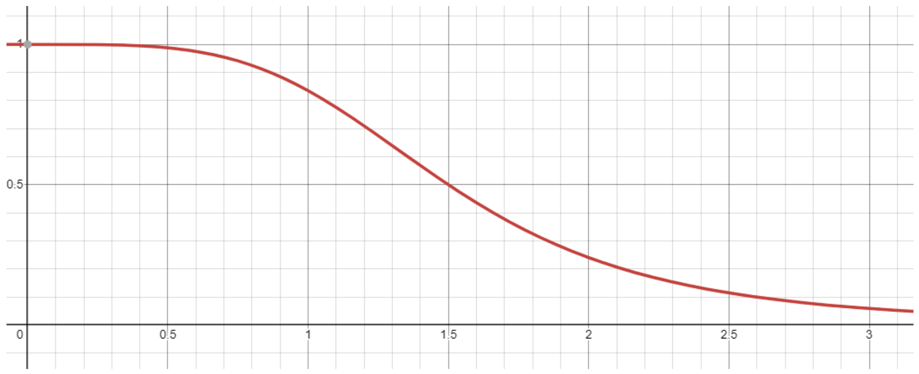
Project 2 Report

Structure of the approach

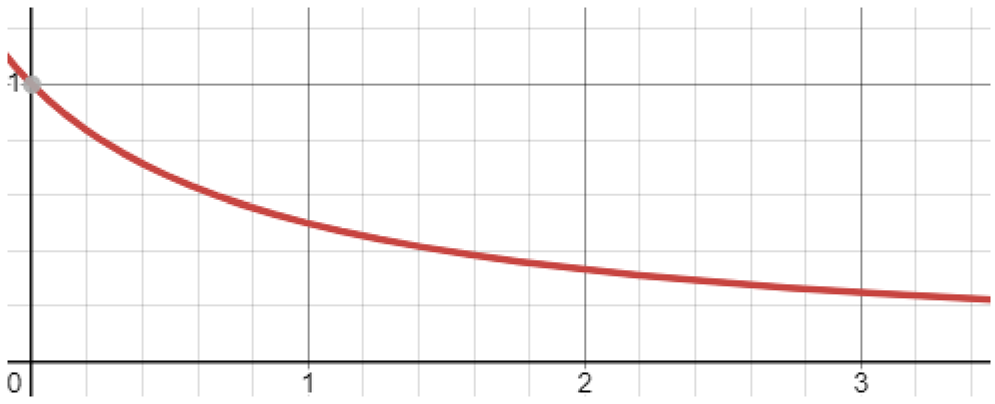
- The overall structure of this project consists of two states: a safe situation where no monster is present (WORLD_SAFE) and a dangerous state where the world contains bombs and/or monsters (WORLD_HAS_BOMB). The character initialization includes the name, avatar, starting x and y positions, and the variant number currently running for loading weights. The character then defines its state by checking its condition to determine whether the world contains a bomb, monster, or explosion. Another timer for the bomb is set to keep track of the world's bomb time, and if the world does not contain a bomb, then its value is set to infinity. The algorithm checks whether the character is close to the exit to take immediate action if it is near for the highest reward.
- The state machine comes into play. If the world is in a safe state (WORLD_SAFE), it proceeds to use the A* algorithm to find the best path to the exit. A weight is added to A* when calculating the cost for being near the wall for the next step to make the algorithm more aware of the surroundings. If the next step is not a wall, then the character takes the next action indicated by A*. However, if the world has an explosion in front of the character, it halts its next movement. Otherwise, it places a bomb at its current position with the customized bomb timer set to the explosion time given in map.txt. The world state is thus changed to WORLD_HAS_BOMB.
- If the world is in a dangerous state (WORLD_HAS_BOMB), the character's move follows the approximate Q-learning algorithm. It first reads the saved weights from the document, then passes them into the approximate_Q function. The returned weights are saved to the document for the next iteration.
- In approximate Q-learning, hyperparameters alpha (learning rate) and gamma (discount factor) are used with values of 0.01 and 0.9, respectively. The grid near the actions is given in a list to loop through for argmax(Q(s,a)). A reward dictionary, a Q-value dictionary, and a bomb information dictionary are created, with the tuple for movement as the key. For each of the possible actions, a new dummy world is generated to simulate the action. If the movement is within world bounds and not in the explosion range, a new character is generated. If the world is empty at its next target location, the character takes the move. Otherwise, it places a bomb if the world has no bomb and there's a wall next to it. The bomb dictionary is thus updated with a timer for the bomb as its value. Otherwise, the character does nothing. The reward for this step is extracted and added to 5000 to make it positive for the weight update, then the Q-value is calculated. Argmax is performed after iterating through all actions, and the bomb and action are copied to the real world. To find maxQ(s', a'), all possible actions are looped again with a newly generated world and character. The Q-values are calculated if the next move is valid, then the maximum is taken. At the end, delta is calculated with the equation [r + gamma*(max(Q(s', a'))] - Q(s, a), and weights are updated correspondingly with w_i - alpha*delta*f_i(s, a). Initially, I had 7 features, but eventually, extra ones were removed, leaving two: distance to exit and distance to the closest monster.

Interesting bit of your approach

- feature engineering - distance to exit
    - For the feature concerning distance to the exit, I originally used the Euclidean distance from the character's perspective. However, this approach resulted in volatile weight changes, leading to uncontrollable character behavior. Subsequently, I switched to the Manhattan distance, but encountered similar issues. Following some guidance, I normalized the distance using 1/(1+dist), but this often led the character to get stuck in corners, which are the worst positions for avoiding monsters. Considering the map's layout, I then applied a multiplier of 10 to the y-distance, ensuring that updates related to the y-distance were prioritized over those to the x-distance. This adjustment proved to be very effective.
- feature engineering - distance to closest monster
    - Regarding the feature for distance to the closest monster, I initially experimented with both Euclidean and Manhattan distances. However, these methods were ineffective, as they caused the character to be overly cautious around monsters from the outset. If the character did not start placing bombs, the reward would not increase significantly, hindering learning. I then normalized the distance, which yielded better results than previous methods but was still not entirely satisfactory. I found that maintaining a distance of 1 was too small and 2 was too large, so I designed a new function for this feature that follows this specific curve: feature =1/(1+(dist/1.5)**4)



original feature = 1/(1+dist)



- Feature removal
    - Originally, 7 features were introduced, including distance to the exit, distance to the closest monster, whether the character is in the explosion range, whether the character is in a corner, time left for the explosion, the number of monsters present in the world, and distance to the wall. However, the character seemed to be confused by extraneous information since it assumed that the features were independent, leading to uncertainty in decision-making. I wondered if this issue was related to the learning rate, but even with a very small learning rate, the character remained overly cautious about taking risks. I eventually realized that many of the features were too similar to each other, causing the root of the confusion. Ultimately, I reduced the number of features down to two and integrated the avoidance of explosions into the overall structure of the code to achieve the desired result.
- Bomb timer
    - Initially I attempted to use the given parameter for world.bomb_time for keeping track of when the bomb will explode to avoid either being too sensitive about the explosion or not caring about it. However, it seems like that is a constant and does not change even if the time is ticking. So I defined my own bomb timer. At the start of the code, I began by setting the value equal to infinity, so that nothing will change if timer = inf (no bomb), as math.inf - 1 = math.inf. Then, whenever the bomb is placed, the bomb timer is set based on the bomb_time of the world +1, and the timer begins to tick as time passes by.
- Dictionaries for q, reward, and bomb values
    - Initially, I planned to use a separate grid to track each value for Q, reward, and bomb time. However, since not all grids contain that information, and in fact, most grids do not, I opted to use dictionaries to keep track of the values. This approach helped avoid the inclusion of unnecessary information.
- Weighted a*

  - Originally, the A* algorithm from a previous assignment was used, but that version of A* did not account for the fact that walls induce a cost on the next movement if it is blocked, since placing a bomb is necessary to proceed. To address this issue, I added 10 to the cost if the next movement leads to a wall.
- Read and write weights
  - All weights used are extracted and saved to a document with the variant name for real-time monitoring of how the weights are changing. This approach greatly assisted with parameter tuning.

Results

random seed	variant 1	variant 2	variant 3	variant 4	variant 5
123	Success	Success	Fail	Success	Success
60	Success	Success	Success	Fail	Fail
223	Success	Success	Success	Success	Fail
12	Success	Success	Success	Fail	Success
323	Success	Success	Success	Fail	Success
567	Success	Success	Success	Fail	Success
43	Success	Success	Success	Success	Success
423	Success	Success	Fail	Fail	Success
57	Success	Success	Success	Success	Success
23	Success	Success	Fail	Success	Fail

Variant 1: 10/10
Variant 2: 10/10
Variant 3: 7/10
Variant 4: 5/10
Variant 5: 7/10