Eric Arthur

Przemek Gardias

Daniel Sullivan

Group 01

2/28/2019

CS4341 Final Project – Bomberman

We have two different approaches to Bomberman depending on which scenario we are running. Scenario 1 variants 1-4 run on expectimax and A*, scenario 1 variant 5 and all of scenario two uses approximate Q learning. Below we have divided up the report into sections based on each scenario and individual variants if applicable.

Scenario One:

All of our solutions to scenario 1 variants 1-4 use very similar code. It is copied into three separate files as we change some slight parameters depending on the monster we are facing. Therefore, they all use the same functions so we will describe them in the order we developed them based on the need for them.

Functions shared amongst scenario one solutions that are trivial include:

Distance (x1,x2) – returns the direct distance between x1 and x2. This is not the Manhattan distance but simply takes the absolute value of the difference in their x and y coordinates.

A* - Used to calculate the shortest path to the goal but also used to calculate our distance from monsters. Since our distance function simply returns distance we use the length of the A* path to find the shortest Manhattan distance.

monsterDead – returns true if the monster is dead, false otherwise. This is used to see if we can just A* to the exit without having to worry about the monster.

Variant 1: Variant 1 simply runs A* to calculate the shortest path to the goal. Since there are no monsters we just move towards the goal. This is the foundation for all of our scenario one solutions.

Variant 2: To solve variant 2 we simply run A* until we come close to the monster. We check our distance to the monster by checking the length of the path returned by A* when we run A* on our current position and the most aggressive position from the monster. Below we explain what functions we use to calculate this but we use the function predictAggressiveMonsterMove to predict where the monster will go. If the length of the A* path between the monster and us is 2 or smaller we run expectimax. By running expectimax we are able to avoid the monster until it either passes or we can run by and after we have passed it we just run A* again until we win. By basing our expectimax off of the worst case scenario we are able to avoid the monster almost all of the time. This is also where we developed our bomb placement policy. It’s really relaxed and simply states that if we must move backwards to avoid the monster we also try to drop a bomb if the monster is close (within an A* distance of 2). If one is already out no big deal otherwise we drop one to try and kill the monster. Here we only run expectimax if the distance to monster is less than or equal to 3 from out current position because the monster is not smart so we don’t worry about it killing us until it gets very close.

Functions used in Variant 2-4:

Expectimax Discussion: Our expectimax runs with a depth of 4 but that is only two moves for us as we must simulate monster moves also. Therefore, we first generate all of our moves that we can take immediately, then the monsters and then recurse once more and do these again. By summing the values of the states that are now two moves ahead for both us and the monster we are able to avoid getting trapped or misplaying and moving towards the monster. To generate the monster move we use predictAggressiveMonsterMove which is discussed next.

PredictAgressiveMonsterMove – Runs A* on the monster's position and ours and return the best move the monster could make to threaten us. We use this to determine whether or not we are in danger of the monster approaching us. This makes our AI very cautious and sometimes he won't take a slight advantage as he believes it is too risky.

Variant 3: In trying to solve variant 3 we first started running into issues of our AI killing himself with the bomb. Because of this we developed functions that allow for us to see whether or not he will die in the future due to a bomb. Unfortunately, due to our limited understanding of the game code we are not sure this function behaves exactly as we'd like but it works well enough that bomberman wins 80% of the time on variant 3.

This is where we implemented our next strategy which is if the self-preserving or aggressive monster is chasing us we run back to spawn. This way we can almost always lose him on a corner or place a bomb and have him run into it. This works really well but occasionally runs into the issue of us running into our own explosion. To combat this we changed the distance we care about the monster to an A* distance of 3 which means the monster is currently 4 away. And we also changed the distance we drop bombs to be within A* distance of 2 from the monster. This prevents most case scenarios where we die to the bomb but for example if we drop a bomb and the monster runs away we sometimes just A* into the bomb.

We tried combatting this with a function cautionBomb() which was supposed to make bomberman run expectimax if a bomb is about to blow up to prevent A* from running us into it. This seems to work about 90% of the time but sometimes bomberman still runs in and dies. Due to limited time we weren't able to figure this out.

Variant 4: Variant 4 is where our AI starts to perform sub-optimally in certain situations. For example sometimes when the aggressive monsters waits in the last little area for him our bomberman will place a few bombs and wait but eventually gets caught beneath the explosion and is cornered with the monster. Because of this we invented a new A* that considers explosions and won't allow for him to move through them but instead path around them. This works to a certain extent but our heuristic for calculating utilities of non-terminal states started to be visibly wrong. The expectimax formula for calculating utility values looks like:

$$F(s) = 3 * distanceToMonster - 2 * distanceToExit - 1000\ * explosionNextTurn$$

Where the function values for distanceToMonster and distanceToExit are not normalized. We tried messing around with normalizing these values but the behavior of bomberman didn't seem to change very much. We tried making the weight the largest for explosionNextTurn in hopes that he wouldn't kill himself but he still does sometimes. On variant 4 we increase the distance to worry about a monster since the aggressive monster is intelligent. Here we worry when the monster is within a theorietical A* of 4 which means the monster is 5 away currently. This allows for bomberman to react accordingly to the aggressive monsters antics.

On top of this we tried to add conditional statements such that if bomberman is going to die using the move that expectimax returned try calculating another move. If the expectimax or A* move puts us in an explosion we try to catch it with an if statement in the greater do function and react accordingly. Turns out having conditional expectimax makes it near impossible to debug and because of this we could never figure out what was causing our guy to sometimes run into explosions. To combat this we our way of placing bombs. This makes it so bomberman drops less bombs and therefore is less likely to kill himself due to the decreased probability of explosion tiles. This works pretty well and bomberman is able to win over 80% of the time. There still are occasional seeds where the monster traps him in spawn or the corner but other than that he wins.

Variant 5: In order to beat variant 5 at a consistent rate, we decided that we would have to use q-learning. Our Expectimax function would not work, as our agent would sometimes get stuck up near the spawn, hiding from the first monster, and be unable to find its way out. When it was able to get out from the spawn area, it would get trapped between the two monsters and have a difficult time interpreting between the two monsters and would just avoid one monster into another monster. By performing q-learning, our agent had a much higher success rate.

The process by which we perform q-learning is explained in the scenario two section. By doing q-learning, we increase our rate of success on this variant, but it is still far from perfect. Our logic for placing bombs is heavily based on being near walls to prevent us from being trapped or if we are moving away from the exit. This is primarily because the agent is optimized for scenario 2, where it is trapped and does not have a lot of room to move, so he drops bombs often to open it up more. When it has more open space, it follows the same bomb placement logic, which ends up restricting his movement and gets him stuck sometimes. Sometimes our agent also freezes in the beginning when the dumb monster comes towards us, but the agent is normally able to make proper adjustments to get past this once he starts moving and bombing.

<u>**Scenario Two:**</u>

In order to solve scenario two in all the variants, we took the approach of q-learning. We used different weights in order to optimize our agent's play and to maximize the win percentage. To do this, we began with some basic weights in order to determine our agent's next move. These primary weights were based on distance to exit and if there was an explosion in the future.

Distance to exit was chosen in order to ensure that our agent would prefer to move towards the exit. Otherwise, the agent would just stand still and not have any incentive to progress towards the exit at all. By having this, our agent's Q-value is improved when he is closer to the exit. This is done in our function called distanceToExit, which takes a position and a world state. The distance to the exit is determined by the length of the path returned by running an A* algorithm from our current position to the exit cell. This length is then divided by the length of the distance to the exit from the origin by A* to determine how much closer we got from the move that was made. One is then subtracted by this in order to promote the agent to make the value higher for the closer he is to the exit, giving it a higher Q-value.

Having the ability to tell if there was an explosion in the future is important to have as a weight because it improves our rate of survival. Dropping bombs, especially when it is a necessity in the second scenario when our agent has no open path to the end, must become very frequent. With this, however, it makes certain moves result in the death of our agent when the bomb eventually explodes. In order to prevent this, we created a function called explosionNextTurn, which takes a move for our agent to perform and a world state. In this function, we simulate into two worlds ahead, as one world is not enough for our character to determine if an explosion occurs soon enough since explosions occur before it moves. By simulating two turns ahead. This gives our agent enough time to move out and ensure he will not move into an explosion and die. By using this function, we lower the q-value of any move that would result in the agent moving into an explosion.

Once we got these weights properly working, we added two more in order to improve performance of the agent. The first is distance to monster, and the second is recognizing corners. Once a monster was added into the variant, our agent would not perform well since he was only considering the path to the exit and explosions, not considering the monster which could kill him. In order to start avoiding the monster to survive, we created a weight for distance to monster. This is performed by the function distance_to_monster, which takes an action and a world state. If the monster is found, this function searches finds the A* and Manhattan distances from the new agent position to the closest monster's position. This distance is then averaged. If that distance is within a distance that we consider to be a threat to the agent, which is determined by max_distance_to_detect_monster, we return the average distance to the monster by the max_distance_to_detect_monster to lower the q-value of moves that move the agent towards the monster. If the monster is not within that distance, we return 1 to have no effect on the q-value. If the monster cannot be found, then we also return 1.

The recognizing corner weight came about when we started noticing that our agent, when avoiding monsters, would sometimes trap himself into a corner when running away and trying to avoid a monster. This would result in him having no moves to avoid the monster, eventually resulting in his death. To prevent this kind of behavior, we created the move_into_corner function, which takes a position. It returns a 1 if the agent is in a corner, .25 if the agent is in a pre-corner, and 0 otherwise. A corner is defined as an area on the map where the character only has 3 valid moves, and a pre-corner is any cell that surrounds a corner cell. These corners are determined once in the beginning, and update any time after an explosion clears, as the map may have been modified, possibly clearing corners that once existed. This is done in the update_corner_values function, which takes a world state.

Another important function that is used is predict_monster_aggressive_move, which takes a position and a world state. This is used for us to predict that the monster is going to move towards us at all times in order to play conservatively and maximize the probability of survival. This is used when check the next world state in order to determine our next move, so we are able to take the monster's position into account.

The weights are adjusted in the function gen_new_weights, which takes an action, q-value, a Boolean value that tells us if the game has ended, and a world state. This uses those arguments from the previous state and uses them to calculate the q-value of the new state. This is done by the approximate q-learning formula:

$$w_i \leftarrow w_i + \alpha * \Delta * f_i(s, a)$$

Where:

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

These weights all adjust and are updated at the end of a turn or once the game has ended. This allows the agent to adjust the way it plays based on every turn. In order to optimize the agent, we adjusted the weights accordingly for each variant in order to maximize the win percentage.

We currently have the α value, or learning rate, set to zero after we found a weight that would consistently win.

If we were to have more time, we would have also included a weight to reduce the chance that the agent would step into a new area with only one opening in scenario two. This is so he may maximize his opportunity to escape if being chased by a monster. We also would have improved our bomb placement policy. We would have made it in order to maximize blowing up walls through having some max weight based on the score for blowing up walls and such.

Variant 1: In this variant, all our agent primarily cares about is getting exit and not dying to an explosion in the process. Our weights reflect this by having the reward for distance to exit so high, making that the best move to make all the time.

Variant 2: Here our agent must find a way to avoid and survive the dumb monster. As soon as he kills the monster, he takes off directly for the exit. However, sometimes if our agent does not kill the monster, the agent will stand still until the monster comes near him and forces him to move towards the exit, and then he will start going towards the exit. This is reflected by the high value in the weight for this variant, as well as the large negative value for avoiding monsters. The weight of avoiding corners also increases, making the agent less scared of going into corners due to the behavior of the monster.

Variant 3: As the monster gets smarter, our need to avoid the monster gets higher as well. However, the weight of monster becomes much lower the more he plays, as his evasion to the monster is very good and does not die often to the monster in this variant. This is because the monster chases him more, and the agent ends up killing him more often then not, allowing him to then go directly to the exit.

Variant 4: With this smart monster at a lower level, we need to ensure that we are either a safe distance away from him or that we kill him prior to exiting the level. The way the agent plays is he often enters the second to last chamber with the monster and freezes or he will camp out the entrance. It then takes the monster coming at him to move him backwards so he places a bomb which can possibly kill the monster and open a way up to the exit. If he goes in just a little bit too far, he will get stuck with the monster and will trap himself in a corner. By decreasing the weight of the agent going into a corner, he normally avoids this move and will bait the monster out and kill him before getting to the exit.

Variant 5: Having two monsters in this situation does not have as much as an effect on us as they did in scenario 1 variant 5 since it is more difficult to converge on us. Our agent will go through his normal actions of getting into the new area with the first monster and perform his normal actions of avoiding, and either killing or getting into the next portion. From here, he will normally take the same actions he does on the second monster. Since these two are separated by a whole section, he can take normal actions and proceed through the variant as he does in previous variants. If his avoidance of monsters stays low (around -15 to -12), he should win. However, there are points in which where that number gets higher, causing him to take riskier moves or having him go into the monster.