Bomberman AI Project Design Document

By: Nathan Nguyen, Nathaniel Breiter, and Noble Kalish (Group 17)

Introduction:

To complete this assignment our team decided to do a modular state machine for each variant. While each scenario would be the same, each variant would be customized to deal with the specific challenges each one introduced. In general, our bomberman ran a similar code set. First, we would check if a direct route, i.e there was a route without having to blow up walls, to the exit cell. If such a route existed and no monsters were closer to the goal than we were, we would ignore all else and use that route. Second, we used A* accounting for monsters and walls to traverse the playing field otherwise. If A* determined that the next best move was through a wall, we would plant a bomb. Second, we used a modified expectimax algorithm if a monster was detected within a certain range. With a customized depth and heuristic we used this expectimax to either move away from the monster or plant a bomb. Our bomberman would continue to use expectimax until no more monsters were detected near themselves. In the next sections, we will explain our modified expectimax and A* algorithms as well as how we customized each variant to better win.

Expectimax:

We implemented a variable depth Expectimax to serve as the movement engine when close to monsters. How close the character can get before switching to expectimax changes depending on the exact variant being run, but it's always between 2-4 spaces of manhattan distance.

Possible moves are scored using a combination of board state elements. First a variety of possibilities are assigned static scores, such as reaching the goal, being eaten, and destroying a section of wall. Then the distance between the character and closest monster, as well as the character and goal, are taken into account. At first we tried to use A* to measure these distances, but we had to resort to manhattan distance to use expectimax at any depth greater than two.

Our implementation of expectimax uses a variety of modes to account for monster movement. When faced with a stupid monster, it treats each legal move as equally possible. However, it can also tell if a monster is self preserving or aggressive. For self preserving

monsters, the monster will look for any character within range 1 and attack. The same is true for aggressive monsters, except with a range of 2. This system helps the character stay out of kill zones and usually results in backing away until things are safe and the state machine can switch back to A*.

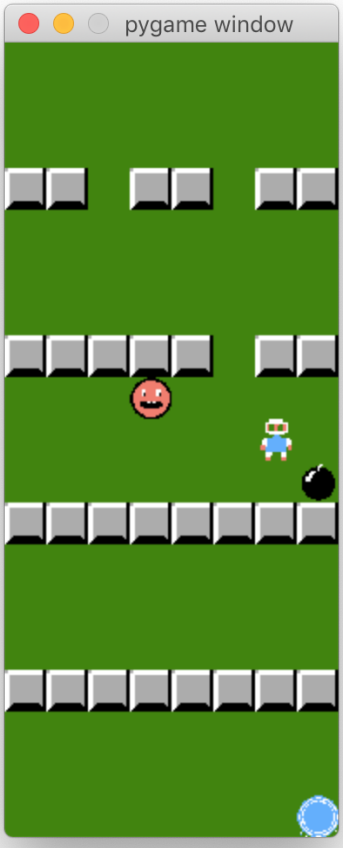


Figure 1:The exact point where Expectimax takes over. After the bomb is placed, the character will just stand still unless a monster approaches.

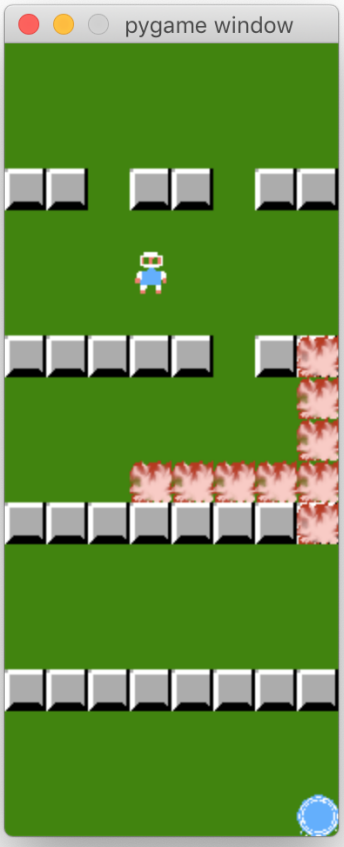


Figure 2: And here is the point where A takes control again, with the bomb gone and no monster in sight (no monster at all in this case).*

This expectimax will also avoid any bomb or explosion spaces. As such, it's use is continued past a monster leaving if there's a bomb on the play field. This avoids getting ourselves blown up. We've also observed this ability working to our advantage in testing when the character dodges an explosion while a pursuing monster is killed.

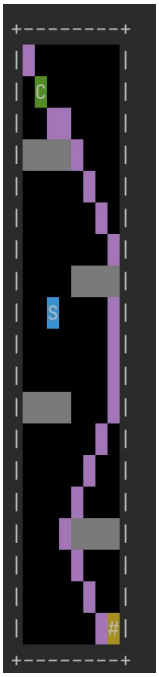
A*:

We implemented A* search both to use directly as a path for our character and as a tool for other logic. As such our implementation of A* takes in both the start and goal points as (x,y) tuples. This flexibility was handy when we added checking for a clear path to the goal, where we run A* from the monsters on the field to the goal in addition to finding the character's best path.

We also added an input variable that decides if walls are impassible or if they should be considered valid spaces. If A* ever tries to go through a wall, the state machine will take over, place a bomb, and then move out of the explosion and wait for the hole to appear. If walls are considered valid moves they are given a higher cost than normal spaces, as placing a bomb and waiting for it to explode takes longer than walking around to an existing hole.

Finally, we have a second input variable that decides if monsters should be avoided. When this is enabled, this assigns a heavy extra cost to spaces within 4 manhattan distance of a monster on the map, leading to the character hugging the wall farthest from the monster(s). This remained an option as some uses of A*, such as checking the real distance between the monster and character, don't require extra costs at all.

The main use of A* in our state machine is to control movement when not near monsters. The following screenshots show what those paths looked like with walls considered valid targets and attempting to ignore monsters.



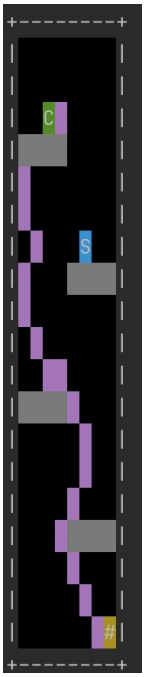


Figure 3: Initial A path avoiding the monster.*

Figure 4: Modified A path after the monster moved*

A* is also avoided when there is a bomb on the field in favor of sitting still or using expectimax. This fixed observed issues where A* would drive the character into a bomb explosion. When it is used, the A* path is always recalculated between moves rather than cached to account for new holes and moving monsters.

The other secondary use is running A* with walls or avoiding monsters to detect if the character has someone gotten past the monsters and has a clear shot at the goal. If this is true that path is just followed without any additional logic to prevent

Variant customization:

- **Variant 1:**

In scenario 1, our character uses A* algorithm to locate the goal. Since there is no monster in variant 1, the character can go to the goal directly. As such the A* used here treats walls as impassible and doesn't look for monsters. In scenario 2, A* is still used. However, the character will place a bomb down if it faces an obstacle.

- **Variant 2, 3, 4 and 5:**

For variant 2, 3, 4, and 5, the same methods were used with different parameters. First, the character verifies if there is a direct route towards the goal or not. If there is a direct route towards the goal, then the algorithm would perform A* to find and move the character directly to the goal. If there is a wall between the character and the goal, then the character would place a bomb next to the wall and move to a safe position. However, if there is a monster nearby when the bomb was planted, the algorithm will perform expectimax to move the character away from the monster without killing itself by the explosion. Then, the character checks for monsters nearby. For variant 2, because the monster moves randomly, the character checks for the monster 3 blocks away from the character, and 4 blocks away for variant 3,4, and 5. If there is a monster nearby, the algorithm would use expectimax to find the best move for the character. The probability value of the expectimax algorithm was calculated based on the distance between the monster and the character. Since we're assuming that the monster is always aggressive, the closer the monster is to the character, the higher the chance the monster would go in that direction. Therefore, the heuristic to calculate

the utility score for the expectimax function was tuned to effectively move the character away from the monster and closer to the exit.

Testing:

Once our state machine, A* movement engine, and expectimax movement engine were in place we designed a variety of testing tools to find bugs and tune values in A* and expectimax. As shown above, we used colorama to display the paths A* generated while we settled on working costs for walls and being near monsters. We also used a series of print statements that declared which movement method was used for the character at every step, which helped nail down issues caused by improper switching.

Later in testing, we also used a script that simulated the grading steps where each unique Scenario + Variant is run 10 times. The following is an example output of that script run on Scenario 2 when we decided we were comfortable submitting the results.

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

There were also a number of methods we tried but eventually deemed unworthy of further work when they didn't improve actual win rates. These included using Minimax rather than Expectimax with the aggressive monster, placing heuristic value on simply having a bomb in play, and smarter methods of detecting a direct path that took the proximity of monsters to the path into account.

Possible Improvements:

Given we weren't able to consistently complete the hardest scenarios there are many possible improvements we could have made. The first obvious choice would have been to implement Q learning. Having the time to implement Q learning or other reinforcement learning would have possibly made a larger impact on how well the code would have worked. We also wanted to implement a more intense bombing state. One that was more accurate and increased the number of bombs that we were placing. Bombs had the ability to avoid the

character from being trapped, but also caused issues with working around the bomb and avoiding future collisions.