WPI CS:4341 (Introduction to Artificial Intelligence)

Group 28 (Nikolas Gamarra, Adam Moran, Ying Zhang)

Term Project: Bomberman AI

Project Strategy

Our overall strategy for this project was to begin with implementing A* to ensure we had a simple basic algorithm that could tackle most of the challenges. Once A* was implemented we set about improving our A* heuristic until we hit a limit of what we thought was achievable with A*. In order to try and solve some of the shortcomings of A*, we attempted to implement more advanced algorithms like expectimax and policy iteration.

A* Search Algorithm

In order to make our path we did an implementation of A*. There were no major differences between our implementation and traditional A* except that our priority queue was structured as a tuple of a location tuple and a priority. We also created a couple of simple helper functions to find the distance between two tuples, and check if a monster is within a certain radius of a cell in order to inform our heuristic. We also needed to implement a simple priority queue class. For our heuristic, normal cells were given a value of 1, walls were given a value proportional to the fuse time +20, cost of 60 was given to cells within a radius of 5 of a monster and a cost of 80 was given to cells within a radius of 2 of a monster. The A* algorithm returned a dictionary of the came from path. This dictionary was iterated backward into a list and then the next move was extracted. In the event that A* could not find a path (which happened when we were stuck in a corner between an explosion) a try-catch statement would prevent the program from dying and move the character to a random safe cell. Because walls had a very high cost, our agent generally avoids them but will path to them if there is no other option or if the other option gets too close to monsters. In the general motion of the character, a bomb is placed whenever a move is attempted into a wall.

In order to avoid pathing into cells that would be exploding in the next turn, several helper functions were added to keep track of where a bomb had been placed and how long till it exploded. If the character ever detects that it is attempting to move into a cell that will explode it will instead make a random safe choice.

A* 7 was made in an attempt to solve the issue of pathing into corners with avoid_monster3. Unfortunately, this implementation did not produce any better results for any variant than the other implementations we had.

Utility Functions

In order to tackle the problem, we created several helper functions that are universally helpful.

Place smart bombs and several other helper functions keep a record of which cells will be exploding and when. Using this A* can have the ability to peer into the future without the help of looking into future states of the map or using expectimax.

Using this we could simply and effectively create an A* solution that would never path into a cell that is about to explode.

Another important helper function we created for our A* was avoid_monster. This function was improved over three versions. In the event that a monster is found within a certain radius of the character A* stops and instead the character tries to run away from the monster. If the monster is in line with a potential bomb placement or within a certain radius a bomb will be placed. The avoid_monster2 behavior has a three part heuristic. First, the valid neighbors are considered. Of those, cells closer to the exit, further away from the monster, and bordering fewer walls score better. Avoid walls was added to attempt to avoid getting cornered. Ultimately We found that this was not enough to always avoid getting trapped in corners.

To solve getting cornered we created avoid_monster3. This function does a simple look into the future with sensed worlds and checks if it will survive running in for a certain depth in each direction if the monster is aggressive and chases it. The set of results where it survived is examined and the one furthest from the closest monster is chosen as the next move.

Expectimax

In expectimax search, we estimated the probabilistic model of how the monster will move in any state from the sensed world, we obtained utilities from the outcomes of the current states of the world, and emerge the action. The implement of the probability-based algorithm and reward system allows our agent to evaluate the risks and rewards of each move, which largely increased the pass rate.

The pseudocode of Expectimax we used is as follow:

```
def value(s):
        if s is a max node
                return maxValue(s)
        if s is an exp node
                return expValue(s)
        if s is a terminal node
                return evaluation(s)
def maxValue(s):
        values = [value(s') for s' in successors(s)]
                return max(values)
def expValue(s):
        values = [value(s') for s' in successors(s)]
        weights = [probability(s, s') for s' in successors(s)]
                return expectation(values, weights)
```

For future improvement, we would spend more time on developing the reward system. The current one is working but sometimes making unnecessary actions which made the agent vulnerable from the placed bomb and monsters. More importantly, we would increase the search level to go deeper for more accurate results.

Policy Iteration

In policy iteration, we basically wanted the agent to find the most beneficial actions in any given state. The idea was that we used a “SensedWorld” to modify the returned world by performing actions or change the state of the environment without affecting other existing world instances. With the exception of the agent occupying the same coordinates as the exit cell, all rewards would be negative to incentivize moving towards the exit. It used the same code used to avoid corners, calculate proximity to a monster, detect explosions before they happen, and calculate distances between two tuples to produce rewards. Unfortunately, how we would end up finding states would not include all of the possible states (especially since we would have to consider each combination of agent position, monster position, bomb position, explosion position, and wall position to have all the states and therefore the best action to do for each state) and would therefore create an incomplete policy, which defeats the purpose of doing policy iteration.

Solutions

In order to evaluate which test character we would use for each variant, we created a spreadsheet to track the performance of each one across multiple unseeded random trials. This way we could ensure we had the best performance possible for each variant and scenario. Below this table is a simple explanation of why we chose that algorithm and why it performed well. For an in-depth explanation of how the algorithm works look into the above sections on each algorithm.

		A*4(retreat)	A*5(normal)	A*6(cheese)	Expectimax 3	Expectimax 4	Policy Iteration	A*7(avoid3)	BEST
Total:		72.36%	74.27%	33.00%	64.00%	77.73%	0.00%	67.00%	81.36%
S1 ADV		66.0%	72.5%	66.0%	70.0%	76.0%	0.0%	74.0%	80.0%
S1	V1	100.0%	***100.0%***	100.0%	100.0%	100.0%	0.0%	100.0%	100.0%
S1	V2	90.0%	***100.0%***	100.0%	90.0%	90.0%	0.0%	100.0%	100.0%
S1	V3	50.0%	72.7%	80.0%	50.0%	***90.0%***	0.0%	70.0%	90.0%
S1	V4	40.0%	50.0%	50.0%	60.0%	***60.0%***	0.0%	50.0%	60.0%
S1	V5	***50.0%***	40.0%	0.0%	50.0%	40.0%	0.0%	50.0%	50.0%
S2 ADV		78.7%	76.0%	0.0%	58.0%	79.5%	0.0%	60.0%	82.7%
S2	V1	100.0%	***100.0%***	0.0%	100.0%	100.0%	0.0%	100.0%	100.0%
S2	V2	100.0%	***100.0%***	0.0%	90.0%	90.0%	0.0%	100.0%	100.0%
S2	V3	***75.0%***	66.7%	0.0%	40.0%	73.3%	0.0%	70.0%	75.0%
S2	V4	***68.6%***	66.7%	0.0%	50.0%	64.0%	0.0%	20.0%	68.6%
S2	V5	50.0%	46.7%	0.0%	10.0%	***70.0%***	0.0%	10.0%	70.0%

Scenario 1:

Variant 1:

We simply used our simplest version of A* (#5) for maps without monsters as it was reliable and effective. We chose to use the simplest versions for these as it would be less likely to run into errors.

Variant 2:

For scenario 1 we used our normal A* (#5) as it scored the best in our trials

Variant 3:

For this variant we used our implementation of Expectimax (#4) as is scored best in our trials.

Variant 4:

For this variant we used our implementation of Expectimax (#4) as is scored best in our trials.

Variant 5:

For scenario 1 we used our retreating A* (#4) as it scored the best in our trials

Scenario 2:

Variant 1:

We simply used our simplest version of A* (#5) for maps without monsters as it was reliable and effective. We chose to use the simplest versions for these as it would be less likely to run into errors.

Variant 2:

We again used A* (#5) for this variant as it consistently scored 100%

Variant 3:

For scenario 2 we used a special version of A* (#4) that would retreat to the y location of the start position. This behavior was implemented because we realized it was dangerous to enter the tight spaces while a monster was present.

Variant 4:

For scenario 2 we used a special version of A* (#4) that would retreat to the y location of the start position. This behavior was implemented because we realized it was dangerous to enter the tight spaces while a monster was present.

Variant 5:

For this variant we used our implementation of Expectimax (#4) as is scored best in our trials. This was because the different heuristic that was used for it performed better in variants with a lot of monsters as it moved more cautiously and placed more bombs.

Conclusion

In conclusion we were able to successfully exit the map over 80% of the time in our own testing when using the best algorithm for each scenario and variant. While we would have liked to implement some more advanced AIs given more time, we successfully met our target of being able to get a good grade based on the rubric in our own tests.