Assignment 2: Inference-Informed Action

Karun Kanda (Section 7), Agam Modasiya (Section 3), and Varun Gaddam (Section 3)

March 22nd, 2021

1 Usage

To run the basic agent:

```
python Run_Agent.py <dimension of board> <mine density> ba
```

To run the advance agent:

```
python Run_Agent.py <dimension of board> <mine density> aa
```

To run the extra credit:

```
python Run_Agent.py <dimension of board> <mine density> ec
```

2 Language and Libraries Used

For this project we used Python and the Python libraries we used were:

1. numpy - To create nice nested arrays to represent the board and the board used by the agent.
2. random - To obtain random points for the given dimension size.
3. sys - To get command line arguments to dynamically create the game boards.
4. Collections (defaultDict, deque)- DefaultDict was used to create a mine dictionary, while deque was used to establish a queue.

3 Questions about Project 2: (Analysis)

3.1 Representation: How did you represent the board in your program, and how did you represent the information/knowledge that clue cells reveal? How could you represent inferred relationships between cells?

For our program we represented the game board using a 2 dimensional matrix. We used blank spaces, numbers and the character 'm' to represent empty spaces, clues and mines respectively. In the build board algorithm (the code below) we randomized the location of the mines like in Minesweeper and added the clues based off the cell's neighbors like in Minesweeper.

```
# using the information from the initilization step proceed to build the board with the mines
    information and the agent's board
def build_board(self):
    # to track the mines for the board
    mines = self.mines
    # dim array is utilized to pick a random index value for the game board
    dim_array = list(range(0, self.dim))
    # iterate through the board and choose randomly distinct points to add the mines for a more
        randomized board
    while mines != 0:
        i = random.choice(dim_array) # randomly choose a i value (rows)
        j = random.choice(dim_array) # randomly choose a j value (columns)
        arr = [0, 1] # these will represent random choices
        if random.choice(arr) == 0 and mines != 0 and self.board[i][j] == 0:
            # if the condition is met the program should put a mine at the (i, j) spot and decrement
                 the number of mines needed to be placed
            self.board[i][j] = 'm'
            mines -= 1

    # now for the rest of the board add empty spots and clues based on the positions of the mines
        on the board
    # the clue value in a certain cell ranges based on the neighbors that have a mine so that can
        be from 1-8 based on the density
    for i in range(0, self.dim):
        for j in range(0, self.dim):
            count = 0 # to keep track of the clue value that should be placed
            self.bot_board[i][j] = ' '
            if self.board[i][j] == 0: # initially the board is set to all 0s
                # check the up direction for a mine
                if self.check_constraints(i - 1, j) and self.board[i - 1][j] == 'm':
                    count += 1
                # check the down direction for a mine
                if self.check_constraints(i + 1, j) and self.board[i + 1][j] == 'm':
                    count += 1
                # check the left direction for a mine
                if self.check_constraints(i, j - 1) and self.board[i][j - 1] == 'm':
                    count += 1
                # check the right direction for a mine
                if self.check_constraints(i, j + 1) and self.board[i][j + 1] == 'm':
                    count += 1
                # check the upper left region for a mine
                if self.check_constraints(i - 1, j - 1) and self.board[i - 1][j - 1] == 'm':
                    count += 1
                # check the upper right region for a mine
                if self.check_constraints(i - 1, j + 1) and self.board[i - 1][j + 1] == 'm':
                    count += 1
                # check the bottom left region for a mine
                if self.check_constraints(i + 1, j - 1) and self.board[i + 1][j - 1] == 'm':
                    count += 1
                # check the bottom right region for a mine
                if self.check_constraints(i + 1, j + 1) and self.board[i + 1][j + 1] == 'm':
                    count += 1

                # if the count is 0 don't write a hint otherwise write the hint
                if count == 0:
                    self.board[i][j] = ' '
                else:
                    self.board[i][j] = str(count)
```

Then upon executing the build board algorithm here is how a 10 X 10 board looks like printed to the console with the mines, clues and empty spaces present:

```
[[’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’]
 [’1’ ’1’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’]
 [’m’ ’1’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’1’ ’1’ ’1’]
 [’2’ ’2’ ’1’ ’ ’ ’ ’ ’ ’ ’ ’ ’1’ ’m’ ’1’]
 [’1’ ’m’ ’1’ ’ ’ ’ ’ ’ ’ ’ ’ ’1’ ’1’ ’1’]
 [’3’ ’3’ ’3’ ’1’ ’1’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’]
 [’m’ ’m’ ’2’ ’m’ ’1’ ’1’ ’1’ ’1’ ’ ’ ’ ’]
 [’3’ ’3’ ’3’ ’1’ ’1’ ’1’ ’m’ ’1’ ’1’ ’1’]
 [’1’ ’m’ ’2’ ’1’ ’1’ ’1’ ’1’ ’1’ ’1’ ’m’]
 [’1’ ’1’ ’2’ ’m’ ’1’ ’ ’ ’ ’ ’ ’ ’1’ ’1’]]
```

We also had an agent board that represented the cells the agent chose at a time because the actual locations of the mines, clues, empty spaces isn't known prior. Prior to running the program, the agent board was set to all blank to represent an unexplored board. Every time the agent discovered a safe cell or was able to deduce that a particular cell was a mine, the information was represented on the agent board like what we see below.

```
self.bot_board[x][y] = value
```

We represented the knowledge that the clues revealed using a dictionary. The dictionary stored information pertaining to each cell, such as whether a particular cell was open and whether that particular cell was a mine. Which we had a set to indicate if the cell is open and if the cell is a mine in this type of representation: {"open": <TF>, "mine": <TF>}

```
# Initialize the dictionary to keep track of what's opened and at which cells there is a mine
def dictionary_init(self):
    for i in range(0, self.dim):
        for j in range(0, self.dim):
            # sample of how the dictionary is represented
            self.mine_dict[i][j] = {"open": False, "mine": False}
```

When the cell is believed to be a mine its updated in the mine dictionary like this:

```
self.mine_dict[x][y] = {"open": True, "mine": True} #Updating the dictionary when a mine is found
```

When the cell is believed to be not a mine its updated in the mine dictionary like this:

```
self.mine_dict[x][y] = {"open": True, "mine": False} #Updating the dictionary when a safe cell is
    found
```

We represented inferred relationships between cells by formulating equations using clue values among cells that were in the same general vicinity. When our agent lands on a numbered cell (has a clue value), it then forms an equation for that cell. The equation consisted of the neighboring cells that were not discovered and the clue value for that particular cell. We then used equations for neighboring cells to determine which cells in that particular area were mines and which cells were safe to open, if any. The function to build equation for a single cell is shown below which we used the information in the neighbors to build its respective equation.

```python
# A function to build an equation given a cell location by looking at how many of it's neighbors
    are not opened
def build_eq(self, one):
    x = one[0] # x corresponds to the first element in the one array
    y = one[1] # y corresponds to the second element in the array
    z = [] # to assist in building the equation at the end
    # to account for any solved mines in the neighbors
    value = int(self.bot_board[x][y]) - self.mine_updater(x, y)
    # down direction
    if self.check_constraints(x + 1, y) and self.mine_dict[x + 1][y]["open"] is False:
        z.append((x + 1, y))
    # up direction
    if self.check_constraints(x - 1, y) and self.mine_dict[x - 1][y]["open"] is False:
        z.append((x - 1, y))
    # right direction
    if self.check_constraints(x, y + 1) and self.mine_dict[x][y + 1]["open"] is False:
        z.append((x, y + 1))
    # left direction
    if self.check_constraints(x, y - 1) and self.mine_dict[x][y - 1]["open"] is False:
        z.append((x, y - 1))
    # bottom right direction
    if self.check_constraints(x + 1, y + 1) and self.mine_dict[x + 1][y + 1]["open"] is False:
        z.append((x + 1, y + 1))
    # bottom left direction
    if self.check_constraints(x + 1, y - 1) and self.mine_dict[x + 1][y - 1]["open"] is False:
        z.append((x + 1, y - 1))
    # upper left direction
    if self.check_constraints(x - 1, y - 1) and self.mine_dict[x - 1][y - 1]["open"] is False:
        z.append((x - 1, y - 1))
    # upper right direction
    if self.check_constraints(x - 1, y + 1) and self.mine_dict[x - 1][y + 1]["open"] is False:
        z.append((x - 1, y + 1))

    # return in the set format (((x1,y1), (x2, y2), (x3, y3)), 1) where 1 is the clue value
    return {value, tuple(z)}
```

3.2 Inference: When you collect a new clue, how do you model / process / compute the information you gain from it? In other words, how do you update your current state of knowledge based on that clue? Does your program deduce everything it can from a given clue before continuing? If so, how can you be sure of this, and if not, how could you consider improving it?

Once we open a cell it can either be a mine or a clue value. As for the clue value, it can be 0, meaning none of it's neighbors have a mine, or a number greater than 0, meaning there is at least one mine in it's neighboring cell.

Mine: If the agent opens a cell that is mine, it is added to the knowledge base.

```
if value == 'm': # if it's a mine we update the knowledge base accordingly
    self.bot_board[x][y] = 'm'
    self.mine_dict[x][y] = {"open": True, "mine": True}
    self.visited += 1
    # remove the cell from the possible coordinates
    self.possible_coordinates.remove((x, y))
    c += 1
```

Clue Value of 0: It is added to the knowledge base and we look through all it's neighbors. If the neighbors have not been opened before, we add them to a stack named 'q'. Then we go through the stack of neighbors one by one until it's empty. If the neighbor has a clue value of 0, we add that neighbor to the 'q' stack, given it was now opened before. If the neighbor has a clue value that is greater than 0, we add it to a queue named 'numbers'. This allows us to open all the safe cells around a cell that has a clue value of 0, which builds border of cells that have a clue value of greater than 0. The main process goes like this where we first go through the updater where the updater given a specific x, y will update that cell's information.

```
elif value == ' ': # if the cell is empty, meaning the clue value is 0, we open the cell around it
     given they haven't been opened before
    # if the cell hasn't been opened before we update the knowledge base
    if self.mine_dict[x][y]["open"] is False and (x, y) not in q:
        self.bot_board[x][y] = value
        self.mine_dict[x][y] = {"open": True, "mine": False}
        self.possible_coordinates.remove((x, y))
        self.visited += 1

    # check all the directions because if its ' ' that would entail the rest of its neighbors are
        not mines
    if self.check_constraints(x + 1, y):
        if self.updater(x + 1, y):
            self.visited += 1

    if self.check_constraints(x - 1, y):
        if self.updater(x - 1, y):
            self.visited += 1

    if self.check_constraints(x, y + 1):
        if self.updater(x, y + 1):
            self.visited += 1

    if self.check_constraints(x, y - 1):
        if self.updater(x, y - 1):
            self.visited += 1
    if self.check_constraints(x + 1, y + 1):
        if self.updater(x + 1, y + 1):
            self.visited += 1

    if self.check_constraints(x + 1, y - 1):
        if self.updater(x + 1, y - 1):
            self.visited += 1

    if self.check_constraints(x - 1, y - 1):
        if self.updater(x - 1, y - 1):
            self.visited += 1

    if self.check_constraints(x - 1, y + 1):
        if self.updater(x - 1, y + 1):
            self.visited += 1
```

The updater method is below that takes a x, y argument and returns if it has been updated (True or False).

```
# updates the knowledge base for a given cell
def updater(self, x, y):
    if self.board[x][y] == ' ' and (x, y) not in q and self.mine_dict[x][y]["open"] is False:
        # if the value of cell is empty we add it to the q stack
        q.append((x, y))
        self.mine_dict[x][y] = {"open": True, "mine": False}
        self.bot_board[x][y] = self.board[x][y]
        self.possible_coordinates.remove((x, y))
        return True # return true only if there was new knowledge added

    elif (x, y) not in numbers and self.mine_dict[x][y]["open"] is False:
        # if the value of cell has a clue value > 0 we add it to the numbers queue
        numbers.append((x, y))
        self.mine_dict[x][y] = {"open": True, "mine": False}
        self.bot_board[x][y] = self.board[x][y]
        self.possible_coordinates.remove((x, y))
        return True # return true only if there was new knowledge added

    return False
```

Clue Value Greater than 0: It is added to the knowledge base and the queue named 'numbers'. We then go through the 'numbers' stack that takes 3 points at a time and tries to get a solution from the equation built from the cell locations. If there is a solution found it is updated in the knowledge base inside the solver function.

```
elif value.isnumeric(): # if the cell has value that's a clue > 0 we update the knowledge base
    if not self.mine_dict[x][y]["open"]: # if its not already open it should be added to the
        knowledge base and update the bot board accordingly
        self.mine_dict[x][y] = {"open": True, "mine": False}
        self.bot_board[x][y] = value
        self.possible_coordinates.remove((x, y))
        numbers.append((x, y))
        self.visited += 1
```

Lastly, we iterate cell by cell to find more solutions. We go through the first neighbors of the origin cell and then the secondary neighbors of the origin cell. These neighbor cells are added to the 'numbers' queue if they are already opened. Then similar to before, 3 points are sent to the solver function to find a solution until the queue is empty. We do this for every cell and if there are solutions found in the process we iterate through the maze again to use the new knowledge and this is how we know for sure that we have used the available knowledge as much as possible. If there are no more solutions to be found we select a random point to be opened. The agent then keeps playing until all the cells have been opened, either by the agent or marked as a mine by the solver.

```python
# takes 3 cell locations and tries to find a solution from them
# then finds the difference from the solution it found
def solver_3(self, p1, p2, p3):

    # create the 3 equations from the 3 points
    eq1 = self.build_eq(p1)
    eq2 = self.build_eq(p2)
    eq3 = self.build_eq(p3)

    # find solutions from the local knowledge of the cell
    self.solver_1(eq1)
    self.solver_1(eq2)
    self.solver_1(eq3)

    # difference of the given equations
    diff_1_2 = self.set_difference(self.build_eq(p1), self.build_eq(p2))
    diff_1_3 = self.set_difference(self.build_eq(p1), self.build_eq(p3))
    diff_2_3 = self.set_difference(self.build_eq(p2), self.build_eq(p3))

    if diff_1_2 is not None and diff_2_3 is not None: # find a solution from the differences
        self.solver_2(diff_1_2, diff_2_3)
    if diff_1_2 is not None and diff_1_3 is not None:
        self.solver_2(diff_1_2, diff_1_3)
    if diff_1_3 is not None and diff_2_3 is not None:
        self.solver_2(diff_1_3, diff_2_3)

    return
```

This solver function take 3 points and tries to find a solution from then creates an equation that can be formed from the 3 points. The equation builder function iterates through each neighbor of the cell and if a neighbor is not visited it is added to a list. As for the clue value, if one of the neighbors is a discovered as a mine, we subtract that from the clue value. The equation builder function then returns a set, with the list of undiscovered points as tuple and the clue value as an integer. To find the solutions from the functions we first try to solve each function individually by determining if the number of points in the set is equal to the clue value, meaning they are all mines. Or the clue value is 0, meaning all the points in the set are safe.

```python
# finds the difference between 2 given equations
def set_difference(self, eq1, eq2):

    if len(eq1) == 2 and len(eq2) == 2:

        eq1_neighbors = eq1.pop() # list of all the neighbors of cell 1
        eq2_neighbors = eq2.pop() # list of all the neighbors of cell 2
        num1 = eq1.pop()
        num2 = eq2.pop()
        if isinstance(num1, int) and isinstance(num2, int):
            set1 = {eq1_neighbors} # covert the list into set
            set2 = {eq2_neighbors}
            diff = set1.difference(set2) # difference between 2 sets
            num_diff = num1 - num2 # difference between the clue additions
            if any(diff):
                # return a new equation that is the difference of the 2 input equations
                return {diff.pop(), abs(num_diff)}
        elif isinstance(num1, int) and isinstance(eq1_neighbors, int):
            set1 = {num2} # covert the list into set
            set2 = {eq2_neighbors}
            diff = set1.difference(set2) # difference between 2 sets
            num_diff = num1 - eq1_neighbors # difference between the clue additions
            if any(diff):
                # return a new equation that is the difference of the 2 input equations
                return {diff.pop(), abs(num_diff)}
        elif isinstance(num1, int) and isinstance(eq2_neighbors, int):
            set1 = {eq1_neighbors} # covert the list into set
            set2 = {num2}
            diff = set1.difference(set2) # difference between 2 sets
            num_diff = num1 - eq2_neighbors # difference between the clue additions
            if any(diff):
                # return a new equation that is the difference of the 2 input equations
                return {diff.pop(), abs(num_diff)}
        elif isinstance(num2, int) and isinstance(eq2_neighbors, int):
            set1 = {eq1_neighbors} # covert the list into set
            set2 = {num1}
            diff = set1.difference(set2) # difference between 2 sets
            num_diff = num2 - eq2_neighbors # difference between the clue additions
            if any(diff):
                # return a new equation that is the difference of the 2 input equations
                return {diff.pop(), abs(num_diff)}
        elif isinstance(num2, int) and isinstance(eq1_neighbors, int):
            set1 = {eq1_neighbors} # covert the list into set
            set2 = {num2}
            diff = set1.difference(set2) # difference between 2 sets
            num_diff = num2 - eq1_neighbors # difference between the clue additions
            if any(diff):
                # return a new equation that is the difference of the 2 input equations
                return {diff.pop(), abs(num_diff)}

        else:  # for edge cases where the set has switched values
            set1 = {num1}
            set2 = {num2}
            diff = set1.difference(set2)
            num_diff = eq1_neighbors - eq2_neighbors
            if any(diff):
                # return a new equation that is the difference of the 2 input equations
                return {diff.pop(), abs(num_diff)}
```

Next, we find the difference between 2 equations and then use that difference to find a solution the same way as before. In the solver-1 function as solutions are found we update the knowledge base.

```
# Solver 1 uses the pseudocode from the writeup and implements it this way:
# Takes one equation we formed and looks at the number of variables (the number of cells in the
    equation) and the clue value.
# if clue value is equal to the number of cells in the list we infer that they are all mines and if
     the clue value is 0 we infer that all the cells in the list are safe
# and update the knowledge base accordingly.
def solver_1(self, eq1):

    if len(eq1) != 2:
        return False

    neighbors = eq1.pop()
```

```
num = eq1.pop()

if isinstance(num, int) and isinstance(neighbors, tuple):
    if len(neighbors) == num: # all cells are 1 meaning all cells are mines
        for i in range(0, len(neighbors)):
            temp = neighbors[i]
            # set (x,y) cell to 'm' to represent it is a mine
            self.bot_board[temp[0]][temp[1]] = 'm'
            if temp in self.possible_coordinates:
                self.possible_coordinates.remove(temp)
                self.mine_dict[temp[0]][temp[1]] = {"open": True, "mine": True}
                self.visited += 1
        return True # return true if the equations is solved

    if num == 0: # all cells are 0 meaning all cells are safe
        for i in range(0, len(neighbors)):
            temp = neighbors[i]
            self.bot_board[temp[0]][temp[1]] = self.board[temp[0]][temp[1]] # set (x,y) cell to 0
                 to represent it is safe
            if temp in self.possible_coordinates:
                self.possible_coordinates.remove(temp)
                self.mine_dict[temp[0]][temp[1]] = {"open": True, "mine": False}
                self.visited += 1
        return True # return true if the equations is solved

# for edge cases where the set has switched values
elif isinstance(neighbors, int) and isinstance(num, tuple):
    if len(num) == neighbors:
        for i in range(0, len(num)):
            temp = num[i]
            self.bot_board[temp[0]][temp[1]] = 'm'
            if temp in self.possible_coordinates:
                self.possible_coordinates.remove(temp)
                self.mine_dict[temp[0]][temp[1]] = {"open": True, "mine": True}
                self.visited += 1
        return True

    if neighbors == 0:
        for i in range(0, len(num)):
            temp = num[i]
            self.bot_board[temp[0]][temp[1]] = self.board[temp[0]][temp[1]]
            if temp in self.possible_coordinates:
                self.possible_coordinates.remove(temp)
                self.mine_dict[temp[0]][temp[1]] = {"open": True, "mine": False}
                self.visited += 1
        return True

return False # false if the equation is not solved
```

3.3 Decisions: Given a current state of the board, and a state of knowledge about the board, how does your program decide which cell to search next?

Once we open a cell it can either be a mine or a clue value. As for the clue value it can be 0, meaning none of it's neighbors have a mine, or a number greater than 0, meaning there is at least one mine in the neighboring cells.

Suppose the agent opens cell i, j. The program would decide which cell to search next by this process:

1. If the agent opens a cell that is mine, it is added to the knowledge base.
2. If the agent finds a clue value of 0 it is added to the knowledge base and we look through all it's neighbors.
   (a) If the neighbors have not been opened before, we add them to a stack named 'q'.
   (b) Then we go through the stack of neighbors one by one until it's empty.
   (c) If the neighbor has a clue value that is greater than 0, we add it to a queue named 'numbers'
      i. Which allows us to open all the safe cells around the given cell at i, j that has a clue value == 0 building the border of cells that have clue values that can be > 0.
3. If the clue value found is greater than 0 it is added to the knowledge base and the queue named 'numbers'. We then go through the 'numbers' stack and take 3 points at a time and try to get a solution from the equation built from the cell locations. If there is a solutions found it is updated in the knowledge base inside the solver function.
4. Lastly, we iterate cell by cell to find more solutions.
   (a) We go through the first neighbors of the origin cell and then the secondary neighbors of the origin cell.
   (b) These neighbor cells are added to the 'numbers' queue if they are already opened.
   (c) Then similar to before, 3 points are sent to the solver function to find a solution until the queue is empty. We do this for every cell and if there are solutions found in the process we iterate through the maze again to use the new knowledge.
   (d) If we exhausted all the possibilities to form a solution to be found we select a random point to be opened.

Then we keep playing until all the cells have been opened, either by the agent or have been marked as a mine by the solver.

3.4 Performance: For a reasonably-sized board and a reasonable number of mines, include a play-by-play progression to completion or loss. Are there any points where your program makes a decision that you don't agree with? Are there any points where your program made a decision that surprised you? Why was your program able to make that decision?

Below is a Play-By-Play of all the actions the advance agent took for a 10x10 board with a density of 10%.

Advance_Agent_Play_by_Play.txt

```
Advanced Agent Play-by-Play:

Dimensions: 10x10 and Density: 10%

Game Board:
[[’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’1’ ’1’ ’1’]
 [’2’ ’2’ ’1’ ’ ’ ’ ’ ’ ’ ’ ’ ’1’ ’m’ ’1’]
 [’m’ ’m’ ’1’ ’ ’ ’ ’ ’ ’ ’ ’ ’1’ ’1’ ’1’]
 [’m’ ’3’ ’1’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’]
 [’1’ ’1’ ’ ’ ’ ’ ’1’ ’1’ ’1’ ’ ’ ’ ’ ’ ’]
 [’ ’ ’1’ ’1’ ’1’ ’1’ ’m’ ’1’ ’ ’ ’ ’ ’ ’]
 [’ ’ ’1’ ’m’ ’2’ ’2’ ’2’ ’1’ ’ ’ ’1’ ’1’]
 [’1’ ’2’ ’1’ ’2’ ’m’ ’1’ ’ ’ ’ ’ ’1’ ’m’]
 [’m’ ’1’ ’1’ ’2’ ’2’ ’1’ ’ ’ ’ ’ ’1’ ’1’]
 [’1’ ’1’ ’1’ ’m’ ’1’ ’ ’ ’ ’ ’ ’ ’ ’ ’ ’]]

On Play #1 the agent choose 1 at row 2 and column 8
On Play #2 the agent opened a mine at row 5 and column 5
On Play #3 the agent choose ’ ’ at row 3 and column 9
On Play #4 the agent choose ’ ’ at row 4 and column 9
On Play #5 the agent choose 1 at row 2 and column 9
On Play #6 the agent choose ’ ’ at row 3 and column 8
On Play #7 the agent choose ’ ’ at row 4 and column 8
On Play #8 the agent choose ’ ’ at row 5 and column 8
On Play #9 the agent choose ’ ’ at row 4 and column 7
On Play #10 the agent choose ’ ’ at row 5 and column 9
On Play #11 the agent choose ’ ’ at row 5 and column 7
On Play #12 the agent choose ’ ’ at row 3 and column 7
On Play #13 the agent choose 1 at row 2 and column 7
On Play #14 the agent choose ’ ’ at row 3 and column 6
On Play #15 the agent choose 1 at row 4 and column 6
On Play #16 the agent choose ’ ’ at row 2 and column 6
On Play #17 the agent choose ’ ’ at row 1 and column 6
On Play #18 the agent choose ’ ’ at row 2 and column 5
On Play #19 the agent choose ’ ’ at row 3 and column 5
On Play #20 the agent choose ’ ’ at row 1 and column 5
On Play #21 the agent choose 1 at row 1 and column 7
On Play #22 the agent choose ’ ’ at row 0 and column 5
On Play #23 the agent choose ’ ’ at row 1 and column 4
On Play #24 the agent choose ’ ’ at row 2 and column 4
On Play #25 the agent choose ’ ’ at row 0 and column 4
On Play #26 the agent choose ’ ’ at row 0 and column 6
On Play #27 the agent choose 1 at row 0 and column 7
On Play #28 the agent choose ’ ’ at row 0 and column 3
On Play #29 the agent choose ’ ’ at row 1 and column 3
On Play #30 the agent choose ’ ’ at row 2 and column 3
On Play #31 the agent choose 1 at row 1 and column 2
On Play #32 the agent choose 1 at row 2 and column 2
On Play #33 the agent choose ’ ’ at row 0 and column 2
On Play #34 the agent choose ’ ’ at row 0 and column 1
On Play #35 the agent choose 2 at row 1 and column 1
On Play #36 the agent choose ’ ’ at row 0 and column 0
On Play #37 the agent choose 2 at row 1 and column 0
On Play #38 the agent choose ’ ’ at row 3 and column 3
On Play #39 the agent choose ’ ’ at row 3 and column 4
On Play #40 the agent choose 1 at row 3 and column 2
On Play #41 the agent choose 1 at row 4 and column 4
On Play #42 the agent choose 1 at row 4 and column 5
On Play #43 the agent choose ’ ’ at row 4 and column 3
On Play #44 the agent choose 1 at row 5 and column 3
On Play #45 the agent choose ’ ’ at row 4 and column 2
On Play #46 the agent choose 1 at row 5 and column 4
On Play #47 the agent choose 1 at row 5 and column 2
On Play #48 the agent choose 1 at row 4 and column 1
On Play #49 the agent choose 1 at row 5 and column 1
On Play #50 the agent choose 3 at row 3 and column 1
On Play #51 the agent choose ’ ’ at row 6 and column 7
On Play #52 the agent choose 1 at row 5 and column 6
On Play #53 the agent choose 1 at row 6 and column 8
On Play #54 the agent choose 1 at row 6 and column 6
On Play #55 the agent choose ’ ’ at row 7 and column 7
On Play #56 the agent choose 1 at row 7 and column 8
On Play #57 the agent choose ’ ’ at row 7 and column 6
On Play #58 the agent choose ’ ’ at row 8 and column 6
On Play #59 the agent choose 1 at row 7 and column 5
On Play #60 the agent choose ’ ’ at row 8 and column 7
On Play #61 the agent choose 1 at row 8 and column 5
On Play #62 the agent choose 2 at row 6 and column 5
On Play #63 the agent choose ’ ’ at row 9 and column 7
On Play #64 the agent choose 1 at row 8 and column 8
On Play #65 the agent choose ’ ’ at row 9 and column 8
On Play #66 the agent choose ’ ’ at row 9 and column 6
On Play #67 the agent choose ’ ’ at row 9 and column 5
On Play #68 the agent choose 1 at row 9 and column 4
On Play #69 the agent choose 2 at row 8 and column 4
On Play #70 the agent choose ’ ’ at row 9 and column 9
On Play #71 the agent choose 1 at row 8 and column 9
On Play #72 the agent choose 1 at row 6 and column 9
On Play #73 the agent discovered a mine at row 7 and column 9
On Play #74 the agent discovered a mine at row 1 and column 8
On Play #75 the agent choose 1 at row 1 and column 9
On Play #76 the agent choose 1 at row 0 and column 8
On Play #77 the agent discovered a mine at row 2 and column 1
On Play #78 the agent discovered a mine at row 2 and column 0
On Play #79 the agent choose 2 at row 6 and column 4
On Play #80 the agent choose 2 at row 6 and column 3
On Play #81 the agent discovered a mine at row 6 and column 2
On Play #82 the agent choose 1 at row 6 and column 1
On Play #83 the agent choose ’ ’ at row 5 and column 0
On Play #84 the agent choose ’ ’ at row 6 and column 0
On Play #85 the agent choose 1 at row 4 and column 0
On Play #86 the agent discovered a mine at row 3 and column 0
On Play #87 the agent discovered a mine at row 7 and column 4
On Play #88 the agent opened a mine at row 9 and column 3
On Play #89 the agent choose 2 at row 8 and column 3
On Play #90 the agent choose 2 at row 7 and column 3
On Play #91 the agent choose 1 at row 0 and column 9
On Play #92 the agent choose 2 at row 7 and column 1
On Play #93 the agent choose 1 at row 7 and column 2
On Play #94 the agent choose 1 at row 7 and column 0
On Play #95 the agent choose 1 at row 8 and column 2
On Play #96 the agent choose 1 at row 8 and column 1
On Play #97 the agent discovered a mine at row 8 and column 0
On Play #98 the agent choose 1 at row 9 and column 2
On Play #99 the agent choose 1 at row 9 and column 1
On Play #100 the agent choose 1 at row 9 and column 0
```

Based on the play-by-play we see how the agent was able to perform a similar function to the actual game in step 3 by opening all the surrounding neighbors of the cell that had a clue value if 0 that the bot clicked on. The program was able to make this decision as it inferred that all the neighbors of the cell with a clue value of 0 are safe to open. Doing this iteratively for each neighbor with a clue value of 0 we get a border of clue values that are greater than 0. The agent started finding solutions from from 73 with the new knowledge it was able to solve for more mines until step 87 where no more solutions could be found from the knowledge available, so it chose a random cell and it turned out to be a mine. There are times when we don't agree with the random cell the agent opens when it can not find any more solutions from the available knowledge. We believe the random choice can be made better by choosing a random point that is either close to a cell with lower clue values or in an area with a lot of unexplored cells. There we no decisions that surprised us since all of it's decisions were based on all the knowledge that is available to it. Because all the decisions are based on the knowledge base it was able to solve to competition where it chose all the safe cells accurately without triggering a mine.

3.5 Performance: For a fixed, reasonable size of board, plot as a function of mine density the average final score (safely identified mines / total mines) for the simple baseline algorithm and your algorithm for comparison. This will require solving multiple random boards at a given density of mines to get good average score results. Does the graph make sense / agree with your intuition? When does minesweeper become 'hard'? When does your algorithm beat the simple algorithm, and when is the simple algorithm better? Why? How frequently is your algorithm able to work out things that the basic agent cannot?

For this performance test we wanted to demonstrate the capabilities of the advance agent and how much it out performs the basic agent so we choose to test on a $30X30$ board and have the mine densities range from (10% to 50%). For the test we decided to run the tests (5 tests for each mine density) side by side for the basic agent and advance agent to get the most accurate results. The figure below is the results we got from conducting this test.

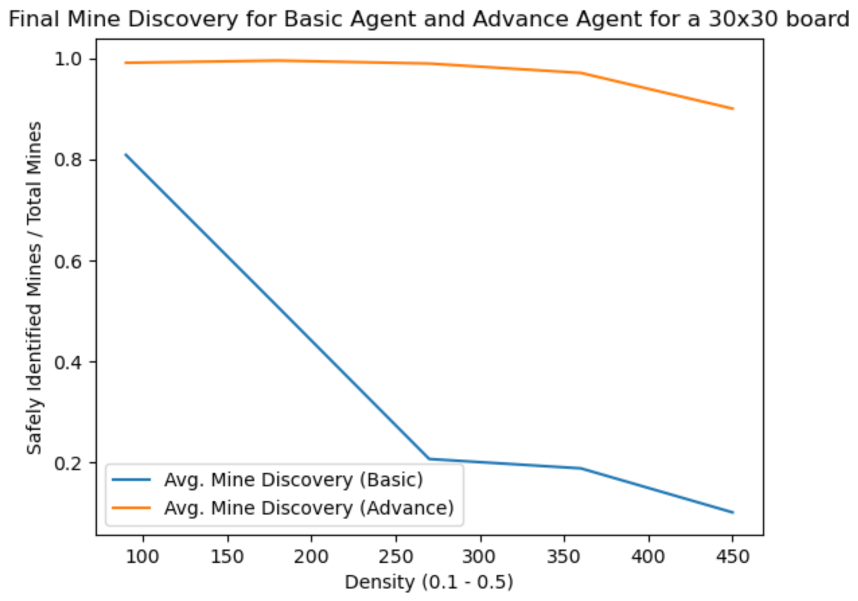

Figure 1: Final Average Mine Discovery for the Basic Agent and Advance Agent

From the way we implemented the basic agent and advance agent for this project the graphs do make sense with our intuition. The basic agent is all local data so it wouldn't be able make the right decisions that the advance agent hence it doing well with low densities (10%) and immediately dipping because it wouldn't be able to make the right decision because there are many more mines and its working off local knowledge known to the cell. Then the advance agent works on global knowledge which would help way more because at each cell it knows what decisions to make based on changes that happened 2 cells away from it. As we see when the mine densities go to 35% and up for both agents it tends to dip down more because there are more mines to work with (even though the advance agent tends to stay in the range (90% success - 100% success). For our implementation, the advance agent tends to beat the basic agent in all cases because how much more information the advance agent is working with. Like I mentioned before local knowledge will get the agent so far until it starts to not be able to make the right inferences. While the advance agent can work on the inferences it premade and make more inferences based on more information it gathers and reduce the irrelevant knowledge. In all cases, the advance agent is able to work out things that the basic agent cannot because with local knowledge the basic agent can get so far until it reaches a point where its not sure where to go to next. While, far in the future the advance agent is able to deduce an inference that can help it succeed way more often.

3.6 Efficiency: What are some of the space or time constraints you run into in implementing this program? Are these problem specific constraints, or implementation specific constraints? In the case of implementation constraints, what could you improve on?

A space constraint that we ran into when implementing this program is the usage of a stack to store empty neighbors for a particular empty cell. Through our implementation, the stack would contain empty neighbor cells and also empty neighbors of neighbor cells. In the particular case where a cell is empty and all of its neighbor cells are empty, the stack would contain all eight empty neighbors cells along with empty neighbor cells for each one of those eight neighbors. This would possibly lead to the size of stack becoming large, and therefore taking up a lot of space. The size of the stack would grow $8, 16, ..., 8n$ where n is the level outward from the initial origin cell, making the space complexity O(n). Even though in our implementation the stack has linear space complexity is not bad this is specific constraint can be solved by using a heap. A heap is a better option than a stack because allocation of memory is better for heaps than stacks. Where the heap size can be dynamically allocated where it would grow slower or faster depending on the cell. So heaps are better equipped to handle large amounts of data.

4 Bonus

4.1 Global Information: +10 points available Suppose you were told in advance how many mines are on the board. Include this in your knowledge base. How did you model this? Regenerate the plot of mine density vs average final score with this extra information, and analyze the results.

We decided to model knowing how many mines were on the board via a field for the class. Upon creating the board we made a field that held how many mines were placed on the board. Then upon launching the advance agent we added a condition that it should continue until visited variable == self.dim∗self.dim or all the mines discovered == this added field variable of the total amount of mines placed on the board. Conducting the same mine density vs average final score test again we compared the old agent without this global information to the advance agent with the global information and got the results below.

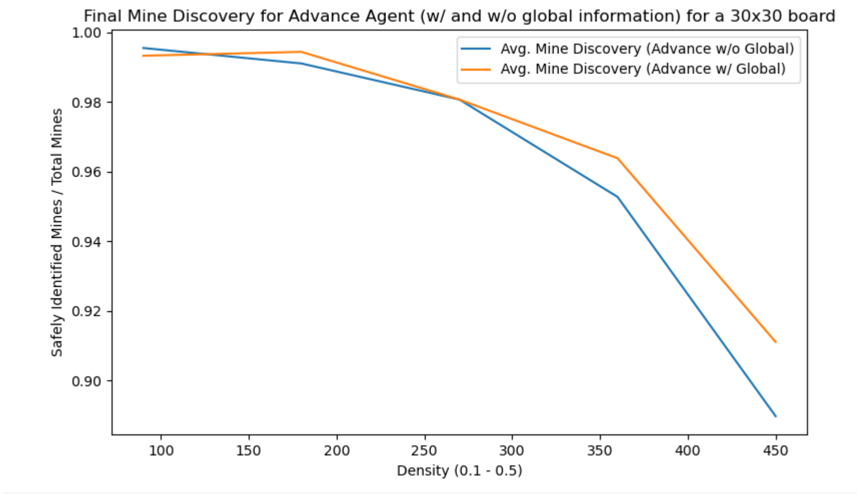

Figure 2: Final Average Mine Discovery for the Advance Agent with and without global information

From the figure above we see that the advance agent with the new global information was able to perform about the same as the advance agent without this global information. But the agent with the added global information had reduced the amount of computations time as it would terminate if all the mines have been found. This is able to reduce the amount of moves the advance agent has to make, which lead it to not explore the entire board unlike the advance agent, without the knowledge of total mines on the board, would have to do before terminating

5 Contributions and Acknowledgments about Rutgers Academic Policy

Group Member: Karun Kanda **Netid:** kk951 **RUID:** 184007619

For the final write up I worked on the two performance questions because I worked on the performance aspect of the advanced agent and worked on the code for the basic agent.

I, Karun Kanda, certify that the work submitted in this project is my own work and not copied or taken from online or any other student's work.

Group Member: Varun Gaddam **Netid:** vrg24 **RUID:** 187008136

For the final write up I worked on the representation and efficiency questions because I worked on the advanced agent representation and code.

I, Varun Gaddam, certify that the work submitted in this project is my own work and not copied or taken from online or any other student's work.

Group Member: Agam Modasiya **Netid:** ajm432 **RUID:** 185009911

For the final write up I worked on the decisions and inferences part because I was responsible for coding those parts.

I, Agam Modasiya, certify that the work submitted in this project is my own work and not copied or taken from online or any other student's work.