Tyler Reiser, Matthew Gulbin, Hunter Trautz

WPI - CS4341

18 March 2021

Intro to Artificial Intelligence Project 2 Writeup

Design:

The defining element of our agent is that it uses a finite state machine to traverse the world and deal with detected threats. Our state machine has 3 main states, which are “Walk”, “Bomb” and “Finish”. We implemented this using the python-statemachine module which comes with a very useful pre-built class structure to implement the basic requirements used in a state machine.This includes the states themselves, the ability to check the current state, and transitions between states.

The initial state in our Bomberman agent is the “Walk” state and it is also home to a majority of our detection code as well as the A* and Expectimax algorithms. While in this state, Bomberman determines the path to the next target in the queue using A* path planning. If there is a scenario where Bomberman cannot reach his next target, either blocked by a wall or potential explosion along the path, the goal is temporarily placed at the closest point along the traversable portion of the path. Bomberman then scans his surroundings and the goal is reinstated to its original position when a viable path becomes available. Once a path has been determined to Bomberman’s current goal, a check is run to determine if any monsters are within detection range along this path. Each type of monster has a customized range determined by monster name, which allows our model to be more accurate for risk assessment.

If Bomberman is not in danger from a monster, he follows the A* path that filters our walls and potential explosion spots. However, in the event that Bomberman is at risk of dying from a monster, we use an intelligent Expectimax implementation to calculate the most optimal dodging move to get away from the monster instead of just using A*. Combined, this allows for an adaptive system where Bomberman will prevent himself from walking into explosions, while also dodging monsters that are near Bomberman's current location during times when he is waiting for a viable path to open up.

The A* implementation is taken from Redblob, uses a priority queue, and keeps track of each path's cost so far. This cost is affected by three main things. The first is the length of the path, where the longer the path is, the less likely Bomberman is to take it. The second is the cost of walls, Bomberman needs to prioritize moving away from walls because he will be trapped and have less acceptable dodge moves. Bomberman also heavily prioritizes the third cost which is the cost of explosions. Obviously, there is the highest score as Bomberman needs to try and remain alive at all times.

Our expectimax implementation for determining the best move to make while in danger was the most difficult aspect of the project to implement. We begin the expectimax process by calling a method named initExpectimax, this method will return a tree that contains all possible future worlds with the potential moves for each monster within range in the world, as well as the path Bomberman took to get to his current position in the future worlds with an associated score for how good of a potential move that world represents. This is called recursively until the depth specified is reached. In our initial implementation, we created a new world for each possible monster move irrespective of the monsters actual name and type.

This turned out to be ineffective, not only because it dramatically increased run time, but also because it created a model that wasn't very accurate and caused Bomberman to run into monsters frequently. We changed this system to instead determine the potential moves a monster would make based on the name of the given monster. If it was specified as a stupid monster, we performed the typical expectimax that looked at any viable move a monster could make and averaged the scores assigned to that outcome. The difference in this method however, is how to determine moves for the other types of monsters. We took the detection code from the monster entity files and used that to determine which move the more difficult monsters would make. This approach is similar to minimax, as it assumes the monster will make the best possible move it can, and means that instead of potentially 8 worlds to explore, only 1 is needed which reduces the run time for tests by a large margin.

When initExpectimax is called for a chance node, or a node with no child worlds, we evaluate the world with an assigned score. If the world doesn't have a player character in it, we assign it a weight of -1000, and 1000 for a world where the player reaches the end goal. Additionally, we determine the distance to the next goal in the queue and decrease the score by 5 times the length of the path. This has the end result of rewarding worlds that put Bomberman closer to the exit. Additionally, we increase the score by 2 for every available neighbor that is available in the new world. This gives Bomberman an incentive to move towards positions that are more open and not walk into corners and die like he used to. Lastly, during score evaluation, the list of monsters in the world is iterated over, and for each monster, the path between the player and monster is determined, and if the length of the path is within a certain distance,a large score reduction is applied for the length of the path. This is the most important aspect of our expectimax, as it forces Bomberman to move to positions that keep him farther away from the monsters. Once our initExpectimax function finishes, we call our expectimax method. This method takes in the tree calculated in the initExpecitmax method and evaluates it to determine the best root move that Bomberman can make, using the pseudocode from the examples provided in class.

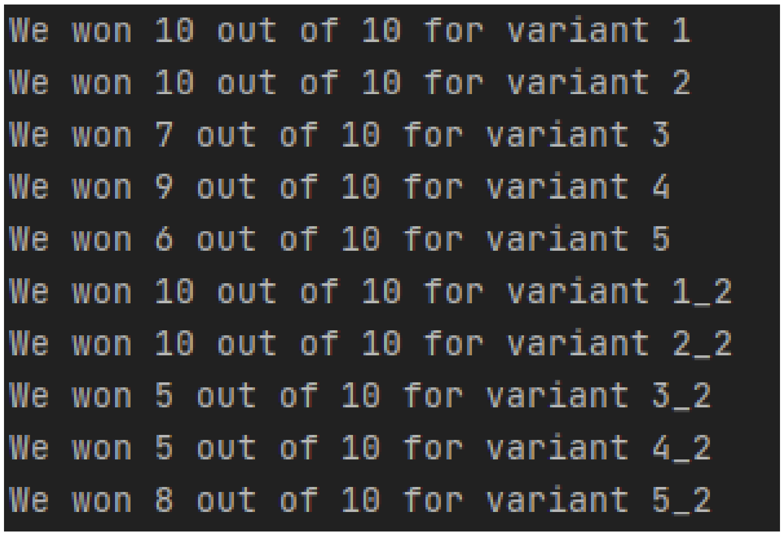
Bomberman is initialized with a list of preset goals, represented by a tuple of integers, that are specific for each level. These goals are essentially the most optimal coordinates on the level to place a bomb. Placing bombs at these locations achieves two main objectives. Firstly, monsters have an adverse reaction to nearby bombs and will actively try to dodge them, thus creating more distance between us and the monster. Secondly, there is a small chance that the Bomb will actually hit and kill the monster, rewarding us with points and ensuring that we will win the level. However, even if the bomb does not kill the monster, we know that it will at least break one piece of wall. This, again, creates more free spaces for us to actively dodge the monsters using Expectimax. In the first scenario, we do not use any goals because our testing showed that our default AI actually performed better than any iteration of goals we could up with for these levels. However in the second scenario, all of the five levels use the exact same map and we implement the same general strategy for each of them. This strategy is to place bombs on both the level and right side of each layer of the level to open up the most escape routes possible.

Agent Testing Procedure:

In order to test our implementation of Bomberman, the project team created a Python script that would run all ten variants a set number of times with different monster seeds and print the results. During our testing, we typically ran between ten and thirty instances of each variant. As mentioned earlier, Bomberman could be initialized with a list of board coordinates to prioritize placing bombs at while navigating towards the exit. This was an extremely invaluable feature to use while testing because the group could observe how Bomberman's expectimax implementation would behave when placed in different scenarios, such as being trapped in a corner or similar. The knowledge that the group gleaned from these tests helped further improve Bomberman's logic in both common and rare scenarios that it would encounter during play.

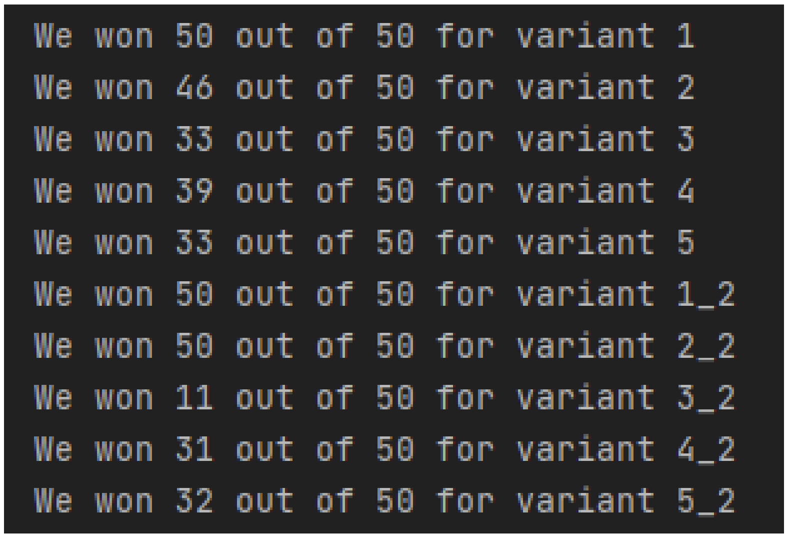
Agent Testing Results:

The following tests were performed by running each variant of both scenarios ten times:



```
We won 10 out of 10 for variant 1
We won 10 out of 10 for variant 2
We won 7 out of 10 for variant 3
We won 9 out of 10 for variant 4
We won 6 out of 10 for variant 5
We won 10 out of 10 for variant 1_2
We won 10 out of 10 for variant 2_2
We won 5 out of 10 for variant 3_2
We won 5 out of 10 for variant 4_2
We won 8 out of 10 for variant 5_2
```

This next set of tests were performed by running each of the scenarios fifty times rather than ten, to gain a better understanding of Bomberman's win/loss ratio.



```
We won 50 out of 50 for variant 1
We won 46 out of 50 for variant 2
We won 33 out of 50 for variant 3
We won 39 out of 50 for variant 4
We won 33 out of 50 for variant 5
We won 50 out of 50 for variant 1_2
We won 50 out of 50 for variant 2_2
We won 11 out of 50 for variant 3_2
We won 31 out of 50 for variant 4_2
We won 32 out of 50 for variant 5_2
```

One of the interesting test results that the project team took note of was the fact that Bomberman performed the worst against the self-preserving monster in variant 3. We found that this was due to the ability to place a bomb and “kite” or run away from the aggressive monsters while they chase you. This often results in monster kills, rewarding us with a satisfactory amount of points. The self-preserving monsters however have code to actively dodge bomb placements and will not just chase you blindly into a nearby explosion. This results in us often dying unless we are extremely lucky and the monster entity makes a mistake and is accidently killed by a bomb.

Future Improvements:

The project team believes that the bomberman agent could be improved by the implementation of some kind of data structure that could assist bomberman by storing his previous moves. This would allow the agent to get out of what we have come to recognize as stuck game states where bomberman has killed all the monsters on the level and cannot achieve his present goal due to a bomb missing the wall. Bomberman only has the ability to place a bomb when monsters or a goal is nearby. This results in him just standing still and never finishing the level. The implementation of this data structure would allow us to say, if bomberman hasn’t moved for a certain amount of moves, just place a bomb and navigate to the goal using A*.