Lily Garfinkel, Madeline Halley, and Alex Simoneau

Project 2 Report

March 18, 2021

Beginning Thoughts

When we began to think about how we wanted to tackle the problem, we started with learning more about Q-learning in hopes of being able to implement it into our AI. Since it is model-free we thought that this would hopefully make it easier to train our agent. We wanted our agent to take in a state and evaluate the total reward of that state. The agent will then follow the calculated path unless it needs to "think" about what it's doing through Q-learning.

After one of the class lectures, we thought about looking into using a library for our agent. The professor had also mentioned that using the libraries is a lot of work and would most likely be more challenging with the timeframe we had to complete the project. Our group decided it would be better to learn these concepts on our own instead of using an already created framework.

Writing Our Code

We began our process by re-reading the readme file to make sure that each of us understood the different modes of the game and the score. We also went through the monsters file to read about how the aggressive, dumb, and self-preserving monsters operated. The character can move in any direction that is available and the goal is to get closer to or exit the game. Our group decided that the first thing we needed to figure out was the distance to the exit and the path to the exit. When thinking about this we initially thought we might be able to use expectimax similarly to how we had used it in ConnectN. The character would need to be able to determine future moves to be predictive, which led us to think we would need to have a recursive evaluation of the path. In our qcharacter file, we outlined a score function which we wanted to be based on the state, meaning that there would be a 1 in 8 probability of a specific state. In terms of non-threat states, we determined that the further a monster is, the better the character's chance is in terms of exiting the game. If a monster is close but the character defends itself with a bomb

and the monster explodes, it will need to divert from the path in order to not be killed. In terms of a threat state, if a monster or wall is in the way, we will again need to decide the character to figure out whether it should defend itself or divert from the path.

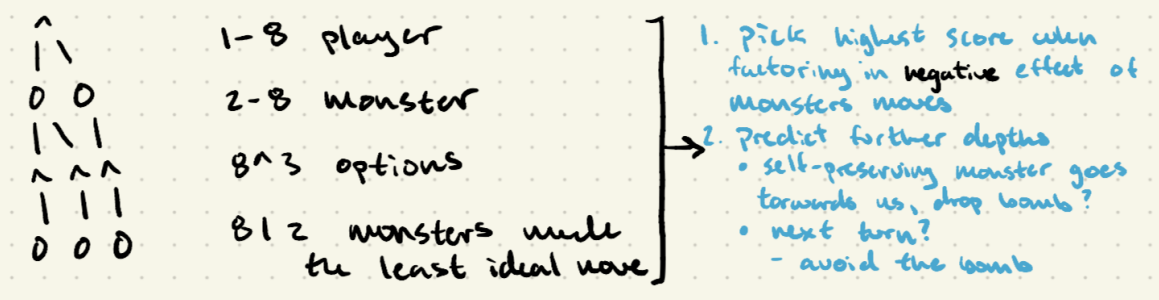

Figure 1: Thinking about probability of certain situations

In order to figure out the distance to monsters and the exit, our group decided to implement the A* algorithm. The information derived from the algorithm could then be applied to our character in order to determine different states. We created an A* file which contained two classes, the Node and Solver classes. We took the world functions that we could use to determine empty, exit and other types of cells and stored them as constants for clearer reference in the order from the readme. The solver.what_is() method is designed to find the "type" of such a cell.

```
class Node:
    EMPTY = 0
    EXIT = 1
    WALL = 2
    BOMB = 3
    EXPLOSION = 4
    MONSTER = 5
    CHARACTER = 6
```

Figure 2: Inhabitants of a cell assigned to a value within the Node class

After creating the Node class, we created the Solver class. The class contains helper methods and storage of world/path information on a per-path basis. We later realized that it is a

better solution to have a stateless solver that takes any world and two nodes as start and end points to determine the path. There are booleans to address wall clipping, ignoring monsters, and avoiding explosions on the path which are used to help get paths to different features of our Q approximation covered later.

```
def solve(self, wallClip=False) -> list[Node]:
    '''
    1 2 3
    4 x 5
    6 7 8
    '''
```

Figure 3: Grid to determine where characters next move should be

The implementation of these two classes helped us to determine the distances from our agent to bombs, monsters, and the exit. Once we had A* working with our agent, it was able to find the path to the exit, but when a monster appeared it was challenging to work with all possible states using expectimax. This is when we decided to switch to using linear Q approximation.

Our hope was once we began to train our agent, it would no longer get scared by a monster that appears on the board. The agent would be able to make a strategic move to stay out of the monsters range and get closer to the exit according to the training of its weights. When thinking about this process we drew out sample boards and began to think about possible states (See figure 4).

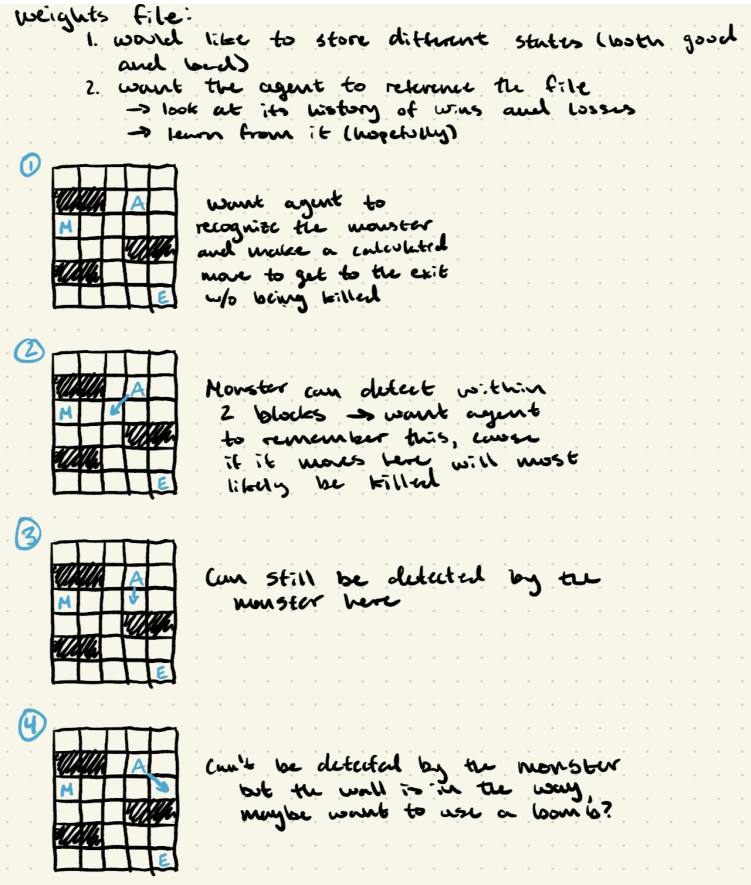

Figure 4: Visual for the the different decisions that we want to be stored in our file

We began thinking about the approximate Q function which would take in a world state and action as a tuple of dx/dy directions. With this in my mind we were thinking about the weights on the different features we wanted and how we would like to keep track of them as we trained our agent. We decided to use a text file to store our weights between runs. By doing this, it helped us to train the agent so we had the ability to see what needed to be improved as we trained.

To train the agent for both scenarios 1 and 2, we wrote the weights to text files so each variant could run concurrently. While training, it became clear that we were not properly considering enough features. At the time we only worried about distances but, per the suggestion of Professor Pinciroli, we included features that considered not just the distance but also the position of the character to the target in question (notably the closest monster and closest explosion). We added a feature that negatively considers the shortest x and y distances from the characters position to the nearest wall to prioritize centering in the middle of the board. We

added helper functions like find_things(), which identifies exit, bombs, monsters, and explosions. This cleaned up our implementation and allowed us to look further into the future to see if moves would put us in the path of a potential explosion with no way to dodge it.

As we trained our agent through each variant within the 2 scenarios, we began to find small bugs within parts of our code. We realized some of our A* class code was not calculating the paths right (which is why wall clipping and ignoring monsters became parameters). Once we found these errors, we were able to get through the first two variants of Scenario 1. Scenario 1 was completed after we were able to properly consider all features.

The bomb placing logic is based on the idea that if a monster gets close or the path to the exit is required to go through a wall, we can place a bomb at the time of this "realization". Placing a bomb changes the AI's behavior slightly and it will float around and often prioritizes the opening of the board especially in scenario 2. When the board is more open, the AI can get timid to push forward into another chamber to place a bomb. This scenario was our primary hurdle against the aggressive monster in s2v4 where the AI wouldn't put a bomb down in the gap between walls to open up the following chamber which contained the exit. This was not something we fully solved, but the AI does sometimes kill the aggressive monster and will make its way to the exit

A challenge we encountered was in adjusting weights. Even currently with the code as it is, the weights might not be calculated quite so cleanly. Essentially the program will save the Q value, action and various other parameters after it uses the Q approximation to make an action instead of following the path. When it does, it will remember to adjust the weights on the next call to do() since we need the "real" successor state rather than a simulated one. When we adjust the weights we will evaluate a Q approximation of the successor (so basically on the double successive state from when we first ran Q approximation). The whole process is quite messy but does follow the mathematical principles of Q approximation and weight correction. It often caused our weights to deviate significantly, so our final implementation is slightly adjusted weights but with minimal training overall to mitigate the negative impact of this bug.

What We Would Improve on

Throughout this project, we learned that our group needed to improve on working together and time management. We tried in the beginning to divide up pieces of the project so we could each tackle different components. As a result, not all of our code was in line with each other and chunks of it needed to be rewritten or scrapped. We wasted time and effort to blend it together for it to function in the way we wanted it to. This process would not have been so bad if we had given ourselves as much time as possible, but 2 weeks was not enough to go through those stages and end up with a dominant project.

Sources

Q-Learning

https://en.wikipedia.org/wiki/Q-learning
https://www.geeksforgeeks.org/q-learning-in-python/
https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/
https://www.freecodecamp.org/news/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe/

Bellman Equations

https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7

Reinforcement Learning

https://bair.berkeley.edu/blog/2019/12/12/mbpo/
https://stats.stackexchange.com/questions/221402/understanding-the-role-of-the-discount-factor-in-reinforcement-learning
https://www.youtube.com/watch?v=3zeg7H6cAJw
Building Game AI Using Machine Learning: Working with ...www.activestate.com › Blog
https://www.youtube.com/watch?v=WSW-5m8lRMs&t=385s
https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/
https://stackabuse.com/introduction-to-reinforcement-learning-with-python/
https://towardsdatascience.com/deep-reinforcement-learning-with-python-part-2-creating-training-the-rl-agent-using-deep-q-d8216e59cf31
https://medium.com/analytics-vidhya/the-epsilon-greedy-algorithm-for-reinforcement-learning-5fe6f96dc870

A* Algorithm

https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2
https://www.simplifiedpython.net/a-star-algorithm-python-tutorial/
https://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf