AI For Robotics Project 1

Team 8: Alex Strickland, Declan Williams

Approach Structure:

The overall code is structured in the form of a state machine consisting of 4 states that acts depending on the state of the character. Every time the do function is run the state selector function checks multiple variables such as the bomb timer, if the character is in proximity to a monster, monster type, and if path planning is needed.

Whenever the bomb timer is not at its initialized value then a bomb is on the map. In this case the minimax state is entered because if a bomb is on the map that means a monster is very close. When a monster is close enough to drop a bomb it proved optimal to use minimax to prevent death no matter what type the monster was.

If the monster is in proximity of depth 3 or less then expectimax or minimax are run depending on proximity distance and monster type. If it is a stupid monster meaning random movement then expectimax is used. If it is another type of monster or the monster is 2 manhattan distance away or less, then minimax is used. Minimax is used in this situation because it is the only state that ever uses the bomb. Therefore minimax takes priority over expectimax when it is either an aggressive monster, or the monster is close enough to have a good chance at using the bomb effectively.

When path planning is triggered it goes into a state that runs A* to return the best path. Then it checks path length to ensure the path exists in case a bomb explosion blocked all paths to the exit. If the path does exist then it makes the first move on it before exiting the state. If no path exists then it simply waits until the next turn to check again hoping the bomb explosion has timed out.

Ultimately the strategy for bomberman was to follow the A* path until a monster was encountered. Using expectimax around stupid monsters, and minimax to kill or unblock a path for the aggressive monsters. If a stupid monster got too close to bomberman, then a fail safe would cause him to be more pessimistic and use minimax on a dumb monster.

Individual Approaches:

Getting Possible Moves:

A function to return possible moves proved very useful when determining options for both the character, and the monsters. Every time the best character move or the most likely monster move was determined, the possible moves function was used. The function worked by returning a list of cells depending on if it was for a character or a monster. If it was the character the list would contain the empty cells and the exit, while for the monster the list contained all cells not containing a wall.

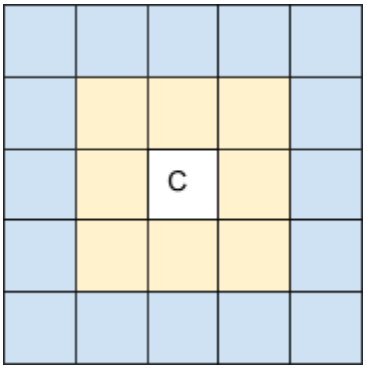
A*:

The A* algorithm works like the pseudocode given in class with the g(n) being the Manhattan distance and the heuristic h(n) being the euclidean distance.

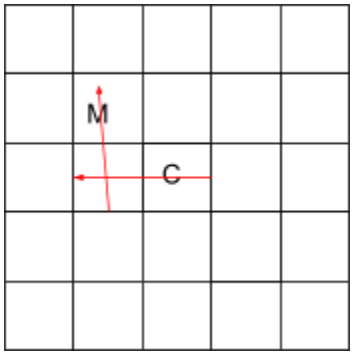
Monster Proximity:

The monster proximity function checks if a monster is within a depth. We have depth as the number of the ring around the character. An example of depth is shown below for more clarification. If a monster is detected within depth 3 then the manhattan distance is checked. When the manhattan distance is less than or equal to 2 the monster type defaults to aggressive. This default is a fail safe for the character to be more pessimistic and avoid death. The function returns a tuple of tuples summarized as ((true if monster in depth, monster name unless fail safe)(location of monster x, y))

Depth:
Blue: depth 2
Yellow: depth 1



Manhattan Distance:
abs(x2-x1)+abs(y2-y1)



Tree Building:

Tree building occurs in 2 stages, building the terminal nodes then building leaf nodes. The terminal nodes are built by assigning a utility for every possible monster move given a character move. The utilities are based on the sum of the change in Manhattan distance to the exit caused after a given character move multiplied by a weight and an assignment from our get reward function. The weight was a way for us to

tune how much it cared about the change in distance from the exit. The get reward function takes the manhattan distance between the monster and the character after the speculated moves then assigns it a value from the table below:

Manhattan distance	reward
0	-100
1	-20
2	-2
3	4
4	8
5	24
other	0

Once all of the utilities are assigned phase two begins. In phase two the expected value for every leaf node is calculated and assigned to a list of tuples containing the (action,expected value). This tree building was only used for the expectimax function.

Expectimax:

The expectimax function takes the expected values from the tree and returns the action as a tuple of dx,dy with the highest expected value.

Minimax:

The minimax function used for this project was modeled after the pseudo code presented in class. The algorithm only goes a singular move into the future and starts by getting all possible moves for both the character and the monster. For each character move it iterates through each monster move and calculates h(n) or the straight line distance from the character move to the monster move. It stores the minimum distance calculated assuming that is the move the aggressive monster will take for each character move. As it iterates through the rest of the possible character moves it updates a max value variable with the largest min value found from each monster move. Finally the function returns the character move direction that corresponds to the largest distance value assuming the monster is moving optimally.

Bomb Usage:

Utilizing the bomb became extremely important when solving board variants 4 and 5. The place bomb action was always used within a created function drop bomb

that also sets a variable to save the location of the bomb given a bomb is not already on the map. Along with the drop bomb function, two other functions were created to help with the usage of the bomb. One function called blast radius with the bomb location and a move as inputs determined if a location was in the explosion path of the bomb. This proved extremely useful when getting possible moves and using minimax to ensure the bomberman did not blow himself up after placing a bomb. Another function called check bomb kept updating the bomb timer with every do iteration and reset the bomb location and timer when it reached zero.

Experimental Evaluation:

Through coding, a lot of things were tested such as different waits for the utilities used in tree building, and different ways of switching between the local search algorithms. After we could complete a variation the random seed was changed and to verify that solution worked. The table below is the validation testing of each variant with our final code. We had one failure with variant 5 due to bomberman getting trapped between two walls and the two monsters. This failure was considered to be an outlier as it only occurred 1 time out of the 10 random seeds we tested.

S = Success F = Failure

Random Seed	**Variant 2**	**Variant 3**	**Variant 4**	**Variant 5**
123	S	S	S	S
234	S	S	S	S
345	S	S	S	S
456	S	S	S	S
567	S	S	S	S
678	S	S	S	S
789	S	S	S	S
891	S	S	S	S
912	S	S	S	F
777	S	S	S	S
Percentage of Success	**100%**	**100%**	**100%**	**90%**