CS 4341: Final Project Report

Isaac Woods, Ryan Racine, Aidan 0'Keefe

March 2, 2019

1 Group 09 - Final Project

Isaac Woods, Ryan Racine, Aidan O'Keefe,

1.1 Code Structure

Our final approach for this project was to use approximate Q-learning to solve each variant with a set of 12 features. The main function, `do`, decides which action to take by calling a function named `calcQ(world, action)`. calcQ can be passed an action or a null action (represented by the value -1). If a null action is passed, it evaluates the features in the current state, then evaluates state changes (which will be explained later). If calcQ receives a valid action as a parameter, it simulates the world with the given move, then calculates the new world's Q value. If the result is a terminal state, the reward from this state is returned.

When our agent is created, you can enable or disable features from the total Q-learning function to tailor the agent to the requirements of each variant. Any weights which are set to 0 when instantiating the agent represent the features that will be disabled during the default state. A 2nd state exists for the agent called the "exit rush" state, which the agent will enter when it has determined via A* that it is closer to the exit than any enemy present on the map is. In this case, all other features are disabled and the agent simply moves to the exit.

calcQ operates by calling a general function (`calcFeatureN()`) to calculate a given feature value. This function calls a set of individual calcFeature functions specific to each feature that will return the feature value. Below is a list of the features used and their function:

1.2 Features

- `feature0`: This feature is a dummy feature which represents the bias.
- `feature1`: This feature is based on the Manhattan distance to the exit.
- `feature2`: This feature is based on the Manhattan distance to a bomb in the current column or row.
- `feature3`: This feature is the number of neighboring walls.

- `feature4`: This feature is the distance from the closest side wall.
- `feature5`: This feature is the number of bombs on the field.
- `feature6`: This feature is our A* path finding algorithm.
- `feature7`: This feature is based on the closest enemy to the player at a given time.
- `feature8`: This feature is based on the closest enemy to the player within a range of 3 tiles.
- `feature9`: This feature detects corners.
- `feature10`: This feature looks for enemies within a Manhattan distance of 6.
- `feature11`: This feature detects enemies in immediate range of us.

1.2.1 Feature 1:

Calculates the manhattan distance to the exit. Used mostly in Scenario2 to encourage the bot to move towards the exit even when a viable path cannot be found

1.2.2 Feature 2:

Calculates the distance from a bomb in the same column or row within $\text{bomb}_{\text{range}}$. We used this mostly to tell the bot not to stand near bombs when they are about to explode. This at one point was creating invisible lines it would not cross even when chased by an aggressive enemy so we added a delay. This delay told the feature to ignore bombs whose timers were above a certain point (in the end 1 tick) because bombs are only harmful if you're too close when they go off.

1.2.3 Feature 3:

Counts the surrounding walls. The idea behind this was to give the bot some way of knowing it was in a tight spot or in a spot that would be good to place a bomb in. We didn't end up using this feature.

1.2.4 Feature 4:

Find distance to the closest side wall. This was used early on to tell the bot not to get to close to the side walls because these limit its movement options. This was eventually mostly replaced by feature 9.

1.2.5 Feature 5:

Calculates the number of bombs on the field. This was used to tell the bot if it should place bombs. The weight associated with this feature was eventually set to positive resulting in the bot placing bombs off cooldown

1.2.6 Feature 6:

Calculates the A* distance to the exit if a path exists. This was used as the main feature to drive the bot toward the goal, especially in the first variant.

1.2.7 Feature 7:

Calculates the A* distance to the closest enemy on the board. This was the first feature implemented to try to keep the bot away from the enemies. The bot doesnt care too much about enemies really far away which this still detected so the weight was kept low for most variants which favored the following features which limit the max range of detection

1.2.8 Feature 8:

It calculates the A* distance to the closest enemy same as feature 7 except it only considers enemies within 4 steps of A*. This is helpful because the enemy doesnt care about enemies too far away. It also lets the agent know of any eminent danger.

1.2.9 Feature 9:

It calculates a value that is supposed to represent how cornered the bot is. it calculates this using the distance from the closest obstacle on the left or right and the closest obstacle on the top or bottom. This was a useful feature for telling the bot not to put itself in situations that could easy cause it to be trapped and die.

1.2.10 Feature 10:

Calculates the manhattan distance from enemies in a 6x6 square around the bot. We discovered that if an enemy was moving towards the bot and a move would result in the bot being further away but the same number of A* steps the bot ust wouldn't move, so this feature helps the bot understand that moving further away is also helpful though this feature usually has a lower weight than its A* distance counterpart

1.2.11 Feature 11:

Calculates whether or not the bot is in the detection range of any enemies. This has a very high weight to ensure the bot doesnt step into detection range lightly as it often results in the bot dying. This was added to ensure the bot stayed out of these "No go zones"

After each feature has been evaluated, they are multiplied by the weights we have provided, giving us the final Q score for a given `Q(s, a)`. We choose the action with the highest score and take it.

1.3 Solution

Our Q-learning agent was reused for each variant with slight changes to which features were activated per variant. For instance, Variant 1 does not require any of the enemy features to be enabled, so they are left off to save on processing time. We added 3 things to the character's constructor: `active_features`, which sets the initial weights and determines which features have been enabled; decay, which is used during the training process; and lr, which is also used during the training process.

We first attempted to allow the agent to learn on its own. However, we found that the learning process was too slow to feasibly finish this project in time. As a result, we used a "human-in-the-loop" training process where the agent's progress would be observed, and weights would be adjusted by one of the group members to help speed up the learning process. This process took approximately 30 minutes to an hour per variant, with about 15-20 minutes added on to tweak which features would be enabled and disabled during A/B testing. The final weights for each variant included in our submission result in these results over 100 games:

1.4 Results

Key: Variant *: Num Wins:Num Games (win%)

- Scenario 1:
  - Variant 1: 100:100 (100%)
  - Variant 2: 99:100 (99%)
  - Variant 3: 93:99 (93.94%)
  - Variant 4: 80:100 (80%)
  - Variant 5: 85:100 (85%)
- Scenario 2:
  - Variant 1: 100:100 (100%)
  - Variant 2: 100:100 (100%)
  - Variant 3: 100:100 (100%)
  - Variant 4: 96:100 (96%)
  - Variant 5: 88:100 (88%)

1.5 Conclusions

Our agent performed very well across both scenarios in all variants, with a slight drop-off in performance in the more complex variants (Variant 4 and 5). Interestingly we found these variants easier to solve in scenario two. We have surpassed or matched the 80% threshold in each and every variant however and are satisfied with our performance. Approximate Q-learning based on toggleable features is a lightweight, easy to implement solution for a complex problem like Bomberman that has yielded fantastic results for us.

One interesting experiment would be to see, after extensive training, if the approximate Q-learning we implemented would converge on the same values we ended up with after our human-in-the-loop learning process. We believe that, while our weights are very good and win a significant portion of the time, there likely exist more optimal weights that could approach a 100% win percentage over the same 100 games per each scenario. The main limiting factor, and why we did not complete this during the project, was a lack of time to train the agent.