Search and Destroy

Frank Fondell

April 9th, 2021

Problem 1

To efficiently update the belief state, I convert the original conditional probability into a formula to calculate the belief value. I started with:

$P = (Target[Celli]|Observationst \wedge FailJ)$

When Bayes theorem is applied to rearrange and simplify, it looks like this:

$P = ((FailJ|Target[Celli]) * P(Target[Celli]|Obersvationst))/P(FailJ)$

The three different pieces above are now probabilities that are able to be calculated with the information I have. For the left most piece, given target in i, the probability of failure is the part that changes depending on whether the cell in question is being searched or not. If j=i, the chance of failure is equal to the probability of a false negative for a given terrain, otherwise the probability will be equal to 1.0. As for the top right piece, given everything observed so far, this will be the previous value of the belief for a given cell, as stated in the directions, this value is equal to $1/Dimension^2$ at t=0. Lastly, the right most piece represents the chance to return a failure given a cell j. This value changes depending on the false negative value for the terrain shown as "FN," and the current belief state. The formula is modeled as such below. I implemented problem 1's formula in the "numpyUpdate," method, it had originally been under "updateBeliefs."

$P(FailJ) = (1 - Belief[CellJ]) * (Belief[Celli] * FN)$

Problem 2

To find the probability of the target being found given cell i, I converted the formula in the directions to the one shown below:

$P(Target[Celli] \wedge SuccessinCelli|Observationst)$

To figure out a mathematical formula for this one, I thought about it logically. It states above that given our observations so far, what is the probability to find the target in cell i. The chance for a false negative is constant, so the opposite probability of FN multiplied by the current belief state of cell i will give you the chance to find the cell if it is searched. This was implemented under the "probFound," function. The mathematical representation is also below.

$P(Found) = (1 - FN) * (Belief[Celli])$

Problem 3

Unfortunately I wasn't able to get my program to run at dimensions as high as 50 by 50 in a timely manner. There had been a bottleneck for awhile that I could not find. However, I was able to run my code with smaller dimensions, specifically 10 by 10, to get some quality data. For agents 1 and 2, I ran 1000 trials of each to acquire a representative average. The average score for agent 1 was 776 and the average score for agent 2 was 1066. These averages can always be erratic but it seems that agent 2 was a clear loser for my program.

Problem 4

To create an improved algorithm, I started by thinking what would impact the score the most. Since distance traveled throughout the search will almost always heavily outweigh the total number of searches, I decided to change the priorities in my fringe. Through extensive testing and graphing the results of many searches, I discovered a near optimal ratio to use in my new algorithm. I've implemented it under "improvedAgent," and "improvedFringe." In this new algorithm, Ive added a method for finding the distance between two nodes, and using that distance to negatively assess the overall judgement of my fringe towards farther nodes. When I tested my algorithm against the others, at lower dimensions, it was a clear winner with an average score of 667. I eventually found the location of my bottleneck with almost no time to spare and found that my algorithm doesn't perform well at very high dimensions. The original implementation is under "improvedAgent. " And the speed efficient method is under "improvedAgent1."