Irakli Grigolia, Sriranjani Kalimani, Emily Austin **Group13**

CS 4341 Project 2 Report

Introduction

Bomberman is a strategic, maze-based video game franchise originally developed by Hudson Soft and currently owned by Konami.The goal of this project was to code an AI capable of playing the game in 10 different situations encompassing varying degrees of difficulty.

Methods

Our strategy for this project was to begin with implementing A* and see how well it performed on different variants. From testing we discovered that A* would only have success rate on variant1, and occasionally on variant 2 and 3 from scenario1. In order to solve other variants, we attempted to implement different algorithms like expectimax and Q-learning.

A* Search

In order to find the shortest path we implemented A*. There are no major differences between our implementation and traditional A*. In addition we created some helper functions which are used in A* like distance(a,b) that returns absolute distance between tuples and eCell(node,wrld) that evaluates different cells and checks if it is safe to go there. A* is used in solving variant 1 of scenario 1.

Expecti-max

In the expectimax algorithm, we estimated the probabilistic model of how the monster would move in any state from the sensed world, we obtained utilities from the outcomes of current states of the world, and chose the action.This is done by going through all possible moves of the AI and the monster and giving the score/probability to each move. Our expectimax uses depth of 2 and has a helper function called eval(sens, wrld) that goes through the world and checks if there is any monster around us and returns a score based on the distance between AI and monster.

Also we use parameter focus when creating our AI. Higher focus means that the distance between AI and monster should be larger. In other words, Bomberman must stay away from the monster as long as it can, while lower focus means we can get closer to the monster. Focus is adjusted based on what kind of monster we are facing. If the monster is stupid we have focus set to 100, if it is aggresive monster then to 800, etc.

We also implemented a slightly different expectimax algorithm with a reward system. This system in theory should allow our agent to evaluate the risks and rewards of each move, which should increase the win rate. But after testing we discovered that it did not perform as good as we anticipated. Win rate was significantly lower compared to the first version expectimax. Nonetheless, there was one variant for which the win rate was slightly better for expectimax version 2 than version 1. For variant 5 in scenario 1 first version had a 50% win rate while the second one had 60%. Win rates for other variants can be found under the Results And Testing section.

Q-Learning

In order to solve difficult situations like variant 5, we attempted to implement Q-Learning. The objective was to train the Bomberman on a smaller map with both stupid and aggressive monsters in obstacle filled spaces. To do this, we implemented a basic implementation of Q-Learning.

The Q-table was an array containing states and action mapping. The states were identified with a helper function and appended to the array. Initial weights and rewards were set. The reward system penalized closer distance to the monster and/or walls, and positively rewarded proximity to the goal. The Q function used is as follows:

$$\mathbf{Q(s,a) = Q(s,a) + \alpha[r + \gamma * max_{a}'Q(s',a') - Q(s,a)]}$$

Although in theory this should enable Bomberman to survive difficult situations, in reality it did not. Attempts were made to tune the reward system and change α and γ.

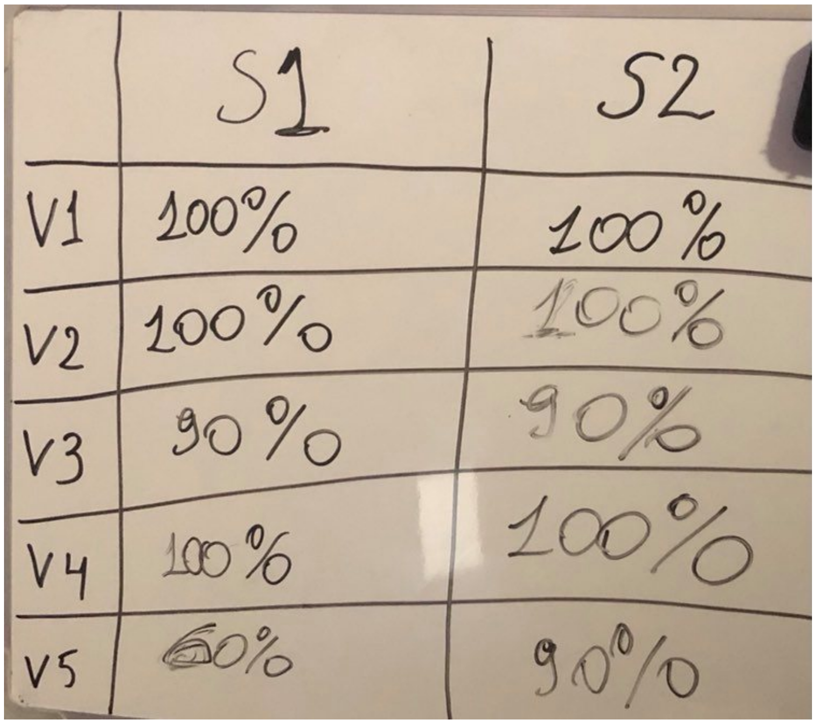
However, in testing, Bomberman performed worse with Q-Learning than the previously implemented Expecti-max. This led us to revert back and remove Q-Learning.

Results and Testing

Testing Procedure

We created a small script called test.py in order to test the consistency of our AI.Bassically, we ran different variants with different random seeds (from 0 to 10) and calculated the percentage of wins.

Win Rates and Consistency



	S1	S2
V1	100%	100%
V2	100%	100%
V3	90%	90%
V4	100%	100%
V5	60%	90%

Figure 1: Win rates by scenario and variant

As seen in Figure 1, the win rate for all variants except variant 5 in scenario 1 was 90% or above. We tried modifying our algorithms but were unable to get a higher win percentage for variant 5.

Failure Mode

We believe the reason our reward-based expectimax did not work is that it depends heavily on the monster's move. In the case of supid monster, Bomberman sometimes takes excessive waiting time to get closer to itself. In essence, the AI was extremely cautious and played very safe, and in some cases ran out of time and lost. This problem was addressed in our second version of expectimax which made our AI less cautious.

Another possible reason why it did not work well is that the reward system was not the best. It would work but sometimes would make unnecessary actions which made the AI vulnerable to bomb explosions and monsters.

Improvements

In order to improve our AI and get a better win rate, we would finish implementing our Q-learning algorithm or even implement approximate Q-learning based on the results we would get from basic Q-learning.

For minimax, one option is to increase search depth. The current depth is 2, but increasing it should in theory increase the win percentage. A better reward system could also fix the issues mentioned earlier. Finally, successful implementation of Q-Learning could have solved variant 5.

Another possible solution is to implement genetic algorithms. The python library NEAT(NeuroEvolution of Augmenting Topologies) enables the coding of different genomes and generations which can lead to the best AI for our project.

An example of how this could work is as follows. We would put our AI into the various situations with no prior knowledge of the world. We give it some inputs like distance to the bomb, distance to the wall etc. and get outputs such as move left, move right, place bomb, etc. We also assign fitness scores to each action, like a reward, killing the monster +10 to fitness, and so on. We run x different generations (e.g., 50) with 50 different agents in each of them. A new generation is built from the best agents of the previous generation and so on. In the end, we get the AI that knows the world well and can solve every level.

Conclusion

In conclusion, we were able to successfully win over 80% of the times when using the best algorithm for each scenario and variant. While we would have liked to implement more advanced algorithms given more time, we are satisfied with the results we got and enjoyed working on this project.