Albert Lewis

Kay Siegall

S. Taylor

RBE 470x: Artificial Intelligence for Robotics

Prof. Pinciroli

February 9th, 2024

Project 1 Report

Structure of the approach

Simply put, our agent utilizes a three-depth expectimax search. We went through many iterations to get to this point, but our end result was just an expectimax without pruning, where the monster's turn was the chance node. The only main deviation from the base algorithm is that our chance nodes also take into account the current evaluation of the world as part of its calculation. Other than that, our evaluateState heuristic is heavily tuned, balancing the difference between the distances between the goal and both our agent and the nearest monster to figure out if we need to care about its position before bee-lining it to the goal. We also have a reward term for the number of valid moves from a given position, which helps prevent the agent from blocking itself in a corner.

Our chance nodes inspect the monsters to figure out what type they are. If they are a stupid monster, we branch for each random possibility. If they are self preserving or aggressive, we check if it needs to change direction (monster.must_change_direction()), and if it does, then we branch. If the self-preserving or aggressive monster doesn't need to change direction, then its behavior is deterministic and we can just pull the resultant move by letting it run do().

Interesting bits of approach

First, we implemented the A* pathfinding algorithm, and a way for the agent to follow the path. We had planned to use A* to evaluate a state, by measuring the length of the A* path from the character, to the exit. However, we later switched to the wavefront algorithm as a heuristic, to save on time and processing complexity. This worked well for the first three variants, however, it could not succeed against the aggressive monster present in both variant 4 and 5.

To deal with the aggressive monster, we initially turned to bombs. If our character was at the same X or Y coordinate as any monster or the exit cell, it would place a bomb. The bombs have a 10-turn timer before they explode, destroying entities and walls. For this approach to be successful, a depth-13 search would be needed. Unfortunately, this was not feasible for us to develop for this project, due to the high run time which made debugging extremely cumbersome.

Eventually, we came to the conclusion that this was not the right approach, so we scrapped all of our code, and rewrote it from scratch using expectimax. For this iteration of the code, our goal was to make it more readable, and easier to debug. We discarded the bomb strategy, and instead opted for a more functional movement algorithm.

Within our evaluation function, we based the utility of a state off of the character's distance to the goal, distance to a monster, and world events. World events like our character dying or our character winning are very high value, as they can end the game. We heavily penalized our agent for dying, and heavily rewarded it for winning. To calculate the distance to the exit, we ran a wavefront algorithm, with the origin being the exit cell. This is much lower time complexity compared to running A* for each of our agent's movements, because the wavefront only needs to be calculated once (for now). To penalize our agent for its proximity to a monster, first we check where the monster is on the wavefront. If a monster is closer to the exit, or, if

between our agent and the monster there is a difference of 2 or less steps to take to the exit, we calculate the A* distance between our character and the monster, and subtract half that value from the utility. If the A*-generated path between our character and the monster is less than 3 steps, then we penalize our agent further by subtracting 50 points for each “step” closer they are. Additionally, we rewarded our agent the number of potential moves present at its next location, so that it wouldn’t back itself into a corner. Using this function, we ran into many situations where many different outcomes all resulted in the same Utility value. So, we decided to run A* as a tiebreaker. Whichever “equally good” result had the shortest A*-generated path to the exit was the chosen action.

We also made a testing script that would run our algorithm on many sequential seeds, and return a list of the seeds our agent succeeded on. This enabled a more “hands-off” testing workflow, and it allowed us to perform our own “parallel search” where we run several tests on our own computers at the same time.

Experimental evaluation

Our agent survives 100% of the time in variant 1. We use the A* algorithm to find a path to the goal, and the agent follows the generated path.

Variant 2 and variant 3 also work very consistently. Variant 2 was successful for all the trials we ran, and variant 3 succeeded more than 75% of the time. The random monster is easily evaded by our algorithm at depth 3, and the same is true for the self-preserving monster.

Variant 4 worked for 14 out of 20 trials. This works well at depth 3, and is able to evade the monster pretty effectively.

Variant 5 worked for 11 out of 20 trials. It also runs at depth 3, but is noticeably slower than Variant 4, likely due to the extra monster adding more time complexity.

Variants 1 and 2 run very quickly, as they evade the random monster very easily. Variants 3 and 4 both run at approximately the same speed, as we only have to evade one self-preserving monster. Variant 5 is the slowest, each turn takes approximately 5 seconds to complete.

Super Secret Extra Stuff :)

So there's a few exploits that exist within this framework. We didn't have time to flesh out proper solutions for either thing, so we're not looking for extra credit, but take a look at our abusing_bugs.py. There's 4 variants of Thanos in there that beat the game pretty much every time using a couple different solutions.

Rocketman starts by instantly killing every monster on the board, and then runs towards the exit, destroying any walls in his path, and leaving behind a trail of explosions. This one is our favorite but runs too fast to watch in real time :(

Earthbender deletes all walls, and then summons walls surrounding any monsters in the world, encasing them in stone and allowing for an easy journey to the exit.

The Flash removes the monster's turn by setting its movement to (0,0). This is the only one of these variants that doesn't *always* win, since it sometimes runs right into the monster (path following is just A* so it doesn't actually care if it hits a monster.)

These three variants exploit python being terrible at code safety. We accessed the real world object by replacing the SensedWorld.from_world static method with a simple lambda that returns the world object. We wait for our next turn, and voila, phenomenal cosmic power!

This next variant doesn’t even need to do that! Abra just sets our self.x and self.y to be one away from the exit cell, and then calls move() to move us to the exit, beating all 5 variants in one turn.

Sorry for breaking your game! :)