Joshua DeOliveira, Evan Hatton, Andrew Shanaj
3/18/21
CS 4341
Project 2 Report
Group 9

1. Introduction

The goal of the Bomberman project is to find the exit of the game in five variants with two different boards. The first of the boards is a maze-like setup with walls coming from both the right and left of the map spaced vertically by four spaces, while the second map was cut into five sections by walls stretching from left to right across the entire map. The variants are differentiated by the type of monster on the board trying to kill the character ranging from none at all to an aggressive and a random monster at the same time.

In the game of Bomberman the character can move to any of the eight cells that are directly adjacent to its current location. It can also place a bomb before it moves if it chooses to do so. Each character can only have one bomb on the board at a time. Bombs will explode after a number of turns decided in the map.txt file which is in ten in both scenario one and two. The explosion will go out from the bomb a set distance of four cells along the cardinal directions and last for two turns. If the explosion hits anything that is as far as it will go in that direction. Bombs can destroy walls and kill both characters and monsters. Due to the way the map is set up in scenario two bombs become necessary to get through the four sets of walls to the exit. However even when not necessary they can be useful; when all of the monsters in the game are killed the game reverts to a trivial A* search for a path to the exit. However, if bombs are to be used as a tool to kill monsters and not exclusively to break walls in scenario two we need to be careful that we do not kill ourselves with our own bombs.

The monsters in the game range from not existing at all to being aggressive. The first variant of the game has no monsters and just requires a basic path that will get the character to the exit without killing itself with its own bombs. The second variant has a stupid or random monster. This monster will randomly select its next location from all cells that are not out of bounds, or a wall. The third variant has a self preserving monster, this monster will pick a direction to move at random, and will continue to follow that direction until it either can no longer go that way, or would walk into an explosion. It will also move toward the character if it is in a cell adjacent to the monster. The fourth variant has the hardest monster to deal with the aggressive monster. This monster behaves just like the self preserving monster, however it will move toward a character that is within an 8-distance of 2. The final variant is the most difficult with both a stupid and aggressive monster. This version due to the starting location of the monsters could behave like variant 2 at the start when the character is close to the stupid monster, and then like variant 4 when the character is closer to the aggressive monster. If the game happens like this it should be easier to stay alive and reach the exit. Unlike if the aggressive monster were to work its way up the board while the stupid monster is still alive, meaning there are two monsters in a confined area around the character to deal with.

When initially looking for a strategy that will be successful against any monster the easiest thing to do is stay as far as you can from the monster while still eventually getting to the exit. However this can be difficult to do, as the character can get backed into a corner, on a wall or the edge of the map. In order to combat this if we try to stay toward the center of the map we are less likely to get stuck on an edge, and in at least the first scenario we will be far less likely to get stuck on a wall and die. To improve upon this strategy we can drop bombs in somewhat strategic locations to try and kill the monster. This does however add another level of danger for the character as it needs to avoid the explosion as well as the monster, but if and when the monster is killed the game is reduced to a simple search as stated above.

With all of this in mind when we were beginning to come up with ideas for how to solve this problem we thought of two drastically different ways to go about it. The first of which was to use an algorithm that incorporated a heuristic or some other way of deciding what the best move for the character is. The second was to use some reinforcement and deep learning method that would be able to solve all of the above variants of the game. Finally we could also use a state machine implementation that will determine its moves based on the world around it. We came to the conclusion that a reinforcement learning solution would be best, but that given the time constraints involved with the project it would not be wise to put all of our eggs in that basket, and also implemented a heuristic based solution and state machine solution.

2. Smart A*

The heuristic based solution that we used is heavily based on the A* algorithm as the name implies. In fact the first thing that this solution does when it is the characters turn is it creates a path to the exit using A*. If the exit can not be reached, due to walls blocking the way, a path is created to the location that is closest to the exit, and the string “bomb” is added to the end of the path. The A* uses a eight distance calculation to find how close the exit is. This distance is calculated by taking two points (x1, y1) and (x2,y2) and taking the maximum of the distance between x1 and x2 and the distance between y1 and y2. This in effect is the minimum number of turns that it will take the character to reach the exit.

With a path created on future turns the character will just follow this path. Moving in the direction of the first element in the path's list, because as the character moves along the path it removes the places it has been to, because it does not need to know them. When following the path and the first element of the path is the string "bomb" the character will drop a bomb and continue on its path. So far this is in no way smart and does not take into account any monsters.

In order to handle monsters the character will take what would be its next location, and look around itself with a radius of four for monsters. If a monster is found in that 9x9 square the character will find a new place to move to. It will find this new location by looking at the eight adjacent cells and finding those that are the furthest from the monster. From these cells that are the furthest from the monster the character will take the one that is the closest to the center of the map horizontally. This to prevent the monster from getting stuck in the corner between a wall and the edge of the map. Once the character knows where it will go next it will clear its path because it is no longer up to date, it will place a bomb and move to the new location that it just found. The bomb is placed here because there is a chance that the bomb will kill the monster that is close to us, and if it does the game turns into variant one and is easily won. Also the placing of bombs will destroy walls, which is not only rewarded with points, but also opens more pathways for the character to get to the exit. This is very beneficial in scenario two because the spaces are very confined and it is hard to run away from the aggressive monster.

This implementation successfully reached the exit 80% of the time for all variants, as can be seen below. The table is the number of times the character reached the exit out of 1000 games.

Table 1. Successful trials completed by Smart A* agent (n = 1000)

	Scenario 1	Scenario 2
Variant 1	1000	1000
Variant 2	988	987
Variant 3	919	904
Variant 4	904	837
Variant 5	822	848

This implementation of Smart A* was settled on after testing with a character search distance of three, which only reached the 80% threshold on variants 1-4 for scenario one, and only variants 1-3 for scenario two. For the other three variants the character reached the exit at least 70% of the time, however it was much better with a search distance of four, so that is what will be used.

3. Finite State Machine

The Finite state machine (FSM) approach involves the bomberman agent operating under behaviors (making decisions within a specific class in the environment) based upon a finite state machine where each state is a different agent behavior. At each time step, key factors in the environment along with the agent's current state are considered to activate possible state transitions, thus potentially changing the agent's current behavior. Under this system, the agent can take on 5 different states/behaviors: SNEAK, SURVIVE, BREAK, WAIT, and ESCAPE. When multiple state transitions become activated, the agent will transition to the state with the greatest priority. The priority of the states in decreasing order are as follows:

1. ESCAPE
2. SURVIVE
3. BREAK
4. WAIT
5. SNEAK

The ESCAPE behavior activates when the agent determines they can make it safely to the exit even if the monster operates under perfect play (including when there are no living monsters left in the world). Once in the ESCAPE state, the agent utilizes A* to follow the optimal path to the exit found. The ESCAPE behavior is the only final state in the FSM, thus when operating under ESCAPE behavior the agent will never leave ESCAPE behavior since ultimately the agent wants to exit the world safely and quickly as soon as possible. Since the entire FSM is designed to reduce the necessary time steps necessary to transition to the ESCAPE state, the agent will always prioritize transitioning to the ESCAPE behavior over all other behavior transitions when applicable.

The SURVIVE behavior can be activated when a monster comes within 2 eight-distances from the agent, and will transition to SURVIVE if no other available transitions have higher priority. The SURVIVE state stores the last non-survive state the agent was in before transitioning to SURVIVE since this state in essence acts as a subroutine behavior that activates whenever the agent is in immediate danger from dying by a monster. The agent will then attempt to evade the monster for as long as possible until the 2 eight-distance threshold has once again been upheld. At this point, the agent is considered "safe" and the agent will then revert back to the stored behavior state and continue the previous behavior per status quo.

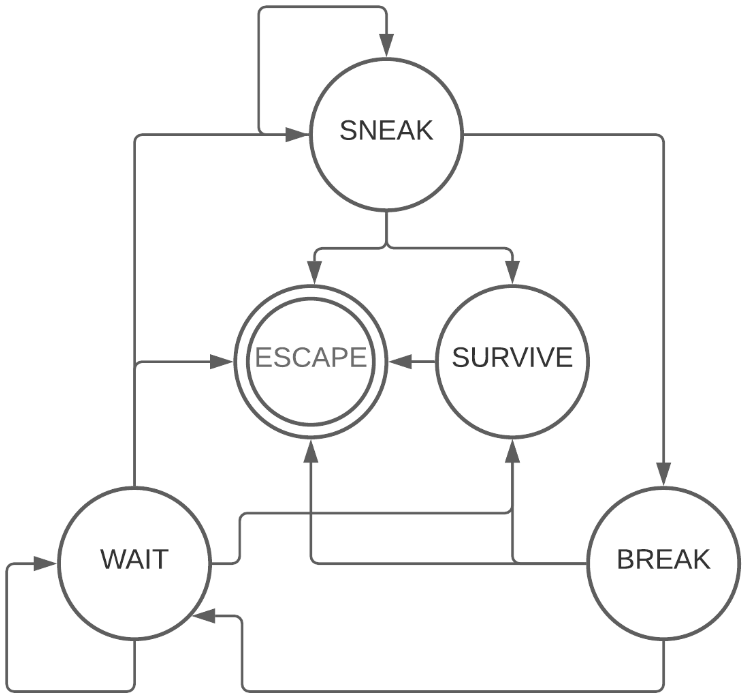

Figure 1. Finite state machine schema for FSM agent

The BREAK behavior can be activated when the agent has reached the pre-calculated bombing position from when the agent entered the SNEAK behavior. Once again the agent will only transition to BREAK if no other available transitions have higher priority. In this state, the agent will place a bomb at its current position and immediately transition to the WAIT behavior state on the same time step. WAIT starts a turn timer at 10 in order to wait the number of time steps until the bomb that the agent just placed explodes. WAIT's job is to wait for the bomb to explode and not put the agent in a position in which the agent will die from the bomb blast nor any living monsters. Also, once the bomb explodes it signals the agent that another bomb place is available, activating the transition to SNEAK behavior.

The SNEAK behavior activates when the agent is not in any immediate danger, does not have an immediately apparent opportunity to escape, and can place a bomb. When entering SNEAK from a non-SNEAK state, the agent locates a critical area to place nearby bombs to break nearby walls. By periodically dropping bombs as frequently as possible that will always explode away at least one wall tile, each bomb blast has a chance of killing monsters while simultaneously generating the new paths through the broken wall for out maneuvering any living monsters.

While intuitively this approach seemed to have promising performance in bomberman, this approach unfortunately has several shortcomings. Firstly, the agent has a difficult time determining good moves to position itself away from a bomb about to explode while trying to evade a monster. This is particularly apparent when facing against the aggressive monster in which a bomb blast doesn't kill the monster as the agent is cornered between the world's edge and a wall. Furthermore, the agent puts too much emphasis on trying to move to a position in which they can explode more walls. Even though in most situations the agent can just immediately run to the exit the agent is never incentivised to go closer to the exit until it is completely safe. Thus, the agent is too overly passive in its maneuvers and gets caught in overly compromising positions too often to be more successful than Smart A*.

Table 2. Successful trials completed by FSM agent (n = 10)

	Scenario 1	Scenario 2
Variant 1	10	10
Variant 2	9	4
Variant 3	9	7
Variant 4	6	0
Variant 5	0	0

4. Reinforcement Learning

Reinforcement Learning is the act of rewarding an agent based on an action the agent takes. On top of smart A*, our team went ahead and implemented Q learning, a reinforcement learning technique rewarding our agent's behavior. For policy we used an epsilon greedy approach. This means that we randomly pick an action with a probability of GAMMA. On top of that for our Q Table, we used a default dict with a lambda expression setting all 10 possible actions to 0. We also persistently save our Q Table as a CSV file. The most important part of Q Learning is the way we represent the state and and rewards.

For rewards, I decided that the way we wanted to set this up was as follows. Based on the manhattan distance, if our agent travels closer to the exit, we give him a plus one, if he moves further than, we subtract 1 from his reward. If we die in any way, we subtract 1000 from that action and if we win, we add 1000.

More important than the rewards is our state. To create the state, the agent himself has to track its location and actions throughout his journey. I edited the parents' place bomb and move function in order to keep track of our moves. From there, I am able to calculate my agent's state by checking his surroundings. My state is as follows. The first 9 features in my state are wall features, if a wall is 1 block near our agent or he is near the outer bounds, we get a 1 depending on the walls location. The next 9 features are features that would kill us excluding the monster. If the agent is a block away from the fire or the bomb is 3 steps within the explosion that bit turns into a 1, 0 otherwise. The next 3 features are representative of the monster. The first of those 3 is if a monster is present in 4 spots near us. The next 2 is the direction of the monster. The direction is either -1,1. -1 means the monster is on top for the 2nd feature, left for the third and 1 means down, right. Our next feature is if we have dropped the bomb. The idea for this feature is that our monster should drop the bomb, if he hasn't, and shouldn't if he does. The reward gives a plus 1 if he drops the bomb when he has a bomb to drop, otherwise -1. The last 2 features are the direction of the exit. They follow the idea of the direction of the monster.

As for the results, the Q Learning agent is able to do very well with the dumb monsters. He was able to learn his environment and navigate his way to the exit in both maps with no problem. His preferred maneuver is moving all the way to the right, dropping a bomb and then avoiding it.

One of the biggest problems that our agent faces is when encountering the smart monster. Due to the extensive amount of features, I have been unable to teach the agent effectively of avoiding the smart monster. This has come about the way I am handling the monster features, as well as not getting the event that a monster killed me, and only realizing that recently.

Deep Learning

After realizing that the monster features were inadequate, and I wasn't going to get far, and deep learning was the most recent topic in class, I decided to go ahead and try to implement a deep learning version for the agent. The idea was simple, screenshot the current state, grey scale it and transform it down, then run it in a neural net. If we win 1000 points, if we die -1000 points. When implementing this I made it as a script to quickly test. After running my script overnight, I woke up to 10000 runs, with the chance of winning a little over 50% of the time just for scenario1, variant1. I quickly learned and realized that Deep learning was going to take days to train.

5. Conclusion

With all of the different implementations as far along as they could be by the deadline we decided to settle on the Smart A* implementation as it was the most successful. This implementation does not handle two monsters as well as it does one monster because it does not look for more than one monster in its search to change course. This could be fixed by using a list of nearby monsters instead of the single tuple that it does.