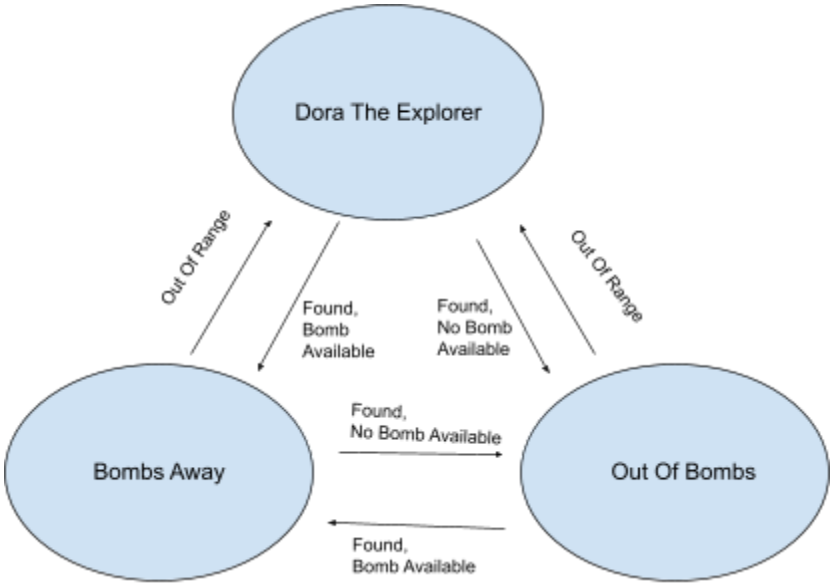
Project 1
Team 09
Mason Figler, Luke Hoy, Martin Bleakley

Approach:

The sections of code we wrote for this project primarily applied to the testcharacter.py file where we wrote the logic for pathfinding, dropping bombs, and evading the monsters. The high-level logic was contained in the character's do() function which runs in the main loop. Inside this function, we placed conditionals that would form our state machine and decide the behavior of the character. To find the optimal path to the exit cell we implemented the A star search algorithm using slides from lecture. In general, the character will follow this path unless it sees a monster in which case it will drop a bomb and evade using expectimax. It will then resume A*. If It encounters a monster and does not have a bomb to drop, it will just evade. Below we describe some of the key elements in our code and our evaluation of the character on all five variants.

State Machine:

Our character's state machine had 3 states. This was contained in our character's do() function where a conditional would first check if the character had been found by a monster. If the monster had found the character it would enter one of two states: fleeing with a bomb and fleeing without a bomb. Both of these are necessary as you cannot plant a bomb at every step, even if you would want to. The default when we are found is to drop a bomb and move away from the monster. If we are found and have dropped a bomb recently enough that we cannot drop another we enter a fleeing state which uses a version of expectimax to escape the monster. If we have not been found by a monster we simply use the A* search algorithm and Chebyshev distance heuristic to try and reach the exit cell.



A Star:

Our implementation of A* creates an easily traversable path from our start point to our goal. Similar to any other A* search algorithm, we are able to find the most efficient path to our exit point. One major aspect of our algorithm that makes it distinctly unique from others is that it utilizes the Chebyshev distance formula to help determine which nodes are more desirable than the others. We calculate our initial A* path at the very beginning of our program and start following it immediately upon its completion. We will continue to follow this path up until we detect a monster within our range. At that point, we move on to the remaining methods outlined in the rest of our report.

Expectimax:

An additional algorithm we implemented in our code was the expectimax away function. The main idea behind using our expectimax function was to use the probability of our possible moves and the monster's possible moves. The expectimax function would only run if we locate a monster within a specified range. This expectimax function will return the best possible move that the character could make to stay alive. The functionality of the implementation of our expectimax works by first determining each possible move we can complete. This list of possible moves is used in tandem with a list of possible monster moves with a depth of 3. To calculate the values of the leaf

nodes, we used the change in Chebyshev distance from the character's possible move and the monster's initial position with the Chebyshev distance of the character's possible move and the monster's possible move. Once all of these values are calculated for the leaf nodes, we calculate the values of the chance nodes by multiplying the leaf node values with their probability. These probability values are based on the number length of possible character moves. Finally, the possible character moves are iterated through, and choose the best move based on the calculated chances node values.

Bombs Away:

One interesting method we implemented in our code was the 'bombs away' method. This strategy allowed us to create a new path to travel around any potential threats and had the bonus effect of killing the monsters. As we followed our path created through the A* method, we actively looked for monsters in a range of three around us. Once we encountered a monster within the specified range, we then changed our approach from following our current path to following the 'bombs away' method. To begin, we place a bomb at our current position, regardless of where we are or where the monster currently is. Next, we analyze our surroundings to look for the best possible escape route to avoid getting caught in the blast. This takes into account any walls or borders surrounding us, as well as the anticipated blast-zone. Once we decide on our path, we begin moving along it. This typically will take us in the opposite direction of the monster through on expectimax search where we tend to make very cautious moves. The moment the explosion disperses, we return to our initial search with a new path given our new state. This might take us through a hole we just created in a wall, or it may leave us on a map with no monsters remaining alive. In either case, this significantly increases our odds of reaching the end goal unscathed.

Traversable Walls:

Another method we incorporated into our strategy was the ability to pathfind through walls. This seems like an approach that would result in failed pathfinding, but we use it in such a way that

allows us to escape from sticky situations. When we search for walkable cells, we look for those that are empty, have an exit, and in this case, those that are walls. This is important for specific situations where we are not yet in range of the monster, but it is blocking our path substantially. In order to execute this properly, we gave traveling through a wall a much higher cost than traveling elsewhere. This cost was calculated by counting how many steps it would take to place a bomb, move out of the blast-zone, wait for the explosion to disperse, and then continue along our path. If during this routine we find a better path because of the monster's movement, then we will take the newly discovered path. This is especially useful for situations when there is a monster clearly blocking our path up ahead and we are up against the border of the map and adjacent to a wall. In this case, rather than traveling all along the length of the wall and putting ourselves at risk of running into the monster, we instead keep a safe distance from the monster and save time by quickly bombing our way into a safer area. The only downside to this method shows up very rarely. In certain situations, we might feel the need to blast through the wall in a disadvantageous position, setting us up to be killed by the aggressive monster.

Evaluation:

To evaluate our progress we ran 100 tests per variant and documented the results in the table below. As the data shows, the variants became increasingly harder with the addition of both stupid and self-preserving monsters. Variant 1 had no losses as there were no monsters and no chance of not getting to the exit cell before time ran out. Variant 2 had only one stupid monster added to the map and in 100 tests it only managed to kill our character one time. On the third variant, a more intelligent monster with a relatively limited range replaced the stupid monster and proved to be much more effective against our character, scoring 13 kills and reducing our win rate to 87%. For variant 4 the same monster had its range increased from 1 to 2 and gained a less significant 2 kills over its predecessor. Our worst variant, variant 5, however, still recorded a 75% win ratio which is 25% better than the project requirements. Overall our character is significantly effective at evading or otherwise killing monsters and reaching the exit cell in time. Finally, to

further explore our successful algorithm, we implemented variant 6 which includes 6 stupid monsters and was still able to exit 74% of the time.

Variant	**Stupid Monsters**	**Self-Preserving Monsters**	**Wins (exited)**	**Losses (failed to exit)**	**Win %**
1	0	0	100	0	**100%**
2	1	0	99	1	**99%**
3	0	1	87	13	**87%**
4	0	1	85	15	**85%**
5	1	1	75	25	**75%**
6 (meme)	**6**	0	74	26	**74%**