SCHOOL OF SCIENCE & ENGINEERING

SPRING 2024

CSC 4301

Term Project: Report

April 2nd, 2024

Realized by:

Imane Bouchtaoui

Chahd Maatallaoui

Supervised by:

Professor RACHIDI, Tajjedine

TABLE OF CONTENTS

I - Introduction:

Our third project consists of the implementation of the "Bust the ghost" game, an interactive game that uses a grid-based environment. The goal is to locate the ghost within the "grid" and bust it by using the sensory cues provided within the game, combining logics and deduction reasoning alongside the usage of probabilities. The player gets 10 attempts when it comes to the number of cells it can click but only 2 attempts to bust one of the clicked cells to find the ghost. The player can choose to showcase the probabilities of each one of the squares, which gets updated after each click of the cells, using Bayesian probabilistic inferencing. After each click, a color is displayed based on the Manhattan distance between the clicked cells and the ghost's position and by sampling the appropriate color (and the appropriate direction in the case of the bonus) from the conditional distribution table.

We developed this game using the python library Tkinter, as it allows us to create a responsive interface for the game as well as implement the needed elements and logic allowing the player to have a smooth experience.

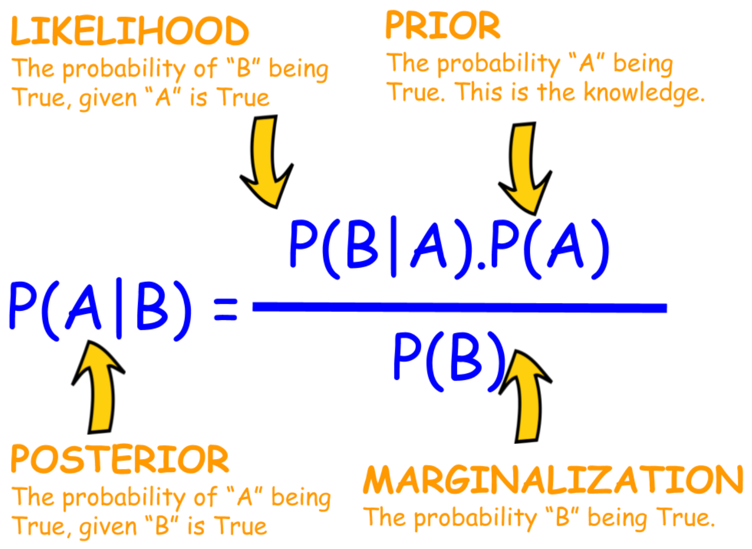
Our tasks were split into the following:

Task	Person in Charge
Setting up the User interface	Chahd
Implementation of the distance sensors	Chahd
Implementation of the direction sensors	Imane
User interface update	Imane
Report Writing	Imane and Chahd

II - Bayesian Probabilistic Inferencing:

Bayesian Inference is a method used in statics which makes usage of Bayes' theorem to update the probability of a hypothesis as we get more information being made available. Bayesian inference consists therefore of updating the probability estimates regarding certain hypotheses/parameters based on prior knowledge as well as new acquired data. It therefore differs from classical statistical inference which mainly focuses on frequency and likelihood without taking prior consideration into consideration.

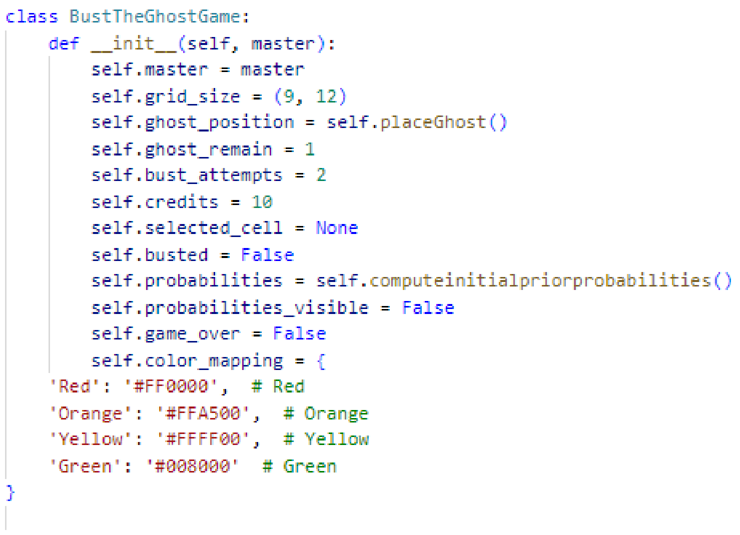
As it's heart, we find the Bayes' theorem which describes how to update the probabilities of hypotheses when provided with further evidence. It states that posterior probabilities are proportial to the likelihood times the prior probabilities.


LIKELIHOOD
The probability of "B" being True, given "A" is True
PRIOR
The probability "A" being True. This is the knowledge.
$P(A|B) = \frac{P(B|A).P(A)}{P(B)}$
POSTERIOR
The probability of "A" being True, given "B" is True
MARGINALIZATION
The probability "B" being True.

One of the strengths of Bayesian inference is how it gives the ability to quantify uncertainty by working with probability distributions rather than single points estimates.

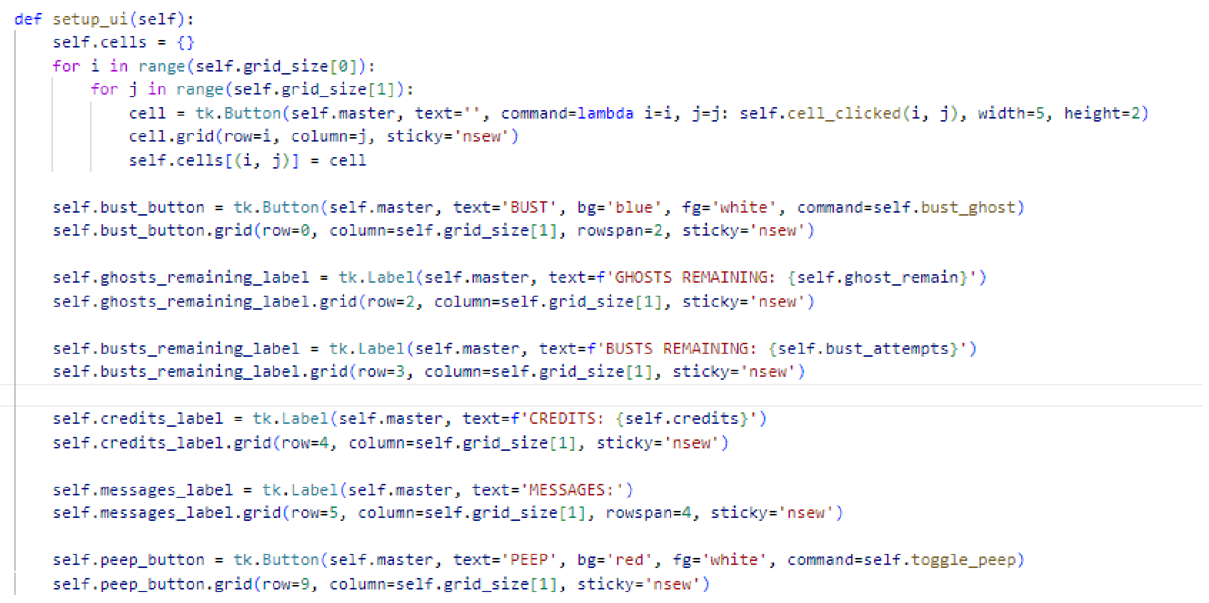
III - Initial Set up of the Interface:

i. General Set up:



```
class BustTheGhostGame:
    def __init__(self, master):
        self.master = master
        self.grid_size = (9, 12)
        self.ghost_position = self.placeGhost()
        self.ghost_remain = 1
        self.bust_attempts = 2
        self.credits = 10
        self.selected_cell = None
        self.busted = False
        self.probabilities = self.computeinitialpriorprobabilities()
        self.probabilities_visible = False
        self.game_over = False
        self.color_mapping = {
    'Red': '#FF0000',   # Red
    'Orange': '#FFA500',   # Orange
    'Yellow': '#FFFF00',   # Yellow
    'Green': '#008000'   # Green
}
```

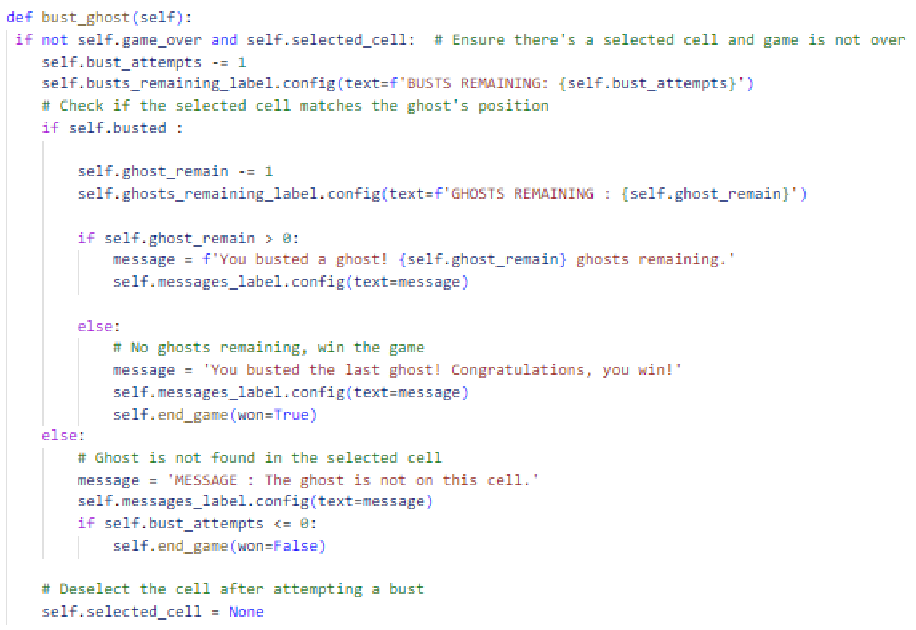
Using python’s object-oriented properties, everything will be put within a BustTheGhostGame class. Within the method **__init__(self, master)**, we start by initializing the class as well as all its attributes. The master here is used as a variable to store the main Tkinter window that will be containing the game’s GUI. For the **grid_size**, we set it to be a (9,12) grid as per the requirements given. The **ghost_position** returns contain the xg and yg coordinates returned by the **placeGhost()** function, which places the ghost in a random cell within a 7 x 13 domain, as precised in the requierements. In the **ghost_remain** variable, we initially store how many ghosts we have (which is 1 here in our case) and we will be decrementing it whenever a ghost is successfully busted. The **bust_attempts** variable stores the number of busts that we can attempt (here, it is set to 2 as per the requierements) while the **credits** variable stories the number of cells that we can click and as soon as we go over that, we get a game over message and the game stops (here, we chose to set it to 10 in order to give the player enough chances while also not making the game too long). The **selected_cell** variable will be used to store the current selected cell and the probabilities variable will come store the initial probabilities before one of the cells are clicked and we update them, and the **busted** variable indicates whether the ghost was busted or not and initially equal to a False. To decide if we show the probabilities or not using the Peep button, we set a **probabilities_visible** variable initialized to False, and we also set a **game_over** variable initialized to False as well. Finally, we have a variable that dictionary which will come to store the color codes for all the needed colours.



```python
def setup_ui(self):
    self.cells = {}
    for i in range(self.grid_size[0]):
        for j in range(self.grid_size[1]):
            cell = tk.Button(self.master, text='', command=lambda i=i, j=j: self.cell_clicked(i, j), width=5, height=2)
            cell.grid(row=i, column=j, sticky='nsew')
            self.cells[(i, j)] = cell

    self.bust_button = tk.Button(self.master, text='BUST', bg='blue', fg='white', command=self.bust_ghost)
    self.bust_button.grid(row=0, column=self.grid_size[1], rowspan=2, sticky='nsew')

    self.ghosts_remaining_label = tk.Label(self.master, text=f'GHOSTS REMAINING: {self.ghost_remain}')
    self.ghosts_remaining_label.grid(row=2, column=self.grid_size[1], sticky='nsew')

    self.busts_remaining_label = tk.Label(self.master, text=f'BUSTS REMAINING: {self.bust_attempts}')
    self.busts_remaining_label.grid(row=3, column=self.grid_size[1], sticky='nsew')

    self.credits_label = tk.Label(self.master, text=f'CREDITS: {self.credits}')
    self.credits_label.grid(row=4, column=self.grid_size[1], sticky='nsew')

    self.messages_label = tk.Label(self.master, text='MESSAGES:')
    self.messages_label.grid(row=5, column=self.grid_size[1], rowspan=4, sticky='nsew')

    self.peep_button = tk.Button(self.master, text='PEEP', bg='red', fg='white', command=self.toggle_peep)
    self.peep_button.grid(row=9, column=self.grid_size[1], sticky='nsew')
```

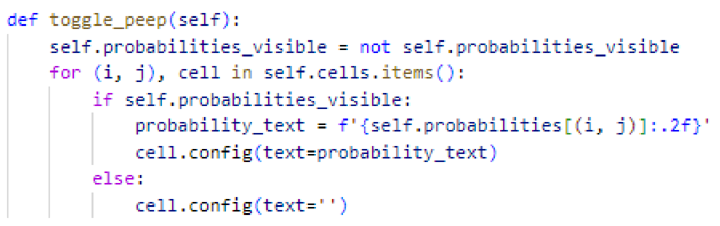
Here, the **setup_ui** method is used to initialize the user interface of the game using the Tkinter library. We create a 9 x 12 grid representing the game's cells stores in a dictionary for easy access, and each one of them is assigned the **cell_clicked** event handler. Additionally, we have special control labels buttons:

- A "Bust" button which triggers an attempt to bust a cell to find the ghost



```python
def bust_ghost(self):
 if not self.game_over and self.selected_cell:  # Ensure there's a selected cell and game is not over
    self.bust_attempts -= 1
    self.busts_remaining_label.config(text=f'BUSTS REMAINING: {self.bust_attempts}')
    # Check if the selected cell matches the ghost's position
    if self.busted :

        self.ghost_remain -= 1
        self.ghosts_remaining_label.config(text=f'GHOSTS REMAINING : {self.ghost_remain}')

        if self.ghost_remain > 0:
            message = f'You busted a ghost! {self.ghost_remain} ghosts remaining.'
            self.messages_label.config(text=message)

        else:
            # No ghosts remaining, win the game
            message = 'You busted the last ghost! Congratulations, you win!'
            self.messages_label.config(text=message)
            self.end_game(won=True)
    else:
        # Ghost is not found in the selected cell
        message = 'MESSAGE : The ghost is not on this cell.'
        self.messages_label.config(text=message)
        if self.bust_attempts <= 0:
            self.end_game(won=False)

    # Deselect the cell after attempting a bust
    self.selected_cell = None
```

- A “Peep” button which allows that can be activated and deactivated by the player to get hints about the position of the position of the ghost using the updated probabilities displayed in each cell after each click

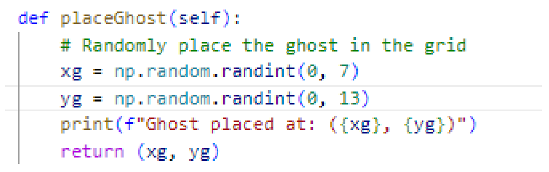


```python
def toggle_peep(self):
    self.probabilities_visible = not self.probabilities_visible
    for (i, j), cell in self.cells.items():
        if self.probabilities_visible:
            probability_text = f'{self.probabilities[(i, j)]:.2f}'
            cell.config(text=probability_text)
        else:
            cell.config(text='')
```

Additionally, we have label spaces:

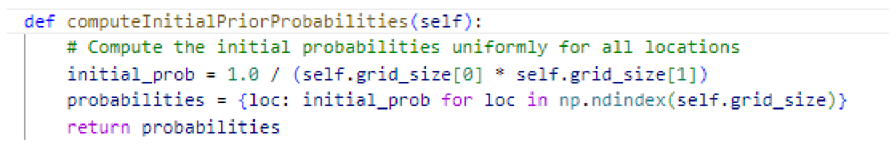
- A label that displays the number of ghosts remaining, the credits (number of cells that can still be clicked) and the remaining bust attempts
- A message label that gives the outcomes to the user for feedback

ii. Addition of the functionalities



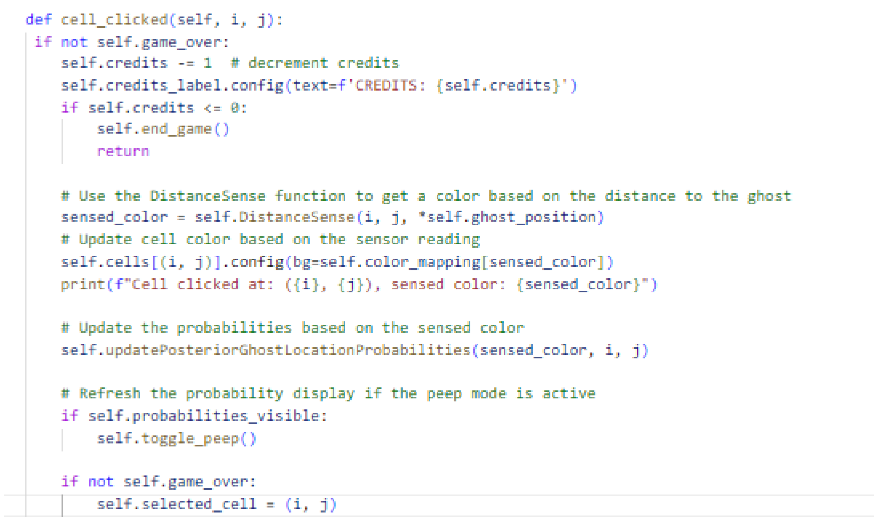
```python
def placeGhost(self):
    # Randomly place the ghost in the grid
    xg = np.random.randint(0, 7)
    yg = np.random.randint(0, 13)
    print(f"Ghost placed at: ({xg}, {yg})")
    return (xg, yg)
```

In the **placeGhost()** function, we generate a random integer between 1 and 7 for the x coordinates of the ghost and between 1 and 13 for the y coordinates of the ghost. We then return the two variables which will be stored in the **ghost_position** variable when we did the initiation of the class.



```python
def computeInitialPriorProbabilities(self):
    # Compute the initial probabilities uniformly for all locations
    initial_prob = 1.0 / (self.grid_size[0] * self.grid_size[1])
    probabilities = {loc: initial_prob for loc in np.ndindex(self.grid_size)}
    return probabilities
```

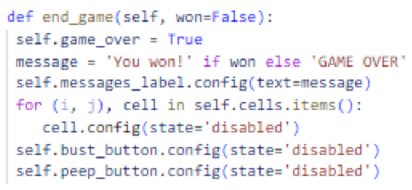
Here, the **computeInitialPriorProbabilities()** function initializes a probability distribution across the grid by assigning to each cell an equal probability of containing the ghost. To do this, we divide 1.0 by the total of cells that we have (108 cells for a 9x12 grid). This will make sure that the sum of the probabilities across all the cells equals to 1. Here, we find 0.00926 for each cell but tkinter rounds it to 0.01 to make it more readable. We then store this into a **probabilities** dictionary where each cell coordinate is linked with its probability, and it is then returned.



```python
def cell_clicked(self, i, j):
    if not self.game_over:
        self.credits -= 1  # decrement credits
        self.credits_label.config(text=f'CREDITS: {self.credits}')
        if self.credits <= 0:
            self.end_game()
            return

        # Use the DistanceSense function to get a color based on the distance to the ghost
        sensed_color = self.DistanceSense(i, j, *self.ghost_position)
        # Update cell color based on the sensor reading
        self.cells[(i, j)].config(bg=self.color_mapping[sensed_color])
        print(f"Cell clicked at: ({i}, {j}), sensed color: {sensed_color}")

        # Update the probabilities based on the sensed color
        self.updatePosteriorGhostLocationProbabilities(sensed_color, i, j)

        # Refresh the probability display if the peep mode is active
        if self.probabilities_visible:
            self.toggle_peep()

        if not self.game_over:
            self.selected_cell = (i, j)
```

The **cell_clicked()** method handles the interactios within the grid. After each click of the cells, the player's credits are decremented. If the numbers of credits that are left is equal to zero, then we end the game. Within this method, we call the DistanceSense method which evaluates the clicked cell's proximity to the ghost and returns the appropriate color based on the conditional distribution table. The cell's appearance is then updated to reflect the picked color, helping guide the user. Additionally, it updates the probability distribution of the ghost's location based on the new provided information by calling the

updatePosteriorGhostLocationProbabilities() method. Finally, we store the current clicked cell into the **selected_cell** variable.



```
def end_game(self, won=False):
 self.game_over = True
 message = 'You won!' if won else 'GAME OVER'
 self.messages_label.config(text=message)
 for (i, j), cell in self.cells.items():
    cell.config(state='disabled')
 self.bust_button.config(state='disabled')
 self.peep_button.config(state='disabled')
```

The **end_game()** method is used here to terminate the game. We start by setting the **game_over** variable to True so that we have no further action. We then display to the player either a “You Won!” message if the victory conditions have been met or “GAME OVER” one otherwise. After this, we disable all the interactive elements of the interface (cell in grid, control buttons..) so that we don’t have any further interaction and we freeze the game state to show that the game has ended.

IV. Implementation of the Distance Sensors:

The equation P(Ghost) = P(Ghost/Colour) = P(Ghost(t-1)) * P (Colour/Distance from Ghost) exemplifies the application of Bayesian inference to revise the likelihood of the ghost being in a particular cell. Initially, the prior probability P(Ghost) indicates the baseline probability of the ghost's presence in each cell before any sensor data is considered.

Following a player's selection of a cell, a sensor reading is conducted, yielding a colour determined by the conditional probability distribution P (Colour/Distance from Ghost). This distribution signifies the probability of observing a specific colour given the ghost's distance from the selected cell. Subsequently, this distribution informs the update of the posterior probability regarding the ghost's presence in that cell based on the latest sensor data.

In this game, probabilistic inference plays a crucial role in estimating the likelihood of the ghost's presence in specific locations based on observed tile colours and their respective distances from the ghost. This process relies on probability theory principles and the information contained within the joint probability table. Here's how it operates within the context of the game:

Joint probability table: This table encompasses joint probability values for each color and distance combination, indicating the likelihood of observing a particular color at a given distance from the ghost.

For the distance sensors, this is the conditional probability table that has been used:

	Red	Orange	Yellow	Green
0	0.80	0.15	0.04	0.01
1	0.3	0.6	0.07	0.03
2	0.3	0.6	0.07	0.03
3	0.03	0.07	0.6	0.3
4	0.03	0.07	0.6	0.3
>= 5	0.03	0.07	0.3	0.6

For distance = 0 :

Color	Probability
Red	0.80
Orange	0.15
Yellow	0.04
Green	0.01

For distance = 1 or distance = 2 :

Color	Probability
Red	0.3
Orange	0.6
Yellow	0.07
Green	0.03

For distance = 3 or distance = 4 :

Color	Probability
Red	0.03
Orange	0.07
Yellow	0.6
Green	0.3

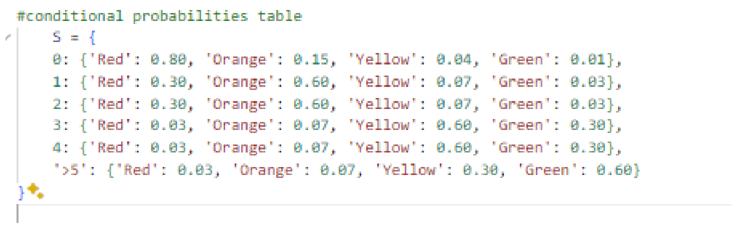
For distance >= 5 :

Color	Probability
Red	0.03
Orange	0.07
Yellow	0.3
Green	0.6

Which reflects on the rules given by the game stating that:

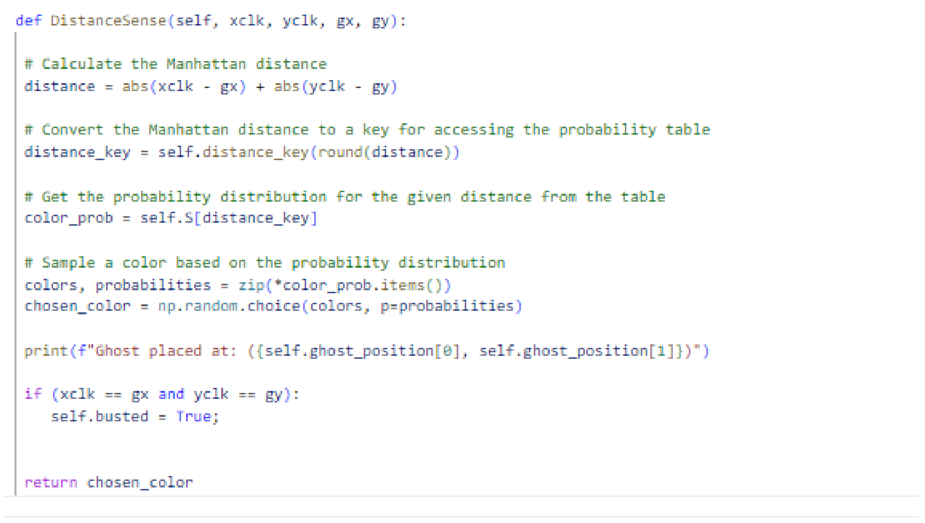
On the ghost: red
1 or 2 cells away: orange
3 or 4 cells away: yellow
5+ cells away: green

We then converted into a dictionary where the key is the distance which we will be using later:



```
#conditional probabilities table
    S = {
    0: {'Red': 0.80, 'Orange': 0.15, 'Yellow': 0.04, 'Green': 0.01},
    1: {'Red': 0.30, 'Orange': 0.60, 'Yellow': 0.07, 'Green': 0.03},
    2: {'Red': 0.30, 'Orange': 0.60, 'Yellow': 0.07, 'Green': 0.03},
    3: {'Red': 0.03, 'Orange': 0.07, 'Yellow': 0.60, 'Green': 0.30},
    4: {'Red': 0.03, 'Orange': 0.07, 'Yellow': 0.60, 'Green': 0.30},
    '>5': {'Red': 0.03, 'Orange': 0.07, 'Yellow': 0.30, 'Green': 0.60}
}
```

Observation: When the player selects a tile, they observe its colour. This observation serves as evidence regarding the ghost's potential location, with different colours carrying different probabilities depending on their distances from the ghost.



```python
def DistanceSense(self, xclk, yclk, gx, gy):

    # Calculate the Manhattan distance
    distance = abs(xclk - gx) + abs(yclk - gy)

    # Convert the Manhattan distance to a key for accessing the probability table
    distance_key = self.distance_key(round(distance))

    # Get the probability distribution for the given distance from the table
    color_prob = self.S[distance_key]

    # Sample a color based on the probability distribution
    colors, probabilities = zip(*color_prob.items())
    chosen_color = np.random.choice(colors, p=probabilities)

    print(f"Ghost placed at: ({self.ghost_position[0], self.ghost_position[1]})")

    if (xclk == gx and yclk == gy):
       self.busted = True;


    return chosen_color
```

The DistanceSense() method helps the players see how close they are to the hidden ghost within the grid. We start by calculating the distance between the ghost (**gx and gy**) and the selected cell (**xclk and yclk**) by using the Manhattan distance, which is more suited for grid environments. The distance is then rounded and used to access the specific distribution from the previously defined table S using **color_prob = self.S[distance_key]**, where we have an association of the different distances with their probabilities for each color. We then conduct a sampling from the retrieved distribution using **colors, probabilities = zip(*color_prob.items())** which extracts the color names alongside their associated probabilities from the distribution into lists. Then, using **chosen_color = np.random.choice(colors, p=probabilities),** we randomly selected one of the colours, with a selection weight by the probabilities. We therefore provide a color-coded indication about how close we are to the ghost to the player. If the clicked cell corresponds to the cell where the ghost is located, the **busted** variable is set to true. This will make it able for us to use in the **bust_ghost()** function while ensuring that the location of the ghost is only used in the **distanceSense()** function. Finally, the chosen color is returned.

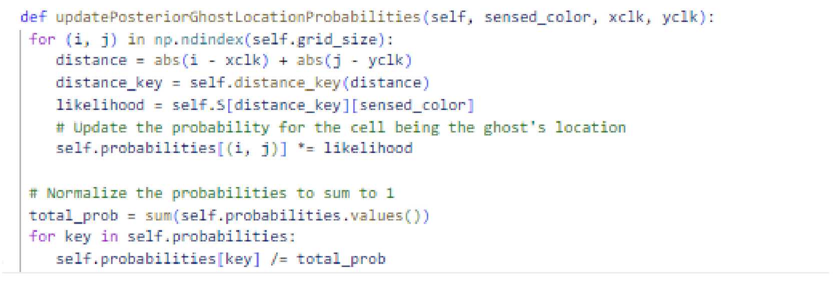
Inference: After determining the likelihood based on the sensed color, the method updates the probabilities for each cell using the **updatePosteriorGhostLocationProbabilities** method. In this method, the likelihood obtained from the joint probability table is used to update the probability for each cell, multiplying the current probability by the likelihood.

Here, we make usage of the Bayesian inference as follows:

$P_t(G=L_i) = P_t(G=L_i / S=\text{Color at location } L_i) = P(S=\text{Color at location } L_{i/} / G=L_i) * P_{t-1}(G=L_j)$

With $P_0 (G=L_j)$ is a uniform distribution (Initial prior probability)

And $P(S=\text{Color at location } L_{i/} / G=L_i) = P(S=\text{Color}/ \text{distance}=0)$.



```
def updatePosteriorGhostLocationProbabilities(self, sensed_color, xclk, yclk):
 for (i, j) in np.ndindex(self.grid_size):
    distance = abs(i - xclk) + abs(j - yclk)
    distance_key = self.distance_key(distance)
    likelihood = self.S[distance_key][sensed_color]
    # Update the probability for the cell being the ghost's location
    self.probabilities[(i, j)] *= likelihood

 # Normalize the probabilities to sum to 1
 total_prob = sum(self.probabilities.values())
 for key in self.probabilities:
    self.probabilities[key] /= total_prob
```

We loop through the grid cells by using **np.ndindex(self.grid_size)** which generates a pair of (i,j) for each cell. After that, we calculate the distance between the location and the cell where the color was sensed and using this distance, the function retreaves a distance_key and then looks up for the **likelihood** of observing the **sensed_color** we generated from that distance using the conditional table S we have defined previously. The probability of the ghost being in each cell is then updated by multiplying the probabilities we have in **self.probabilities[(i,**

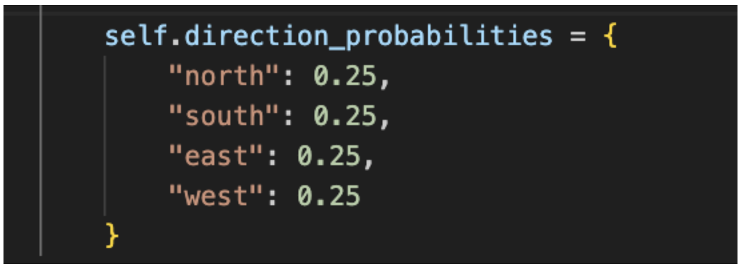
j)] by the like likelihood of observing the sensed color at that distance, applying therefore Bayesian Inferance and updating the belief about where the ghost is depending on the new evidence. Finally, we normalize the entire probability distribution to make sure that their sum equals to one. We do this by dividing each cell’s updated probability by the sum of all updated probabilities.

V. Implementation of the Direction Sensors:

The direction sensor provides valuable information to the player, helping them narrow down the ghost's location and strategize their bust attempts. By increasing the probability of the ghost being in a specific direction based on the player's position, the sensor refines the initial uniform probability distribution.

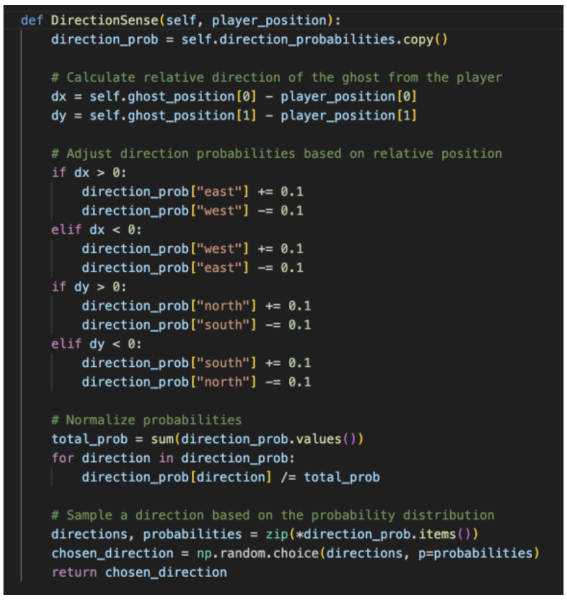
“Functionality”

- Initial Distribution: All directions (north, south, east, west) have an equal probability (0.25). This represents a lack of prior knowledge about the ghost's location**.**



```
self.direction_probabilities = {
    "north": 0.25,
    "south": 0.25,
    "east": 0.25,
    "west": 0.25
}
```

- Conditional Distribution Based on Relative Position: The DirectionSense function takes the player's current cell coordinates as input and calculates the relative displacement (difference in x and y coordinates) between the player's position and the ghost's location.



```python
def DirectionSense(self, player_position):
    direction_prob = self.direction_probabilities.copy()

    # Calculate relative direction of the ghost from the player
    dx = self.ghost_position[0] - player_position[0]
    dy = self.ghost_position[1] - player_position[1]

    # Adjust direction probabilities based on relative position
    if dx > 0:
        direction_prob["east"] += 0.1
        direction_prob["west"] -= 0.1
    elif dx < 0:
        direction_prob["west"] += 0.1
        direction_prob["east"] -= 0.1
    if dy > 0:
        direction_prob["north"] += 0.1
        direction_prob["south"] -= 0.1
    elif dy < 0:
        direction_prob["south"] += 0.1
        direction_prob["north"] -= 0.1

    # Normalize probabilities
    total_prob = sum(direction_prob.values())
    for direction in direction_prob:
        direction_prob[direction] /= total_prob

    # Sample a direction based on the probability distribution
    directions, probabilities = zip(*direction_prob.items())
    chosen_direction = np.random.choice(directions, p=probabilities)
    return chosen_direction
```

Based on this displacement, the function determines the most likely direction (north, south, east, or west) the ghost resides in:

- Positive x-displacement suggests the ghost is to the east, and vice versa.
- Positive y-displacement suggests the ghost is to the north, and vice versa.

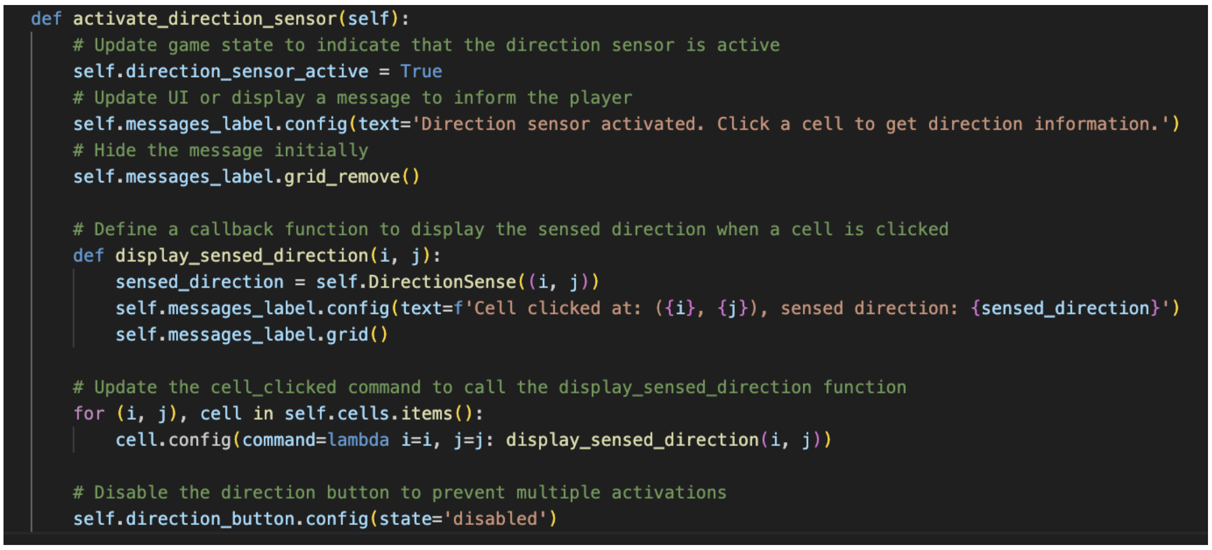
The function then adjusts the initial uniform probabilities for each direction. It increases the probability in the direction corresponding to the calculated displacement and decreases the probability in the opposite direction. To account for uncertainty, the adjustments are not absolute but rather nudge the probabilities in the favoured direction.

*Finally, the function normalizes the probabilities to ensure they sum to 1 and performs a random selection based on the updated probability distribution.

"Integration with Game Mechanics"

- A button (DIRECTION) is included in the game UI to activate the direction sensor.

  - Clicking this button triggers the activate_direction_sensor function. (This function updates the game state to indicate that the direction sensor is active and displays a message informing the player.)



```python
def activate_direction_sensor(self):
    # Update game state to indicate that the direction sensor is active
    self.direction_sensor_active = True
    # Update UI or display a message to inform the player
    self.messages_label.config(text='Direction sensor activated. Click a cell to get direction information.')
    # Hide the message initially
    self.messages_label.grid_remove()

    # Define a callback function to display the sensed direction when a cell is clicked
    def display_sensed_direction(i, j):
        sensed_direction = self.DirectionSense((i, j))
        self.messages_label.config(text=f'Cell clicked at: ({i}, {j}), sensed direction: {sensed_direction}')
        self.messages_label.grid()

    # Update the cell_clicked command to call the display_sensed_direction function
    for (i, j), cell in self.cells.items():
        cell.config(command=lambda i=i, j=j: display_sensed_direction(i, j))

    # Disable the direction button to prevent multiple activations
    self.direction_button.config(state='disabled')
```

- Clicking on a cell after activating the sensor calls a call back function (display_sensed_direction).
  - This call back function retrieves the sensed direction using DirectionSense and displays it along with the clicked cell coordinates in the message label.

$\Rightarrow$ $P_t(G=L_i)=P_t(G=L_i|S_{\text{color at location } L_i},S_{\text{direction at location } L_i})\times P_{t-1}(G=L_j)$

Where:

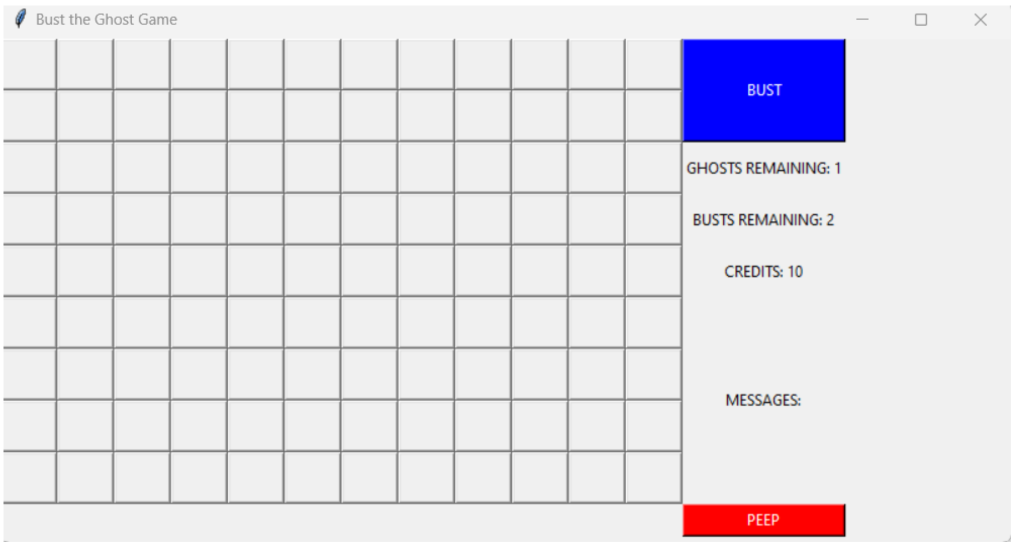
- $P_t(G=L_i|S_{\text{color at location } L_i},S_{\text{direction at location } L_i})$ represents the combined probability of the ghost being at location L_i given the sensed color and direction at that location.
- $P_{t-1}(G=L_j)$ represents the prior probability of the ghost being at location L_j.

In this implementation, the **updatePosteriorGhostLocationProbabilities()** function incorporates evidence from the sensed color, adjusting the probabilities for each location based on the likelihood of the observed color. On the other hand, the **update_probabilities_based_on_direction** function complements this by considering the sensed direction, further refining the probabilities by incorporating directional information relative to the player's position.

These two operations synergistically refine the posterior probabilities, providing comprehensive inference mechanism for determining the ghost's location. By integrating evidence from both sensors into a unified probabilistic framework, our implementation enhances the accuracy of ghost localization, enriching the gameplay experience with nuanced decision-making based on combined sensor inputs.

VI. Demo of the project:

Initial UI:



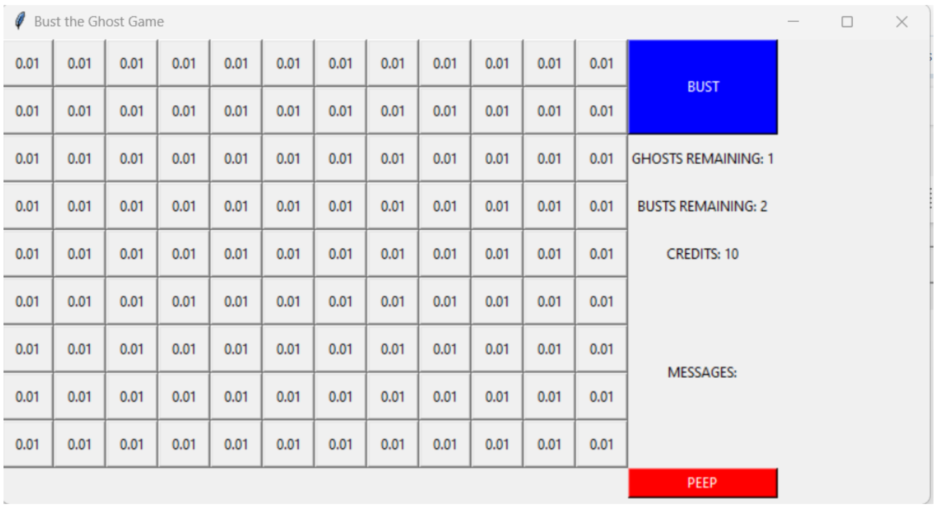
GHOSTS REMAINING: 1

BUSTS REMAINING: 2

CREDITS: 10

MESSAGES:

CSC 4301 Project #3



Bust the Ghost Game

0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

BUST

GHOSTS REMAINING: 1

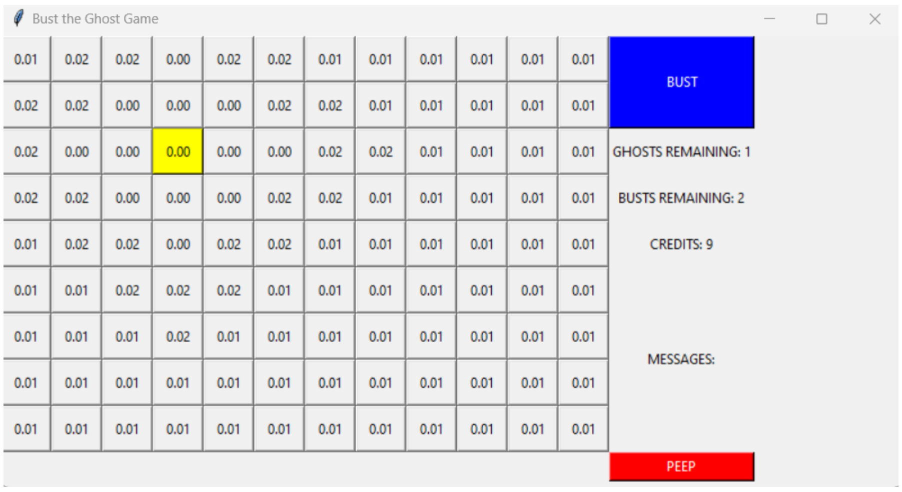
BUSTS REMAINING: 2

CREDITS: 10

MESSAGES:

PEEP

When cell selected:



Bust the Ghost Game

0.01	0.02	0.02	0.00	0.02	0.02	0.01	0.01	0.01	0.01	0.01	0.01
0.02	0.02	0.00	0.00	0.00	0.02	0.02	0.01	0.01	0.01	0.01	0.01
0.02	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.01	0.01	0.01	0.01
0.02	0.02	0.00	0.00	0.00	0.02	0.02	0.01	0.01	0.01	0.01	0.01
0.01	0.02	0.02	0.00	0.02	0.02	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.02	0.02	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.02	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

BUST

GHOSTS REMAINING: 1

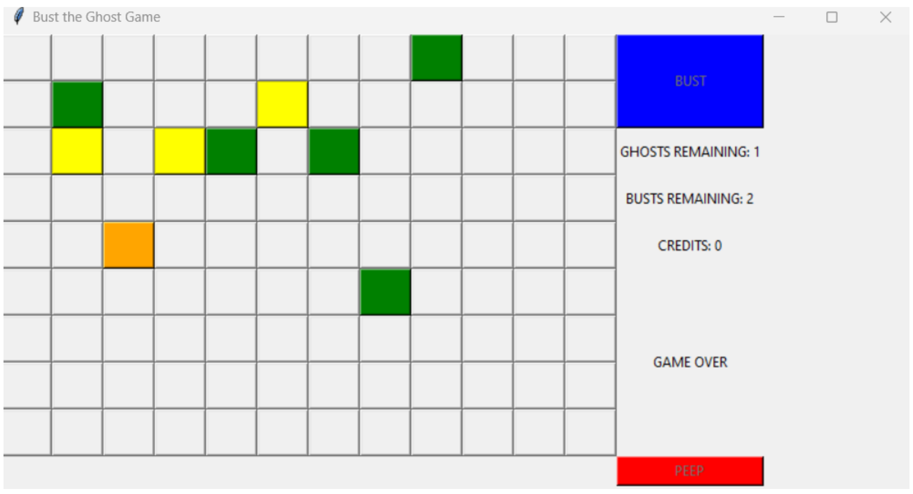
BUSTS REMAINING: 2

CREDITS: 9

MESSAGES:

PEEP

When the player runs out of credits:



Bust the Ghost Game

								🟩			
	🟩				🟨						
	🟨		🟨	🟩		🟩					
		🟧									
							🟩				

BUST

GHOSTS REMAINING: 1

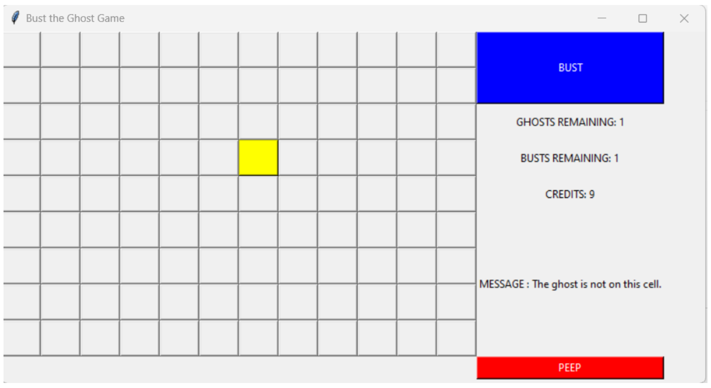
BUSTS REMAINING: 2

CREDITS: 0

GAME OVER

PEEP

When the cell busted doesn't contain the ghost:



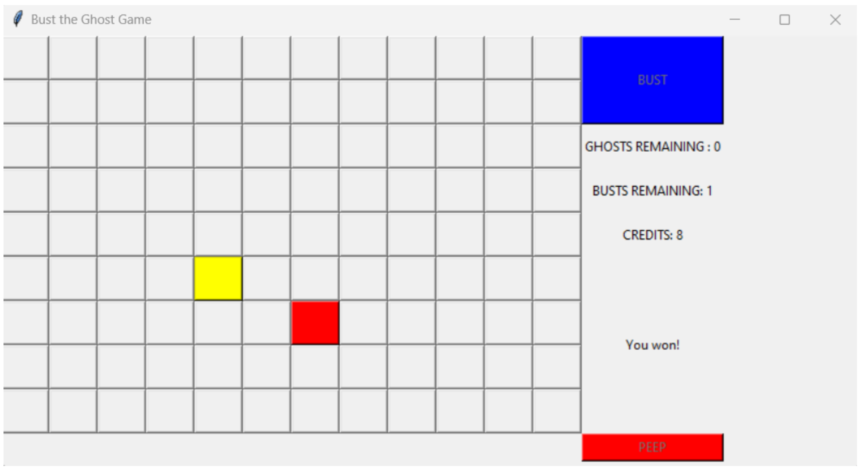
GHOSTS REMAINING: 1

BUSTS REMAINING: 1

CREDITS: 9

MESSAGE : The ghost is not on this cell.

When the cell busted contains the ghost:



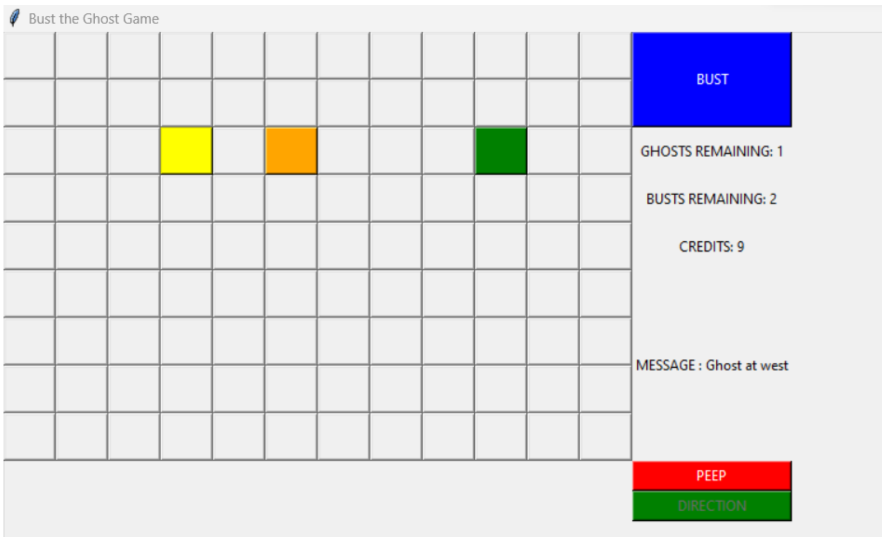
GHOSTS REMAINING : 0

BUSTS REMAINING: 1

CREDITS: 8

You won!

Bonus using the direction sensors:



GHOSTS REMAINING: 1

BUSTS REMAINING: 2

CREDITS: 9

MESSAGE : Ghost at west

Link to the video: https://youtu.be/RlYoMjM1ltw?si=NpG97621dU6MDX21

<u>VII. Conclusion</u>:

In summary, our project "Bust the Ghost" effectively utilizes Bayesian inferencing to enhance gameplay dynamics. Through the integration of probabilistic principles, players engage in an immersive experience where sensor readings inform strategic decisions. Key components such as updating probabilities and considering direction enrich the gameplay, offering both entertainment and educational value. Overall, this project effectively demonstrates how players learn to reason probabilistically by analyzing sensor readings and updating their beliefs about the ghost's location. With further development, the game could be expanded to include additional sensor types or more complex ghost movement patterns, creating a richer and more engaging experience.

References:

Taboga, Marco (2021). "Bayesian inference", Lectures on probability theory and mathematical statistics. Kindle Direct Publishing. Online appendix. https://www.statlect.com/fundamentals-of-statistics/Bayesian-inference.