():
    (z, prev_coor, path) = pQ.get()
    g.reset_grid2(prev_coor)

    g.move_left()
    new_coor = g.get_zeroes()
    new_zeroes = len(new_coor)

    if new_coor != prev_coor and new_coor not in visited:
        visited.append(new_coor)
        pQ.put((-new_zeroes, new_coor, path + [0]))

    if abs(new_zeroes) == open_cells - 1:
```

```
        final_coor.append((path, new_coor))
        break
    g.reset_grid2(prev_coor)

    g.move_right()
    new_coor = g.get_zeroes()
    new_zeroes = len(new_coor)

    if new_coor != prev_coor and new_coor not in visited:
        visited.append(new_coor)
        pQ.put((-new_zeroes, new_coor, path + [1]))

    if abs(new_zeroes) == open_cells - 1:
        final_coor.append((path, new_coor))
        break
    g.reset_grid2(prev_coor)

    g.move_up()
    new_coor = g.get_zeroes()
    new_zeroes = len(new_coor)

    if new_coor != prev_coor and new_coor not in visited:
        visited.append(new_coor)
        pQ.put((-new_zeroes, new_coor, path + [2]))

    if abs(new_zeroes) == open_cells - 1:
        final_coor.append((path, new_coor))
        break
    g.reset_grid2(prev_coor)

    g.move_down()
    new_coor = g.get_zeroes()
    new_zeroes = len(new_coor)

    if new_coor != prev_coor and new_coor not in visited:
        visited.append(new_coor)
        pQ.put((-new_zeroes, new_coor, path + [3]))

    if abs(new_zeroes) == open_cells - 1:
        final_coor.append((path, new_coor))
        break
    g.reset_grid2(prev_coor)
```

Figure 1.3: Example of the output of a random grid

This particular solution achieves exactly what the previous one couldn't, consistency and efficiency. The best part about this solution is how the grid underneath does not matter whatsoever, and that you can also confidently tell exactly what the coordinates of the bot are. This particular implementation does take a lot of time to compute for very high sizes of D though, which is one thing I would've liked to optimize more, but it is still pretty fast to execute.