CS 4341 Project 2

Dan Duff, Blake Dobay, Andrew Levy

Objective:

The objective of this project was to create AI agents to play the game *Bomberman*. In the *Bomberman* games there are two different scenarios, each with variations of map layouts. In scenarios 1 and 2, the objective of the scenario is to exit the map at a given exit tile, while also trying to avoid death by monster.

Variants:

In the first scenario there are five different variants. The first variant the bomber man entity is alone on the map by itself. Variant two features both the bomberman and one monster. In variant two the monster moves randomly through the map. Variant three also has two entities: bomberman and a monster. In this variant the monster moves randomly until a bomberman entity is within 1 tile of itself; once this happens the monster moves to kill bomberman. Variant four is the same as variants two and three however the behavior of the monster is different. In this variant the monster takes an aggressive approach and moves to kill the bomberman entity. The last variant, variant five has three entities on the map; one bomberman entity and two monsters. One of the monsters behaves by moving randomly while the other acts aggressively. Similar to scenario one; scenario two has five variants each with a different behavior of monster.

Rewards:

The game has a few different rewards for the agent. First, the agent gets 1 point for each step that it is alive. For every wall tile that the agent destroys it gets 10 points. For each monster that the agent kills, it gets 50 points. For each other character killed the agent gets 100 extra points. Finally, if the agent escapes, it gets 2 times the time-based points (where the points are worth the time left).

First Steps:

The very first thing that our team tried was to get a pathfinding algorithm working. To do this, our team first tried to implement an a* algorithm. The algorithm treated every tile as a reward of one. For the heuristic function our team used the distance to the exit. The distance took into account both the X and the Y distance and tried to minimize both X and Y distance to the exit. However, by only using this heuristic the character would get stuck. This is because the agent would try to minimize the X distance by moving directly over the exit; then it would minimize the y distance by going down. Eventually our team's algorithm was fully implemented, and the agent could find the exit is no matter where the exit was placed on the map. Obviously, this approach limited the agent heavily as it only tried to find an exit. This left the agent completely helpless with a monster or in scenario number two.

In order to deal with any scenario, our team first tried Q learning. After implementing a Q learning algorithm, our team noticed a few flaws with the approach. First, Q learning was too slow; since weights had to be updated it took too long for the agent to figure out what to do. The other problem is that the agent could not tell what type of monster it was dealing it. After

trying these first two approaches, our team settled on using expectimax to create an agent that would get through all variants in all scenarios.

Expectimax Implementation:

The first and most obvious thing that the expectimax needed to do is maximize the score for the agent. For this, our team implemented a function called expectimax_event_value. This function takes four arguments: self, wrld, events, depth. Self is the entity used to call actions, wrld the world that the agent is in, event is a list of events in the world, and depth is an integer that is the depth that will be searched. The code snippet of the function can be found below.

```
# Bomberman agent wants to MAXIMIZE score
def expectimax_event_value(self, wrld, events, depth):
    # If win or death event occur, return
    for event in events:
        if event.tpe == event.BOMB_HIT_CHARACTER or event.tpe == event.CHARACTER_KILLED_BY_MONSTER:
            # bomberman died, return worst value
            return self.worst
        elif event.tpe == event.CHARACTER_FOUND_EXIT:
            # bomberman wins, return best value
            return self.best
    if depth >= self.max_depth:
        # If depth reached, return evaluation
        return self.evaluate(wrld)
    return self.expectimax_argmax(wrld, depth)[1]
```

As seen in the code the function loops through all the events given to the function. If the agent is hit by a bomb or the character is killed by the monster the worst reward is returned since these are worst case scenarios that we never want to happen. If the agent finds the exit, then we return the max value because this is the best-case scenario. If we have reached our search depth that has been passed in, then we go to the evaluation function to figure out what our value is.

The evaluate function that is used in the expectimax_event_value function is what is what the agent uses to get the value of a given world state. As such the function takes two arguments, self and world that represent the agent and world state respectively. The function starts by taking the next iteration of both the monsters and the agent. Next, the function loops through all the monsters in the next iteration of the world state. For each monster the function calculates a score based off of the distance of the monsters to the agent. The score calculation is shown before.

```
for monster in monsters:
    distx = abs(bomberman.x - monster.x)
    disty = abs(bomberman.y - monster.y)
    if distx <= 2 and disty <= 2:
        if distx <= 1 and disty <= 1:
            score -= 100000
        score -= 10000
    score -= self.monster_dist*100 / (distx + disty) ** 2
return score
```

First the x and y distance of the monster to the agent is calculated. If the monster is 2 away from the agent then a negative score is added to the total score for the world state. Then since it is very dangerous for the monster to be within one tile of the agent it is given a much greater negative score.

The main expectimax function that uses both the above helper functions is called expectimax_argmax that takes the agent, the world state and a search depth, then returns the coordinates of the next node that the agent should go to. The first thing that the function does is loop through all the possible agent actions shown in the code snippet below.

```
# look at all bomberman actions
for bomberman_dx in [-1, 0, 1]:
    # check bounds
    if (bomberman.x + bomberman_dx >= 0) and (bomberman.x + bomberman_dx < wrld.width()):
        for bomberman_dy in [-1, 0, 1]:
            # check bounds
            if (bomberman.y + bomberman_dy >= 0) and (bomberman.y + bomberman_dy < wrld.height()):
                # dont need to check moves that go into walls
                if not wrld.wall_at(bomberman.x + bomberman_dx, bomberman.y + bomberman_dy):
                    # Set move in wrld
                    bomberman.move(bomberman_dx, bomberman_dy)
                    if no_monster:
                        (new_wrld, new_events) = wrld.next()
                        dist_to_best = self.manhattan_dist((bomberman.x + bomberman_dx, bomberman.y + bomberman_dy), self.best_action)
                        expected_value = self.expectimax_event_value(new_wrld, new_events, depth + 1)
                        expected_value -= dist_to_best
                        if expected_value > maximum:
                            action = (bomberman_dx, bomberman_dy)
                            maximum = expected_value
                    else:
                        num_monster_options = 0
                        monster_action_sum = 0
```

The function first checks to see if the action is out of bounds as then it can be ignored. Then it moves the agent in the next iteration to that move. It then calculates what the value of that move would be and if it is greater than the maximum we have found so far, that is the new action to take. Next we do the same for all of the moves that the monster can do.

```
    # check all monster moves
    for monster_dx in [-1, 0, 1]:
        if (monster.x + monster_dx >= 0) and (monster.x + monster_dx < wrld.width()):
            for monster_dy in [-1, 0, 1]:
                if ((monster_dx != 0) or (monster_dy != 0)) and\
                        (monster.y + monster_dy >= 0) and\
                        (monster.y + monster_dy < wrld.height()):
                    # Remove moves that would go into walls
                    if not wrld.wall_at(monster.x + monster_dx, monster.y + monster_dy):
                        # Increase options and expectimax sum from state
                        num_monster_options += 1
                        monster_action_sum += self.simulate_monster_action(monster,
                                                                           monster_dx,
                                                                           monster_dy,
                                                                           wrld,
                                                                           depth)
    dist_to_best = self.manhattan_dist((bomberman.x + bomberman_dx, bomberman.y + bomberman_dy), self.best_action)
    expected_value = monster_action_sum / num_monster_options - dist_to_best
    # Update expected value if new max found
    if expected_value > maximum:
        action = (bomberman_dx, bomberman_dy)
        maximum = expected_value
```

Like with the agent moves the function loops through all possible moves that the monster can make first checking if they are valid moves. Then each move is evaluated and if the value is greater than any we have already found then we set that as the new action to take.

These are the main functions that are responsible for the expectimax algorithm. This algorithm allows for the agent to find the move that will maximize the action to take. When used in conjunction with the a* algorithm the agent can both fight monsters and find a path to the exit.

Putting it all together:

Now that the agent has both a pathfinding algorithm and a value maximization algorithm, the do function puts everything together. First the function uses a* to find the next node in the path to the exit. Then the function sets the next node in the path as the next node. After doing this the function places a bomb. The reason for doing this is it both maximizes the available routes and also kills monsters. The reason that it can just do this is because the expectimax will make sure that the agent avoids the explosion afterwards. The explosion will also give extra points for destroying walls and monsters as well. Finally, the do function moves to the node chosen by expectimax.

Debugging:

To debug what our agent was doing we resorted to coloring tiles. This method was particularly useful when debugging our a* pathfinding algorithm. Coloring the path that the agent has calculated using the algorithm allowed us to visualize what the ai was doing and follow along. This method of debugging is shown in the picture below. Our team also used print

statements throughout the code in order to print variables, statements and other things that allowed our team to gain insight into what the code was doing.