RBE470x Project 2 Report

By Josh Keselman, Sam Markwick, and Cam Wian

1. Overall Structure of Our Approach

Our character control code is contained within a file called `qlearningcharacter.py`. Our code has built primarily around approximate q-learning, with a few helper functions to assist in the learning process, A* navigation, and a state machine to control the character's actions.

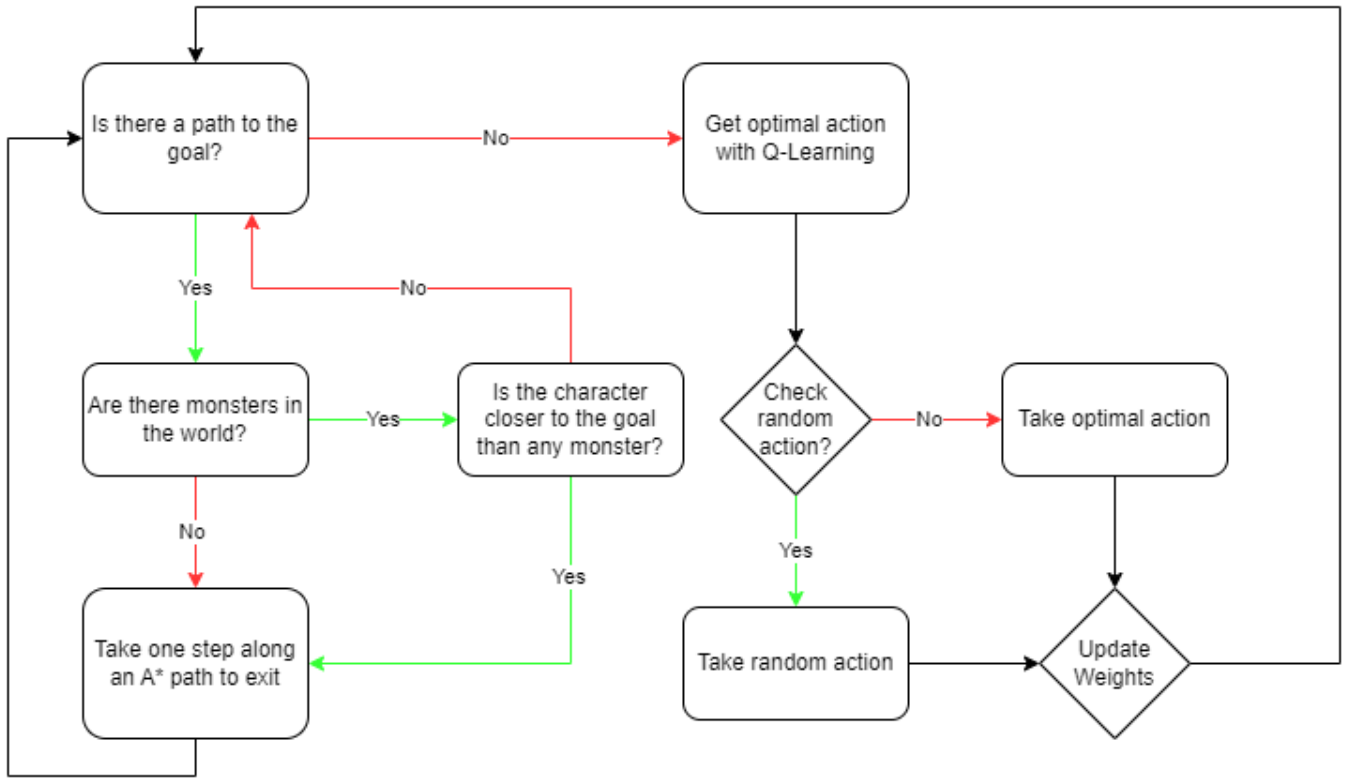
We utilized a state machine to allow our character to switch between an "exploration" state, where q-learning is used to navigate through the map while updating weights to optimize future decisions, and a "goal" state, where the character uses A* to navigate directly to the goal. The character starts in the "exploration" state, and switches to the "goal" state once there is a safe pathway to the goal.

2. Code Breakdown

This section will dive deeper into specific pieces of our code. We will briefly cover the structure of our state machine and our A* implementation, as these have remained largely unchanged from Project 1. We will then discuss our approximate q-learning implementation and the helper functions we used to assist in the learning process, as well as our training environment for q-learning.

2.1 State Machine

Like project 1, our state machine was implemented in the `do` function. The state machine is generally structured as follows:



This structure allows the character to navigate to the goal using A* if there is a clear path, and to use q-learning to navigate if there is not. It also uses a static epsilon value to allow for a mix of exploration and exploitation. This method, of balancing between direct pathfinding and q-learning, ensures that the character is extremely efficient on the easiest variants, as well as placing more weighting emphasis on actions taken further from the goal, where monsters and bombs are generally more threatening.

2.2 A* Implementation and Wavefront

Our A* implementation is primarily the same as our implementation from project 1, with few small change. First, we removed our added heuristics to avoid monsters and bombs automatically, as we want q-learning to able to adjust for these. We also added variables to allow for toggling on or off walls, bombs, and monsters. This allows us to use A* to both generate actions and find pure distances between points in the game world. We also included a toggle to switch between 4-connectivity and 8-connectivity actions.

2.3 Q-Learning

This section will be an overview of our q-learning implementation.

2.3.1 Features

We are using seven features for our q-learning: `EXIT_DISTANCE`, `IS_ALIVE`, `NUMBER_OF_WALLS`, `BOMB_PLACED`, `IN_BOMB_RANGE`, `DISTANCE_FROM_BOMB`, and `DISTANCE_FROM_MONSTER`. These features cover a wide range of features of the game state, with a focus on values that are directly relevant to the character's score. We created a method `featureValue` that takes the wrld and a feature and returns the value of that feature. For all features, values are normalized between 0 (for the least optimal case) and 1 (for the most optimal case). That method is outlined below:

```
if "EXIT_DISTANCE":
    get distance from character to goal
    return normalized distance to goal, using the initial distance to goal as the
maximum distance
if "BOMB_PLACED":
    return 1 there is a bomb in the world and 0 if not
if "IS_ALIVE":
    return 1 if character is alive and 0 if not
if "NUMBER_OF_WALLS":
    get number of walls in the world
    return normalized number of walls using the initial number of walls as the
maximum number of walls
if "IN_BOMB_RANGE":
    if there is a bomb in the world:
        get time to explode
        if time to explode is greater than 1:
            return normalized time to explode
    otherwise return 1
if "DISTANCE_FROM_BOMB":
    if there is an explosion in the world:
        get distance to explosion
        return normalized distance to closest point in explosion
    if there is a bomb in the world:
        get distance to bomb
        return normalized distance to bomb
    otherwise return 1
if "DISTANCE_FROM_MONSTER":
    if there are monsters in the world:
        get distance to closest monster
        return normalized distance to closest monster
    otherwise return 0
if feat_name=="DISTANCE_FROM_MONSTER":
    monsters = self.findMonsters(wrld)
    if monsters:
        dist = float('inf')
        for monster in monsters:
            thisdist = len(self.astar(wrld, pos, monster,throughWalls=False))
            if thisdist>0:
                dist = min(dist,thisdist)
        return 1-1/(dist)
    return 0
```

2.3.2 Main Q-Learning Method

We use a standard approximate q-learning procedure for our algorithm. This has threw core methods: `getDelta`, `updateWeights`, and `getQValue`. `getDelta` returns the difference between the reward and the expected reward, `updateWeights` updates the weights based on the delta, and `getQValue` returns the expected reward for a given action.

`getDelta` uses the formula [r + gamma * Q(s',a') - Q(s,a)] to calculate the difference between the reward and the expected reward. We are using a discount factor of 0.9, and we are using an estimate of the character's

score as the reward.

`updateWeights` uses the formula [w_i = w_i + alpha * -delta * f_i(s)] to update the weights based on the delta. We are using a learning rate of 0.05, low enough not to balloon our weights but high enough to incentive active learning of optimal weights.

`getQValue` returns the expected reward for a given action. This is calculated by taking the dot product of the weights and the feature values of the next world state. This is the core of our q-learning algorithm, as it allows us to estimate the expected reward for a given action based on the values of each feature in the state generated by that action, and to choose the action with the highest expected reward.

2.4 Training

We trained our character using a series of environments based on five provided variants: `training1.py` mirrors `variant1.py`, and so on. This allowed us to train our character to be successful at each variant specifically, as well as to give us a better way to debug our code, by showing different ways that each variant failed. For each of these trainings, we ran the training code 1000 times, updating the weights throughout, to get a finalized `weights.csv` file. For our testing and submission, we are using weights that were trained on the `training5.py` environment, as this was the most complicated environment, and we believe that it will be the most effective at handling the other variants.

3. Experimental Evaluation

During and after training the Q(a,s) function, notable emergent behaviors were observed. These include:

- Always having a bomb placed- the system learned that this maximizes the rewards given for exploding walls and monsters.
- Breaking through walls tends to happen on the edges of the field- this is inferred to be a result of avoiding the monster, as the edge of the field is often the farthest distance from the monster.
- Field "cycling"- due to bomb placement and then subsequent avoidance, the character often takes a cyclical path through the field as it places bombs and runs from them. This is comounded by the typical wall openings at the edges of the field and perpetual bomb placement.
- Monster migration- there is less incentive to, and a lack of a mechanism for, purposefully exploding a monster. This results in many cases where the character evades the monster until it makes its way to the top of the map, at which point the character can exit.

3.1 Performance over training

Variant	Survival Rate	Average Score
1	100%	4997
2	91%	4087
3	69%	2062
4	67%	1777
5	61%	1229