RBE 470X-C24 AI for Robotics

Project 1: AI character for Bomberman in 5 different scenarios

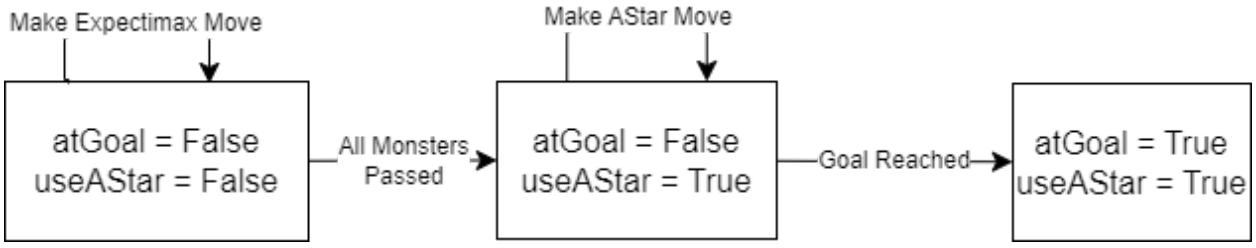
Professor Pinciroli

February 11, 2024

Team 01: Sakshi Gauro, Neil Kale

Structure of the Approach

We approached the project variant by variant. The first variant has no opponent, so we solved it using A* search. Next, we implemented Expectimax to defeat the random 'stupid' monster and Minimax to defeat the intelligent 'self-preserving' monster. Our approach to defeat the 'aggressive' monster involved progressively improving our Expectimax implementation. We also switched to using A* once the agent was closer to the end cell than any of the monsters since it could not be caught by the monsters at that point. This is summarized in the simplified state machine below.



Note, we chose to use Expectimax to handle the intelligent monsters, since for most of the game, the

Individual Sections

AStar

We develop a versatile AStar algorithm designed for efficient pathfinding between two points within the game. This function, integral to our AI's navigation system, accepts the game's current state, a starting position, and a target destination, all represented as tuples. It is made generic so that it can be reused on other parts of the program.

Expectimax

Our Expectimax implementation has unique enhancements to navigate complex scenarios involving multiple monsters with differing behaviors. Firstly, if the agent's shortest path to the goal is closer than the monster's shortest path, the agent will simply move based on the A* algorithm previously implemented. It cannot be caught based on the following proof by contradiction.

Let at time = 0, the monster be m steps from the goal and the agent be n steps from the goal where m > n. Suppose after k steps, the monster catches the agent moving on its optimal path. At that point, the agent is n-k steps from the goal. Therefore, the monster is also n-k steps from the goal and began n steps away from the goal. Contradiction!

For a single monster scenario, we simulate possible worlds based on the type of monster. Each monster results in up to 9 possible worlds (8 directions of movement and choosing not to move). To reduce the branching factor of the search, we track the intelligent monsters' last-known direction of movement to anticipate their next move and trim branches. We also use the knowledge that the intelligent monsters always choose to attack the agent when possible, so other possibilities do not need to be expanded.

When facing two monsters, our algorithm simulates a second layer of potential outcomes based on the first monster's actions, thereby accounting for the compounded uncertainty of multiple adversaries.

For the heuristic, we use the agent's A* distance to the goal summed with a penalty if the A* distance between the agent and the monster is less than or equal to two steps. Our agent aims to minimize this heuristic by reducing the distance to the goal and staying outside the penalty distance from the monsters. We additionally experimented with rewarding the agent for having many open surrounding cells to avoid traps, but this did not work since sometimes it is advantageous for the agent to creep close to the walls to avoid monsters. There is also a large magnitude heuristic associated with the terminal states: +500 for dying and -500 for reaching the exit.

This nuanced approach allows our AI to navigate the game's challenges with a strategic depth that goes beyond simple reactive tactics, making for a more engaging and successful agent.

Minimax

We have the minimax algorithm implemented; however, it’s not the best at fighting the monster. We implemented an *ActionSet* class which stores all the 8 actions/directions in *enums* along with their (dx, dy). The *Minimax* class inherits from *CharacterEntity* and includes methods *do* (to find the best action to take), *takeAction* (best action to take) and *minimax* (performing the minimax algorithm).

The program control starts from *do,* calling the *minimax* algorithm to figure out the best action and then calls *takeAction* to execute the action.

The *minimax* algorithm takes in a world state and depth. It recursively explores possible actions and evaluates their outcomes to determine the best action. The algorithm alternates between maximizing and minimizing the expected utility, simulating the actions of the character and the monster. The character is trying to minimize the utility while the monster is trying to maximize it.

The heuristic method defines the utility function used to evaluate game states, which calculates a heuristic value based on factors such as the character's distance to the exit and monsters. The heuristic is set to infinity (bad heuristic), if the character dies, -infinity (good heuristic) if the character finds an exit by taking that action. If neither is true, the heuristic is calculated by subtracting the monster distance from us to the distance of the exit from us. This made the character prioritize the path that made us closer to the exit while keeping a safe distance from the monster. The *findPossibleActions()* and *findPossibleMonsterActions()* are helper methods to find possible actions for the character and the monster, respectively. The *simulateActionSimple()* is a helper method used to simulate the effects of an action and generate possible future game states for the Minimax algorithm.

Our Minimax implementation still has a few critical bugs that occasionally hamper its performance; hence, we focused our efforts on developing an optimized Expectimax agent that could handle the more difficult Variants 4 and 5.

Experimental Evaluation

We evaluated our top methods over 20 runs on each of the game variants.

	Variant 1	Variant 2	Variant 3	Variant 4	Variant 5
	A Star	Expectimax	Expectimax	Expectimax	Expectimax
Wins / 20	20	17	19	15	11
Win Pct	100	85	95	75	55

Figure 1: Experimental Data on the number of wins for each variant over 20 trials.

We performed very well against the single agents, but just barely passed the 50% threshold for success against the two monster variant (Variant 5). Some further debugging and hyperparameter optimization (particularly of the heuristic values) would likely improve performance against the two monster game.