Group 05: Final Project

By: Kyra Bresnahan, Clayton Dembski and Floris van Rossum

CS4341: Introduction to Artificial Intelligence | Professor Pinciroli

General Solutions and Testing

Listed below are the basic algorithms used to solve each variant alongside the fraction of wins we recorded in our testing. **Please note that we completed our testing with and wrote code for a previous version of Professor Pinciroli's Bomberman code. The last commit is 6237d5cb080e389e92839eef985cccf3897e2a4b (Committed on February 20th, 2019 with message "Fixed logic of keypress and drawing"). This was done based on an agreement with Professor Pinciroli who found we should be able to do this based on the issues with our old character code not working with the latest version of the Bomberman code. Due to the timing of the bomberman code bug fixes that broke our code, we were unable to modify our solution to be compatible with the latest version.**

Scenario 1		**Wins**
variant1.py	Greedy best-first search	1/1
variant2.py	Expectimax	50/50
variant3.py	Expectimax	10/10
variant4.py	Expectimax	38/50
variant5.py	Expectimax	7/12

Scenario 2		**Wins**
variant1.py	Greedy best-first search with bomb dropping	1/1
variant2.py	Expectimax with bomb dropping	10/10
variant3.py	Expectimax with bomb dropping	10/10
variant4.py	Expectimax with bomb dropping	16/20
variant5.py	Expectimax with bomb dropping	8/20

To test the code, we ran each variant in a loop with various different seeds. Each variant was run a certain number of times with a different random seed every time to produce different monster behavior. The code was tested on the given maps, as well as on mirrored maps. The folder "ScenarioWinPhotosFolder" in the group05 folder includes the results of the different trials listed above in the form of screenshots. The number of wins is under the word "EVENTS" in the console. The number of tests run is the range in the for loop in the editing window. To run our tests and see our results outside of the submitted code, clone or otherwise download the ToTestResults tag on our github page.

https://github.com/flipthedog/CS4341-projects/releases/tag/ToTestResults

An explanation of how each scenario was implemented is included in the "Final Solutions" section below.

Our Process

The first thing our team completed for this project is an implementation of the greedy best-first search algorithm which searches for the exit. The greedy BFS would move around walls that did not completely cut off the way to the exit. The idea behind this was that if there were no obstacles in the way, such as monsters or obstructing walls, this would be able to find the exit relatively easily. We completed this quickly, knocking out scenario 1, variant 1.

Then, we began work on an expectimax implementation to handle the variants with monsters to try to find a way around the monsters. We were able to get a solution that worked well for scenario 1 variants 2 and 3 as well as working somewhat for variants 4 and 5. We continued to improve this expectimax by adding in edits to optimize the code, such as making copies of only pertinent information and tightening up the code. In addition, we continued editing the heuristic for expectimax to try to improve the win percentage.

Next, we decided on how to handle dropping bombs. We determined that we would do this by dropping bombs whenever there was a wall that couldn't be passed. The next task we took on was creating a finite state machine so that we would only have to run large algorithms when it was necessary. For example, expectimax only kicks in if the monster is within a certain range; otherwise, the greedy best-first search algorithm runs since it is faster and finds the exit with a near optimal path, since it is not trying to move the character away from the monster. Next, we decided to edit bomb avoidance, since it was a hardcoded implementation where the character would just move diagonally up and left to avoid the bomb. The new bomb avoidance would no longer depend on a free space diagonally up and to the left, it would be able to look for available spaces to move to right before the bomb exploded.

The new bomb avoidance was working, and we were diligently improved our expectimax and its cost functions for the harder scenario 2 variants. This was the point when a new commit was pushed to the Bomberman code that broke our previous solutions (probably because our heuristic had been so fine-tuned). With little understanding of what changed, we went to talk to Professor Pinciroli on it and he was able to give us tips on how to improve the code we had. The professor pointed out that there were numerous shortcomings in our expectimax. We needed to re-order our max and expecting nodes, and adapt specific object references. However, even after implementing these edits, we were not able to get the code working as well as it had been previously. Our team stayed up two nights trying to figure out what was wrong with our code and why we couldn't get our functionality back after the new "bug-fix" commit. Even after talking to the professor again, we could not completely narrow down, why our previous implementation were no longer functional. We suspect it has something to do with the ordering of the monster and character moves.

Our team collected a bug bounty during that time, which we were really proud of, but only solved an unrelated problem of the monster and character getting stuck in our new expectimax. We went to

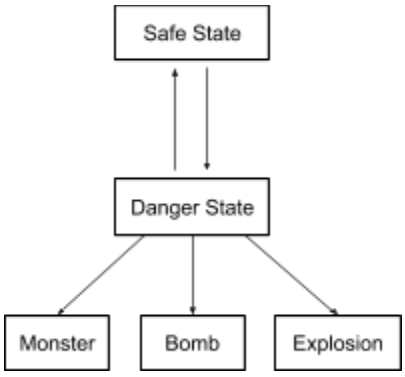
Professor Pinciroli again, and explained that we had made the fixes he had talked about and continued making bug fixes. Yet, we were still worried that we were not making enough progress. We talked through other possible fixes that could be made, but since there was no clear answer as to why our code no longer worked as well, Professor Pinciroli agreed to let us roll back the version of the Bomberman code we were using. We chose to roll back to commit 6237d5cb080e389e92839eef985cccf3897e2a4b.

After we did this, we were able to get back on track for getting, while not a perfect solution, a better solution. After this point, we continued to improve our heuristics and optimization of our old expectimax. We implemented a new bomb avoidance that used our expectimax code. In this phase of our work, we made sure we tested not only on the given map, but also a mirrored version of it, to make sure we weren't tuning our code just to the given map. Finally, we also completed an implementation of A* and a bomb avoidance algorithm based on the danger zone around the bomb, with hopes on combining them to make bomb dropping and avoidance more robust and intelligent. However, we did not have enough time to integrate this into our code.

We spent a large amount of time programming, refactoring, bug-fixing, editing and throwing away code on this project. We hope that is reflected in this report and on our GitHub. While we don't have a perfect score on all of the variants as we had hoped, we are proud of what we were able to accomplish after all of the issues we had and the amount of effort we put into the project. We very much enjoyed the project although it had numerous bumps along the way.

Final Solutions

As mentioned in the previous section, our process led us to the final solutions listed below. The final structure of most of the solutions was based on a state machine with greedy best first search and expectimax. The state machine would decide which state the character was in and toggle the appropriate algorithm. In the case of our program, we had a "Safe" state and a "Danger" state. The danger state was managed by a proximity check to the monster, bomb or explosion. If none of these objects were nearby, we were in the safe state. The safe state will trigger a greedy BFS search for the best path to the exit, the danger state will trigger expectimax to avoid the danger, while still moving toward the exit.



Another notable feat of the expectimax implementation is that we create our own objects for characters and monsters. This is done with the intention of saving the need to perform wrld.next() or copy wrld states, thereby optimizing the tree. This optimization meant that we can dive deeper into the expectimax tree, without sacrificing time needed to calculate moves. These objects carry specifications and logic that allows us to determine the behavior of the monster accurately. For example, when an aggressive monster spots the character, it will proceed directly to the character. This makes the expectimax more accurate as it eradicates the possibilities from the tree that are not going to happen.

Scenario 1 Variant 1

The solution to scenario 1 variant 1 is based on a 8-connected greedy best first search (GBFS) pathfinding algorithm. The 8-connected pathfinding algorithm would run first and check whether a path was available. As there are no monsters, this is a safe approach for this variant. This algorithm removes walls from the possible places it can go to. The heuristic is based on the manhattan distance to the exit from the possible cell.

Scenario 1 Variant 2 & 3

When within a given range of a monster, calculated by finding the maximum of the set {|Char.x-mon.x|,|Char.y-mon.y|} our expectimax function would trigger. The function, given a world, would first extract the character's position and the monsters' positions and ranges, adding them into a position tuple and a position & range tuple. This was done to limit the use of wrld.next(), as, when wlrd.next() used initially, at even a depth of 2 there was significant slowdown (at a depth of 3, with this initial version, we viewed slowdown of just over 5 seconds, and decided to rethink our approach.). We then call the expectation nodes and the max nodes. If, at any point before reaching the max allotted depth, the character has died, then the value of the branch would be penalized for ending and evaluated early. Otherwise it would be evaluated at the depth. The reward for distance to the exit is $-.5 * max\{|Char_x - Exit_x|, |Char_y - Exit_y|\}$. If within range of 1 or less of the monster, the reward is $-100^{1+(Depth_{Max} - Depth_{Current})}$. This penalized extremely the cost of dying early. If within a range of $2 + Range_{Monster}$, the cost is increased by $-5^{(2+Range_{Monster} - ManhattanDistToMonster)} - 1.5^{8-N_{CharActions}}$. This penalized being close to the monster, and trapping oneself into a wall.

Scenario 1 Variant 4 & 5

The expectimax solution for variants 4 and 5 is based on the expectimax version above. The cost function for variant 4, is identical to the one above. The cost function for variant 5 is just the sum of the cost of distance of the two monsters. The main thing that made the expectimax solution successful in these variants was accounting for the behavior of the aggressive monster. If the avatar of a monster is 'A', then the expectimax knew what move the monster would make in certain cases. For example, if the aggressive monster spotted you, the monster would directly approach you, and there were no other possibilities. The monster range was also updated so that the expectimax knew to stay at specific range to the monster to stay safe. This reduced the load of expectimax, and increased its accuracy. Finally, the depth of the expectimax tree was changed to 5, this was done to decrease the chance of the character getting caught.

Scenario 2

Expectimax in scenario 2 worked slightly differently to scenario 1. The first difference was in how it viewed the world. Given a true statement in addition to the previous, expectimax would tick forwards the sensed world without monsters and without characters. It would then tick to generate as many sensed worlds as the depth. This allowed us to examine a non character and non monster world at each step for bomb explosions, allowing us to get out of the bombs way. Once an explosion happened, the algorithms would treat the explosion, like they treat walls, as a location they cannot move into (i.e. not as a possible move).

The second major addition was due to the cost heuristic. This cost heuristic had a pull in the x direction opposite from the direction of the exit. This was what pushes it away from the bomb. The search and pull of the next location for bomb avoidance, however, also depends on the order of the traversal of the for loops used. Because of this, 4 new options were added in to allow for a more robust system: if the exit is flipped across the x axis and y axis, the character search order was reversed, if across only the Y axis, the row order is reversed, and if across the x axis, then both the row order and the search order are reversed.

Scenario 2 Variant 1

Scenario 2 Variant 1 used a modified version of Greedy BFS to identify when to drop a bomb. First, the standard 8-connected Greedy BFS used in Scenario 1 Variant 1 would be called. This function would return None if there was no path to the exit. In this case, when None is called, a second 4-connected GreedyBFS would be called. This second call would return the path, including walls. If a wall was reached along the path, a bomb would be dropped. The character would then avoid the bomb and wait till it explodes. The loop then continues, and the standard Greedy BFS will run unless no path is found.

Scenario 2 Variant 2 & 3

Scenario 2 variants 2 and 3 don't really cover any new ground. They integrate the Scenario 1 variants 2 and 3 expectimax with the Scenario 2 variant 1 bomb planting and avoidance. An addition was made to the expectimax to avoid explosion and bomb cells when in the expectimax tree. Before this addition, the character would run into explosions in an attempt to avoid a monster. This implementation allows the program to plant bombs while avoiding monsters

Scenario 2 Variant 4 & 5

As with Scenario 2 variants 2 and 3, Scenario 2 variants 4 and 5, don't cover much new ground. A difference between it and the scenario 2 variants 2 and 3 solution is that the expectimax runs at a deeper depth of 5, and that the monster range is set to 4. Although the actual aggressive monster range is not reflected in this number, this assists expectimax in avoiding the monster.

Discussion

The main success in our project originated from expectimax. Both before and after the Bomberman commit that broke our previous solutions, we had success with expectimax. The expectimax implementation, especially when optimized with the aggressive monster logic, worked exceedingly well. Greedy BFS also ended up being an algorithm that was crucial in our final solution. It was used in both scenarios in order to go to the exit whenever there was no danger nearby.

The main challenge in this project arose from debugging the Bomberman and expectimax code. Especially when handling the "bug-fix" commit that broke our previous implementation, finding the origin of the problem was difficult. Even while using debuggers in PyCharm, it was very difficult to figure out what was going on in the Bomberman code, let alone the expectimax tree. We attempted to manually calculate out our expectimax tree results in order to verify its correct functionality. Even when we thought expectimax was working well, we would double back and find a few edge cases where it would act strangely. Finding all the small bugs in expectimax in addition to the Bomberman code was a real challenge.

Possible Improvements

If we were able to continue this project, we have some improvements we would make to our code. We were able to get pretty close to the 80% threshold on variants 4 and 5 for scenario 1, as well as just hitting the 80% threshold for scenario 2 variant 4, so with improvements, we probably would have been able to solve those variants. In addition, we were very far away from the threshold for scenario 2 variant 5, so improvements would have vastly improved the effectiveness of our solution for this variant.

One improvement that we had been talking about throughout the paper is that we recognize that our bomb dropping and avoidance algorithms are simplistic. We have done tests for our algorithms on completely blank maps and have verified that our algorithms work better as the map gets sparser. Therefore, we believe that we could improve our solutions greatly if we had an algorithm that dropped bombs more often than just when there is definitively a wall in the way. One solution to this we had come up with was any time the monster was a certain distance away, we would drop a bomb. Therefore, there would be a clearer path for the character overall. We were also speaking about coming up with a heuristic function for dropping bombs that took into account the amount of space the monster was away as well as the amount of space to the exit, in addition to other factors. In addition, as mentioned in the process section, we were working on creating a bomb avoidance algorithm that would work based on calculations regarding the amount of time left before the bomb exploded and the 'danger zone' around the bomb. A bomb avoidance algorithm like this would probably be more translatable to any map configuration as well as being more of a guarantee in general that our character wouldn't explode.

Another improvement we have considered is integrating A* into our solutions. We have developed a working implementation of A*, but we ran out of time to integrate it. We were going to use it to replace greedy best-first search in our state machine. We were also planning on using it to find the distance to the monster, for example, in a more robust bomb dropping algorithm. Basically, there are a few places where A* could improve discovering the best move to make.

Finally, there is also more work that can be done in improving the heuristic and reorganizing the code to make it cleaner.