WORCESTER POLYTECHNIC INSTITUTE

ROBOTICS ENGINEERING PROGRAM

Project 2 Report

SUBMITTED BY

Max Berman

Robert Gunduz

Aliaa Hussein

Date Submitted: 3/9/25

Course and Group: RBE4701 Group 1

Course Instructor: Professor Carlo Pinciroli

Scenario:

Similarly to the previous project, our goal in project 2 is to have our character navigate to the exit square of a predetermined map. The monster entities populating each of the five maps are the same, however with the caveat of there being no path open to the exit at the start of the round. In addition, we are required to implement a form of Q-Learning into our solution. This scenario, by the nature of the map's topography, will require the usage of bombs to clear, unlike in project 1.

Program structure:

To start our implementation, we have implemented a simple Q-Learning algorithm to drive the character. To be more specific, we implemented an Approximate Q-learning algorithm with two features. The first feature is reciprocal of the distance to the exit plus one (to avoid a divide by zero error). In an environment free from monsters, like scenario one, this is the sole feature driving Q-learning and ideally should have a positive weight. Second, there is the distance to the nearest monster entity, utilized in the same equation as the first feature. This should be utilized as a negative weight, reducing the Q-value of a given cell. Over time, the character should learn to avoid monsters as much as possible and, when an opportunity presents itself, proceed to the next wall barrier.

To handle opening new holes into walls, we implement a simple state machine. Most of the time, the character will move according to regular rules. However, if the system thinks the next good move is to a space with a wall, the state shifts and the character places a bomb and moves to the next valid space. To avoid bombs, the character keeps track of the bomb's timer and behaves normally. Once the timer reaches a defined threshold, the cells within the blast radius are no longer considered valid movement targets for the character and will force the character to move out of the way. This way, the only hazard feature we need to track will remain the monster's location. Another state includes simply A* which is switched to when there is an open path to the exit and the monster is not along the path.

To allow for the character to keep training beyond the scope of one run, the weights for the Q-learning algorithm are saved to and pulled from a .json file. This will also allow for us to have different weights for different variants, allowing the character to be less cautious of stupid monsters and more cautious of aggressive monsters.

Important Methods:

Q_value:

The focus of this project is to explore the implementation of Q learning. This method is meant to take the current world state as well as a pair of x and y coordinates to determine the Q value of a given coordinate. The equation used is a simple weighted sum of all feature equations. Originally, we implemented a great number of features recognized by the algorithm, however we have since simplified it to two with great results. The Weights are saved as a global variable within the character class.

Q_exit and Q_monster:

The first feature function, Q_exit, the method takes the world and (x, y) coordinates and returns: 1 / (distance to exit + 1). This became the basis of each distance feature function in our program. Q_monster takes in the same arguments as the previous method. If there are multiple monsters present, the function uses the closest monster to calculate this value.

Reward:

The reward method is meant to read the state of the game and return the appropriate reward for use in Q_value updating. By default, if no special conditions are met, the function returns a predefined transient reward. For reaching the exit, the method checks if the distance between the given coordinates and the exit is zero. Due to difficulties working with the code, the reward condition for encountering a monster is met when the distance between the character and monster is 1. While this may cause an issue in situations where the monster does not kill the character after this condition is met, it still allows the character to learn to avoid adjacency.

Q_update:

Once the next move has been made, the character needs to update the weights associated with each feature. To do this, we require the projection of the next world state as well as the Q values of every possible move in this new world state. The maximum value is taken from this set and discounted, then summed with the reward before subtracting the current Q value of the space. This delta value is multiplied by the learning rate and a weight's respective feature value before being summed with the weight value. When being trained this value is saved to a global variable and an external .json file.

xxx.Json:

To make allow Q learning to work better with difference variants and keep memory between iterations, we use Json files to save weights. In each variant file, a method is used to choose a file name unique to each scenario. In which the Q weights will be loaded and saved too. The Json file consists of a single array, which with the parameters described

earlier, must be at least 2 elements long. These files are accessed during the Q Update function, and upon initializing the file name parameter. There is also a dummy file which exists for character initialization or in the event a unique file is not chosen.

bomb_timer:

This returns the time left for the bomb to explode. This is used during pathing to check if the character has time to walk through future explosion cells.

explosion_locations:

This returns all the cells that explosions are currently in. This is not currently being used but was tested as a feature for Q-learning in the Q_explosions method.

danger_zone:

This method checks where a bomb is and projects the future exploded cells based on length. It returns true if the selected cell is a projected explosion space and when the timer is less than the bomb time threshold.

monster_along_path:

This method determines the direction of the exit and nearest monster and creates a condition to dictate whether the monster is along the path of not. The direction and magnitudes of the exit and monster paths is calculated using A*. The dot products are calculated and divided by the magnitudes which provides the cosine of the angle between them. If the cosine of the angle is less than 0, the monster is either behind the character or on the same level. This allows the character to head to the exit using A* assuming there is an opening to the exit. If the angle is greater than 0, it continues using Q-learning.

state_machine:

This is the main decision process for what the character should be doing. First the character checks to see if it needs to drop a bomb to get to the exit and switches to that state. The 3rd state checks to see if a monster is on the path which then heads to the exit or does Q-learning. If these states are both false, it defaults to Q-learning.

Experimental Evaluation:

Overall, the implementation of the q-learning algorithm achieves on average at least a 50% survival rate. As mentioned previously, different q-weights are used for each variant to improve the performance. For variant 1, 10 trials were conducted are there are no monsters in that scenario. For variants 2-5, 20 trials were conducted to evaluate the AI characters performance with q-learning.

Scenario	Trials	Survived	Survival Rate (%)
Variant 1: Alone in the world	10	10	100
Variant 2: Random monster	20	20	100
Variant 3: Self-preserving monster	20	16	80
Variant 4: Aggressive monster	20	12	60
Variant 5: Stupid & aggressive monsters together	20	10	50

Table 1: *Survival rate of AI character for variants 1-5*

variant1.py:

The character easily completes this variant with a survival rate of 100%. As there are no monsters, the performance of the character is essentially identical for each run.

variant2.py:

Similarly, the character easily completes this variant with a survival rate of 100%. With a smaller threat of the random monster, the character is able to successfully path find and make decisions.

variant3.py:

With the presence of a self-preserving monster, the survival rate decreases to 80%. The character tends to 'lure' the monster to an area near the top of the environment and go around the wall with holes that it has created in the walls to escape. However, occasionally the character can get stuck in a corner after it has placed a bomb.

variant4.py:

With the presence of an aggressive, the survival rate is further decreased to 60%. The character acts somewhat similarly to the pervious scenario where it tries to 'lure' the monster near the top of the environment. However, due to the higher detection range of the monster and its placement near the bottom of the environment, the character gets trapped more often.

variant5.py:

The presence of two monsters, a 'stupid' monster and an aggressive self-preserving one posed a challenge for the character as the survival rate hovers around 50%. This indicates the character's struggle with surviving multiple dangers. The character tends to either kill the 'stupid' monster of evade it, however, it struggles to evade the aggressive monster.

The limitations present in the code particularly highlighted variants 4 and 5, allow for potential improvements in the lack of balance in exploration and exploitation, optimal q-values and reward, and more features to accurately represent the different states.