Intro to AI Assignment 2B

Harrison Gropper, Justin M Hu, Adideb Nag

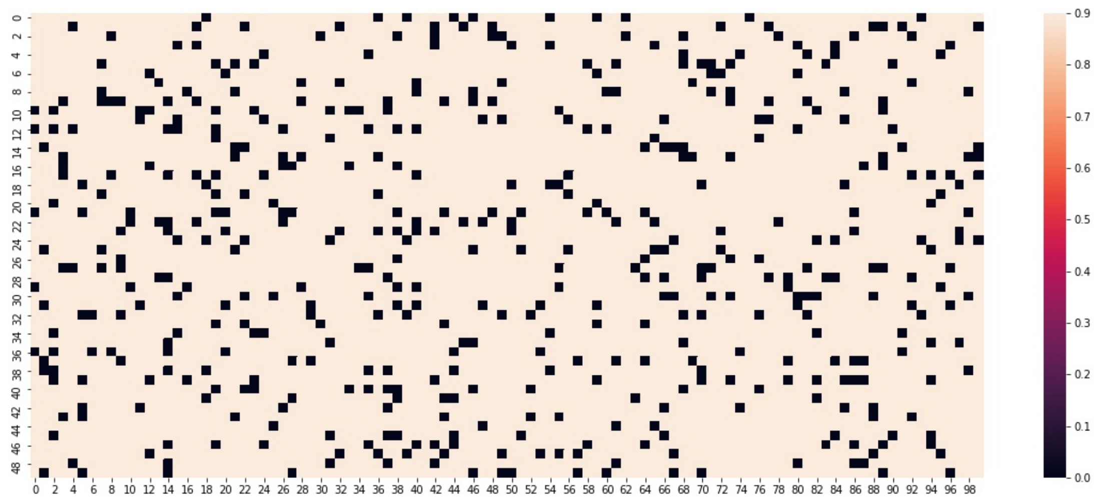
December 14, 2022

Question 5 - Unknown Cells

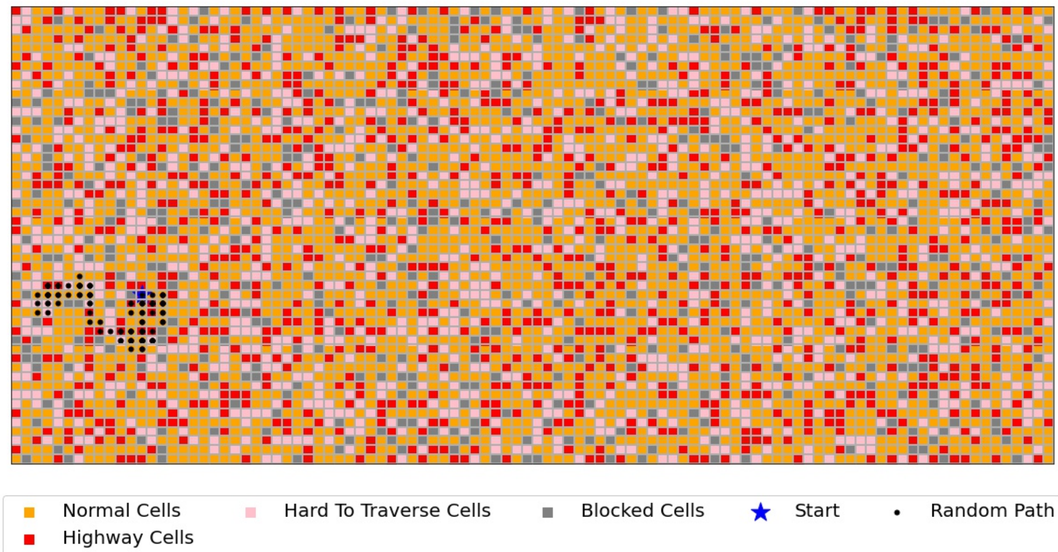
5c) Estimating Positions

Initial Probabilities

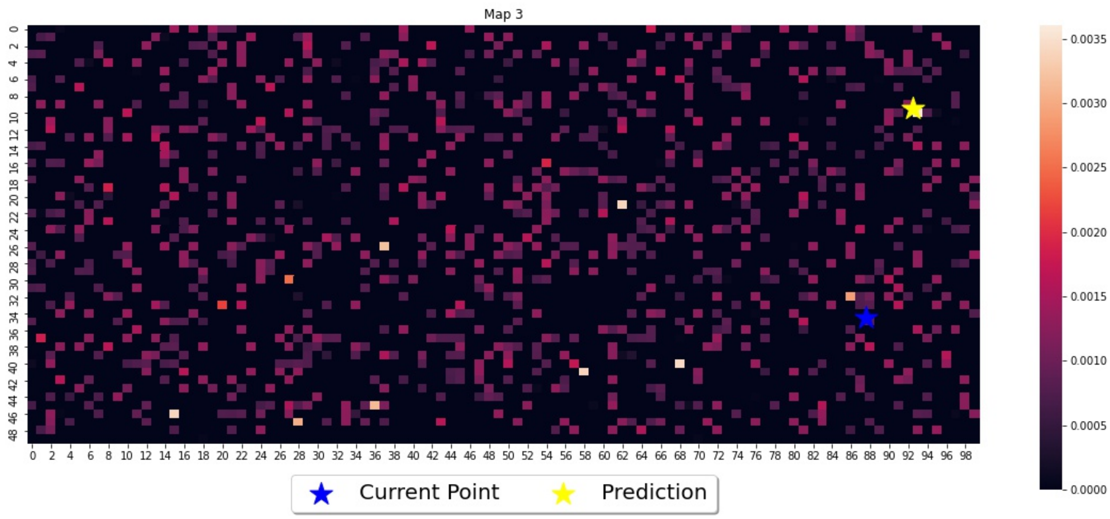
We do not know where our agent is, so there is an equal probability that it could be in each of the unblocked cells. So, our current heat-map looks like this:

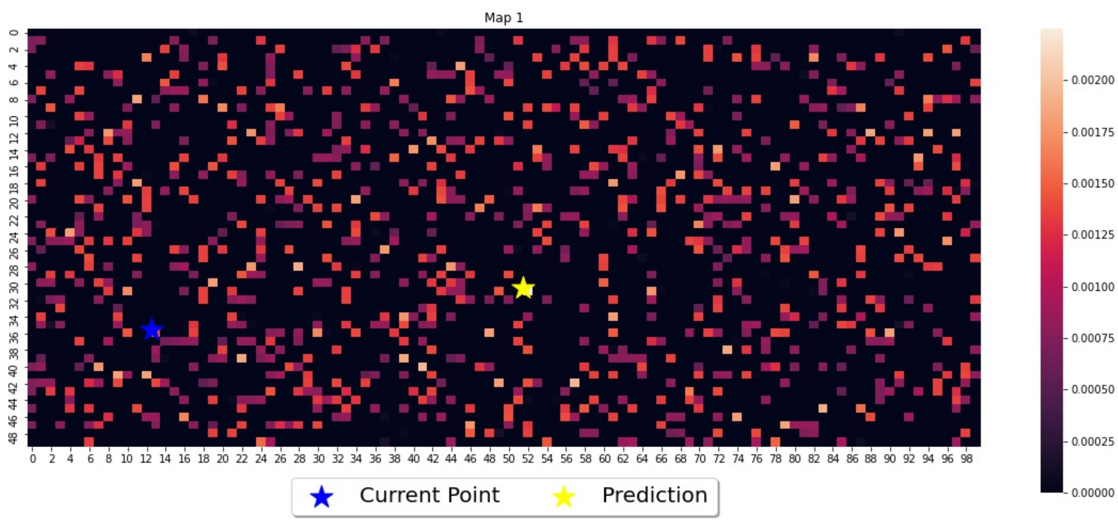


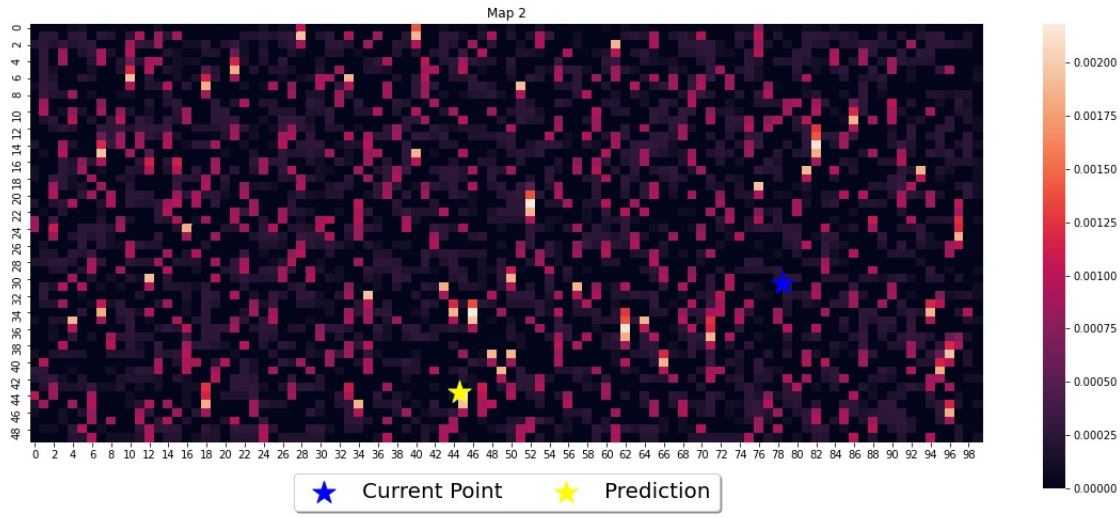
For a map that looks like this:


Normal Cells
Hard To Traverse Cells
Blocked Cells
Start
Random Path
Highway Cells

Example Heat-maps for 10 iterations

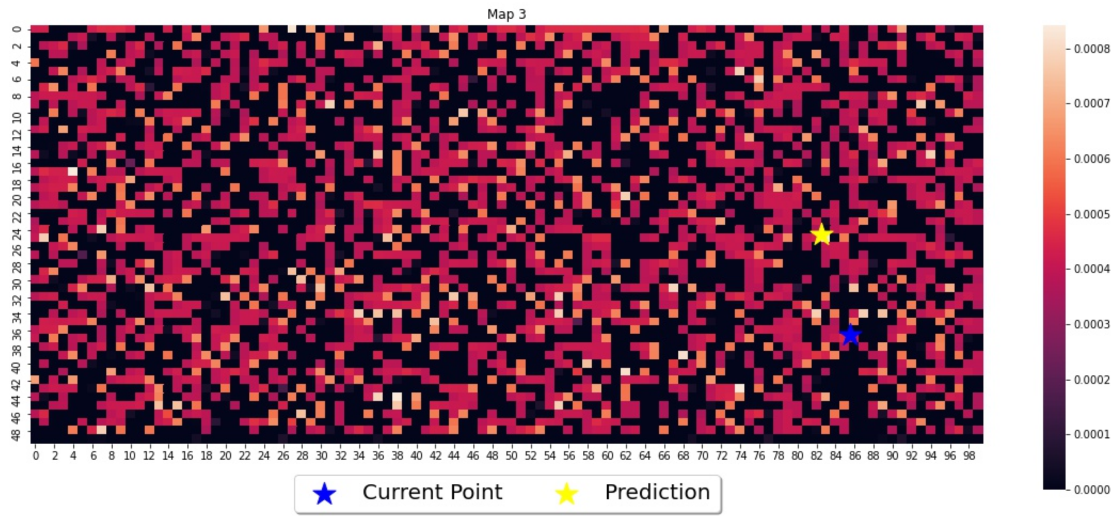


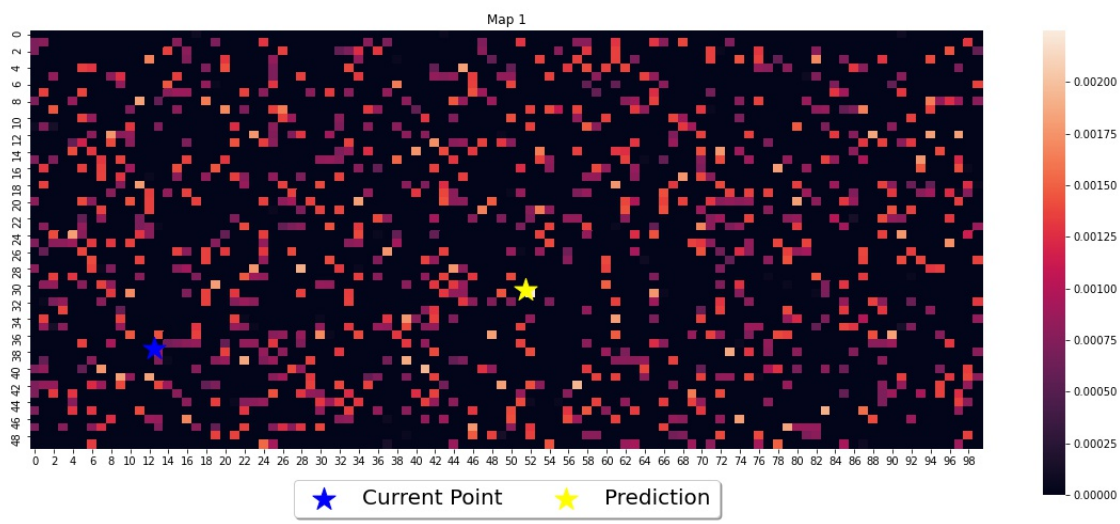


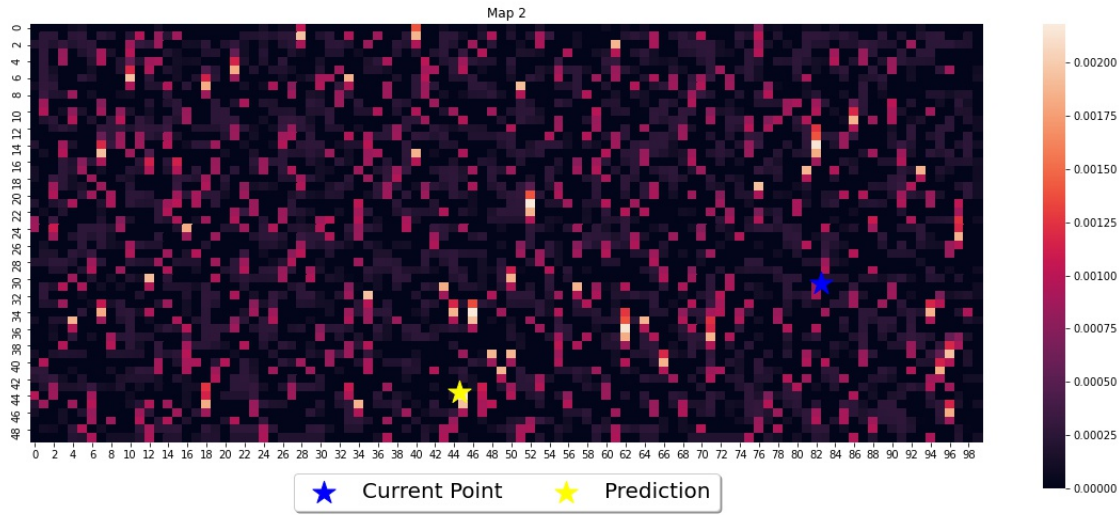


Map 2
Current Point
Prediction

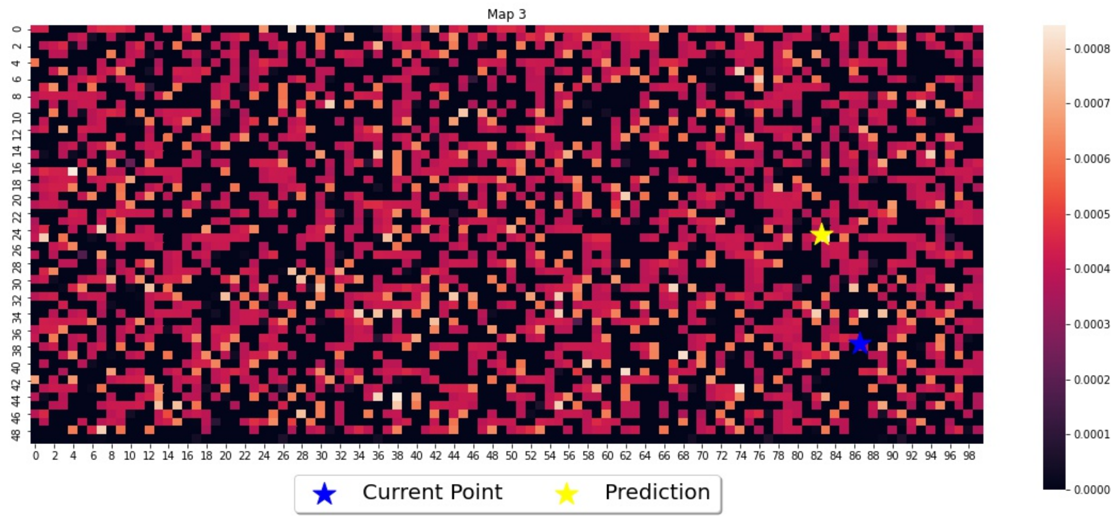
Example Heat-maps for 50 iterations

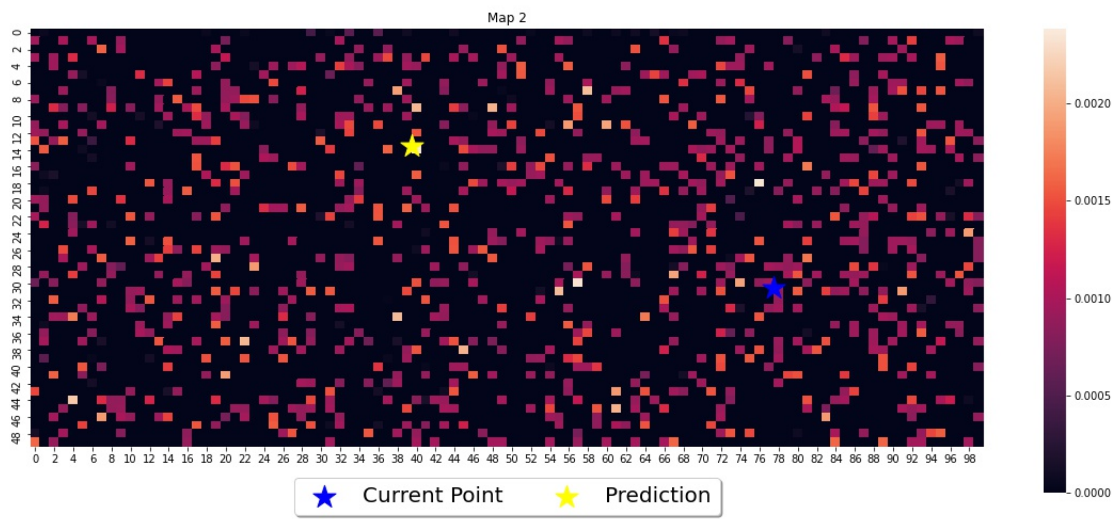


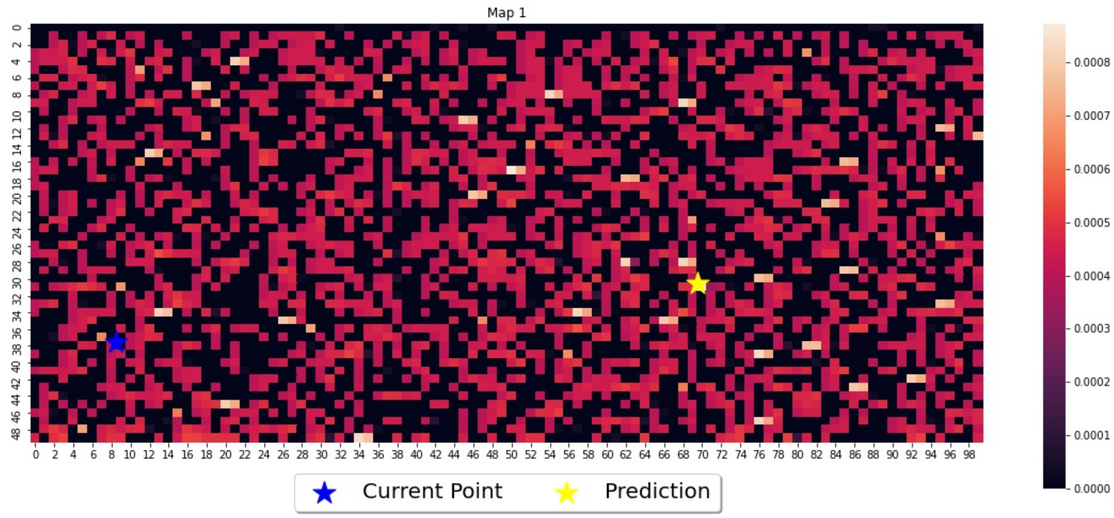



Map 2
Current Point
Prediction

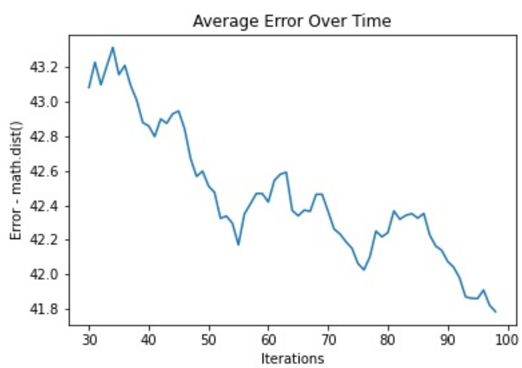
Example Heat-maps for 100 iterations



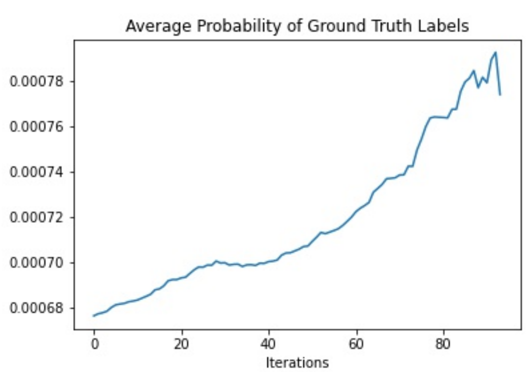




Average Error Over All 100 Experiments



Average Probability of Ground Truth Cells



Extra Features

In addition to all of the computation required to make these graphs we also saved a simulation of all 100 iterations.

To see each iteration you can navigate to the animation folder and find the GIF files to view each experiment.
We have also saved each image in a folder that was used to make these GIFs.
The graph data containing the cell types and the ground truth labels were saved in a folder named data.
We have provided jupyter notebook files with well documented code, so you can see how we implemented the filtering algorithm.
We have also included a .py version of the files as well.
NOTE: The graphs used in this example were not the same graphs from assignment 2B. We generated new graphs for the 100 iterations.

Conclusion

It seems that the less cells covered by the agent gives better results using the filtering algorithm.
When the agent is bounded (in a tight area and does not leave) the filtering algorithm works exceptionally well minimizing the overall error greatly.
This is because it is easier to rule out certain positions in the graph when our sensor reading sequences are relatively similar.

However, when our agent randomly decides to wonder off the filtering algorithm still minimizes error, but is not as accurate.
This is because we have varying sequences of evidence that confuses the filtering algorithm.

Question 6 - The Mechanic

6a) Expected Net Gain From Buying C_1

There is a 70% chance that it is in good shape.
Then, it costs \$3000 to buy and has a \$4000 market value.
This yields a +\$1000 gain.
So, we have that $0.70 * 1000 = 700$.

There is a 30% chance that it is in bad shape.
We have a \$1000 in profit and we need \$1400 in repairs.
So, we have that $0.30 * 400 = 120$.

Our final answer is $\$700 - \$120 = \$580$.
The expected net gain from buying C_1 given no test is \$580.

6b) Bayes' Theorem To Calculate Pass/Fail

Remember that $\frac{P(B|A)P(A)}{P(B)}$. So,

P(pass(c_1)) = P(pass(c_1)$|q^+(c_1)) * q^+(c_1)+$ P(pass(c_1)$|q^-(c_1) * q^-(c_1)$

$P(\text{pass}(c_1)|q^+(c_1)) = 0.80 = \frac{P(q^+(c_1)|pass(c_1))*pass(c_1)}{q^+(c_1)}$

$0.80 = \frac{P(q^+(c_1)|pass(c_1))*pass(c_1)}{0.70}$

$P(pass(c_1)|q^-(c_1)) = 0.35 = \frac{P(q^-(c_1)|pass(c_1))pass(c_1)}{q^-(c_1)}$

$0.35 = \frac{P(q^-(c_1)|pass(c_1))pass(c_1)}{0.30}$

Our answers are below:

$P(q^+(c_1)|pass(c_1)) = \frac{P(pass(c_1)|q^+(c_1))P(q^+(c_1))}{P(pass(c_1))}$
$= \frac{0.80*0.70}{0.80*0.70+0.35*0.30} = 0.84$

$P(q^-(c_1)|pass(c_1)) = \frac{P(pass(c_1)|q^-(c_1))P(q^-(c_1))}{P(pass(c_1))}$
$= \frac{0.35*0.30}{0.80*0.70+0.35*0.30} = 0.158$

$P(q^+(c_1)|\neg pass(c_1)) = \frac{p(\neg pass(c_1)|q^+(c_1))P(q^+(c_1))}{P(\neg pass(c_1))}$
$= \frac{(1-0.80)*(0.70)}{1-10.8*0.70+0.35*0.30} = 0.418$

$P(q^-(c_1)|\neg pass(c_1))) = \frac{P(\neg pass(c_1)|q^-(c_1))P(q^-(c_1)}{P(\neg pass(c_1))}$
$= \frac{(1-0.35)*0.30}{1-(0.80*0.70+0.35*0.30)} = 0.58$

6c) Best Decision and Expected Utility

Given that the car passes the mechanic's test the best decision is to buy the car since there is an 84% chance that is it in good condition and since the expected utility is positive

Expected Utility:

$0.84 * (\$4000 - \$3000) + 0.16(\$4000 - \$3000 - \$1400) - \$100$
$= \$840 - \$64 - \$100$
$= \$676$
Given that the car fails the mechanic's test, the best decision is to buy the car because even though it is only 41.80% that the car is in good shape, the expected utility is still positive

Expected Utility:

$0.418 * (\$4000 - \$3000) + 0.582(\$4000 - \$3000 - \$1400) - \$100$
$= \$418 - \$232.80 - \$100$
$= \$85.20$

6d) The Value of Optimal Information for the Mechanic's Test

Value of Optimal Information = Utility with Mechanic - Utility without Mechanic
Value of Optimal Information = $\$676 - \$580 = \$96$

It is worth going to the mechanic because the value of optimal information is positive even with the $100 paid for the test accounted for in the utility equations.

Question 7 - Markov Decision Process

Original Utilities

In the code, we use the given initial values to iterate through the Value Iteration Formula. Using this process, we created a PossibleActions class, each holding two action names and corresponding weights. It continues doing this, updating the information until the difference goes past a given max_error value.

Five Intermediate Results

Our code starts with the given default utility values:
$V(s_1) = 2.0$
$V(s_2) = 1.0$
$V(s_3) = 2.0$
$V(s_4) = 2.0$

And the given values for the other variables:
$max\_error = 0.001$
$discountFactor = 0.75$

The following are 5 Iterations of the Code:
Iteration 6 :
u_1 : 1.1650237910156256
u_2 : 1.5587219203125005
u_3 : 2.243067409667969
u_4 : 1.6079361086425783
$Policy : s_1 - 2, s_2 - 2, s_3 - 3, s_4 - 1$

Iteration 7 :
u_1 : 1.1395152338378909
u_2 : 1.5796487338476566
u_3 : 2.205952081481934
u_4 : 1.6346657096740724
$Policy : s_1 - 2, s_2 - 2, s_3 - 3, s_4 - 1$

Iteration 8 :
u_1 : 1.1517267108801272
u_2 : 1.5605185589663093
u_3 : 2.2259992822555543
u_4 : 1.611617583225861
$Policy : s_1 - 2, s_2 - 2, s_3 - 3, s_4 - 1$

Iteration 9 :
u_1 : 1.1397295565675358
u_2 : 1.569677353198279
u_3 : 2.208713187419396
u_4 : 1.623420834264439
$Policy : s_1 - 2, s_2 - 2, s_3 - 3, s_4 - 1$

Iteration 10 :
u_1 : 1.1450119340437939
u_2 : 1.5606795154313797
u_3 : 2.217565625698329
u_4 : 1.6126379640779254
$Policy : s_1 - 2, s_2 - 2, s_3 - 3, s_4 - 1$

Iteration 11:
u_1 : 1.1393345685533245
u_2 : 1.5646413027337045
u_3 : 2.209478473058444
u_4 : 1.6178046446522167
$Policy : s_1 - 2, s_2 - 2, s_3 - 3, s_4 - 1$

Final Results

The Last Iteration was: Iteration 17:
u_1 : 1.1382596177940973
u_2 : 1.5600493275743013
u_3 : 2.2092827785843383
u_4 : 1.612773041296828
$Policy : s_1 - 2, s_2 - 2, s_3 - 3, s_4 - 1$

and printed at the end:
Iterations: 18
Runtime: 12910 microseconds

Computation Time and Number of Iterations

As shown, with the setup that the code follows, it takes 18 iterations and 12910 microseconds to compute the following output.

Code Implementation

The Code looks as follows:

The Question7 Class does most of the work and looks like this:

```
package Project2;

public class Question7 {
    public static PossibleAction[] actions = new PossibleAction[4]; // List of Actions that can be taken and their
                                                                     // properties
    public static double[] utils = new double[4]; // Utility Values
    public static double[] utils_next = new double[4]; // Next utility values
    public static double[] reward = new double[4]; // Calculated reward function output
    public static char[] policy = new char[4]; // Policy character array

    // Populates the graph based on the diagram provided in the problem
    public static void populate() {
        utils[0] = 2.0;
        utils[1] = 1.0;
        utils[2] = 2.0;
        utils[3] = 2.0;
        reward[0] = 0.0;
        reward[1] = 0.0;
        reward[2] = 1.0;
        reward[3] = 0.0;

        PossibleAction action0 = new PossibleAction(); // creates new PossibleAction Object and adds it to the
        action0.action_name1 = '1';
        action0.weight1[0] = 0.2;
        action0.weight1[1] = 0.8;
        action0.weight1[2] = 0.0;
        action0.weight1[3] = 0.0;
        action0.action_name2 = '2';
        action0.weight2[0] = 0.2;
        action0.weight2[1] = 0.0;
        action0.weight2[2] = 0.0;
        action0.weight2[3] = 0.8;
        actions[0] = action0;

```

```java
        PossibleAction action1 = new PossibleAction();
        action1.action_name1 = '2';
        action1.weight1[0] = 0.0;
        action1.weight1[1] = 0.2;
        action1.weight1[2] = 0.8;
        action1.weight1[3] = 0.0;
        action1.action_name2 = '3';
        action1.weight2[0] = 0.8;
        action1.weight2[1] = 0.2;
        action1.weight2[2] = 0.0;
        action1.weight2[3] = 0.0;
        actions[1] = action1;

        PossibleAction action2 = new PossibleAction();
        action2.action_name1 = '4';
        action2.weight1[0] = 0.0;
        action2.weight1[1] = 1.0;
        action2.weight1[2] = 0.0;
        action2.weight1[3] = 0.0;
        action2.action_name2 = '3';
        action2.weight2[0] = 0.0;
        action2.weight2[1] = 0.0;
        action2.weight2[2] = 0.0;
        action2.weight2[3] = 1.0;
        actions[2] = action2;

        PossibleAction action3 = new PossibleAction();
        action3.action_name1 = '1';
        action3.weight1[0] = 0.0;
        action3.weight1[1] = 0.0;
        action3.weight1[2] = 0.9;
        action3.weight1[3] = 0.1;
        action3.action_name2 = '4';
        action3.weight2[0] = 0.8;
        action3.weight2[1] = 0.0;
        action3.weight2[2] = 0.0;
        action3.weight2[3] = 0.2;
        actions[3] = action3;
    }
```

```java

    // Calculates the upcoming utility values
    public static void nextUtil(double discount_factor) {
        double util1;
        double util2;

        // calculating the new util1 and util2 values for each i value
        for (int i = 0; i < 4; i++) {
            util1 = reward[i];
            for (int j = 0; j < 4; j++) {
                util1 += discount_factor * actions[i].weight1[j] * utils[j];
            }

            util2 = reward[i];
            for (int j = 0; j < 4; j++) {
                util2 += discount_factor * actions[i].weight2[j] * utils[j];
            }

            if (util1 >= util2) {   //comparing the util1 and util2 value for new policy value
                utils_next[i] = util1;
                policy[i] = actions[i].action_name1;
            } else {
                utils_next[i] = util2;
                policy[i] = actions[i].action_name2;
            }
        }
    }

```

```java

    public static void main(String[] args) {
        if (args.length < 2) {                          //checking if values for both max_error and discount_factor are given within arguments
            System.out.println("Not enough arguments");
            return;
        } else {
            double max_error = Double.parseDouble(args[0]);
            double discount_factor = Double.parseDouble(args[1]);
            if (max_error < 0.0) {
                System.out.println("Error");
                return;
            }
            long start = System.nanoTime();             //keeping track of the time (start value)

            populate();                                 //populating the graph with the necessary information
            int iterations = 0;                         //counting the number of iterations

            while (true) {
                if (iterations != 0)
                    System.out.println("\n");

                System.out.println("Iteration " + iterations + ": ");
                nextUtil(discount_factor);

                System.out.println("u1: " + utils_next[0] + "\nu2: " + utils_next[1] + "\nu3: " + utils_next[2]
                        + "\nu4: " + utils_next[3]);    //printing values of updated util values
                System.out.println("Policy: s1 -> " + policy[0] + ", s2 -> " + policy[1] + ", s3 -> " + policy[2]
                        + ", s4 -> " + policy[3]);      //printing the policy for this iteration

                iterations++;

                double max_diff = 0.0;                  //calculating the difference between the util values
                for (int i = 0; i < 4; i++) {
                    double diff = utils[i] - utils_next[i];
                    if (diff < 0) {
                        diff *= -1;
                    }
                    if (diff > max_diff) {
                        max_diff = diff;
                    }
                    utils[i] = utils_next[i];
                }

                if (max_diff <= max_error) {            //checking if the difference falls below the max_error
                    break;
                }
            }
            long end = System.nanoTime();               //stopping time for timer
            int elapsed = (int)(end - start) / 1000;    //calculating processing time in microseconds

            System.out.println("\n\nIterations: " + iterations);
            System.out.println("Runtime: " + elapsed + " microseconds");

        }
    }
}
```

And the PossibleActions Class looked like this:

```java
package Project2;


public class PossibleAction
{
    char action_name1;              //first action name
    double[] weight1;               //first action weight
    char action_name2;              //second action name
    double[] weight2;               //second action weight

    public PossibleAction() {       //Constructor to create the object and initialize all values
        action_name1 = ' ';
        weight1 = new double[4];
        action_name2 = ' ';
        weight2 = new double[4];
    }
}
```

The code implemented for question 7 has been provided in our submission.