

ROB311: Final Project

Aman Bhargava — 1005189733 — Bharga54

Question 1: Constraint Satisfaction [35 pt.]

A: Boolean SAT [5 pt.]

B: Binary Constraints [5 pt.]

C: Autolab Submission [12 pt.]

D: Algorithm Description [5 pt.]

E: Difficulty of Sudoku [8 pt.]

Question 2: Sales Bot POMDP [35 pt.]

A: Belief-Space MDP Model [5 pt.]

B: Visualizing Utility and Policies [15 pt.]

C: Autolab [10 pt.]

D: Algorithm Description [5 pt.]

Question 3: Learning to Drive [30 pt.]

A: Reinforcement Learning Algorithm [10 pt.]

B: Autolab [15 pt.]

C: Algorithm Description [5 pt.]

Question 1: Constraint Satisfaction [35 pt.]

A: Boolean SAT [5 pt.]

Where variables represent "true" and "false". We want to know if the logical statement made of boolean variables can be satisfied. Does there exist an assignment of "true"/"false" values that yields a "true" statement evaluation?

Notation: Let $x_{ij} = \text{True}$ if the i th cell has value j and False otherwise. Cell numbering is as follows:

```
-----  
| 1 | 2 | 3 |  
-----  
| 4 | 5 | 6 |  
-----  
| 7 | 8 | 9 |  
-----
```

For each value j , we have the following constraint:

$$A_{i,j} = x_{i,j} (x'_{1,j} x'_{2,j} \dots x'_{i-1,j} x'_{i+1,j} \dots x'_{9,j}) (x'_{i,1} x'_{i,2} \dots x'_{i,j-1} x'_{i,j+1} \dots x'_{i,9})$$

Where xy means "x and y" and x' means "not x". This constraint represents that if the i th cell takes on value j , the rest of the cells $1, \dots, 9$ must NOT take on that value. Cell i also must not take on other values (i.e., each cell can only have 1 value). This constraint must hold for some value j for every i . Thus, we can generate the following composite boolean constraint on every cell i :

$$B_i = (A_{i,1} + A_{i,2} + \dots + A_{i,9})$$

Where $x + y$ means "x or y". This statement can be interpreted as, "cell i must take on exactly one value between 1 and 9 not shared by any other cells". The final constraint can be written as:

$$B_1 B_2 \dots B_9$$

That is, constraint B must hold for all cells $1 \dots 9$.

Extension to full 9×9 puzzle: This formulation would need to be repeated for each row and column of the full 9×9 board. Since the constraints on the rows and columns are essentially the same as the ones on the 3×3 boxes, this would be a fairly straight forward process and the same general structure could be used.

B: Binary Constraints [5 pt.]

Let $x_i = n | n \in [9]$ if cell i takes on value n (thus each x_i has domain $[9]$). Cell numbering is the same as part A (see above). The following binary constraints can be generated:

$$x_i \neq x_j \forall i \neq j, i \in [9], j \in [9]$$

These constraints can be extended to row and column constraints by adapting the indexing variables i, j such that they apply to the rest of the board.

This binary CSP formulation compares favourably with part A. There are far fewer total constraints thanks to the non-binary nature of the variables. I would use this formulation to program a solution to Sudoku.

C: Autolab Submission [12 pt.]

12/12 marks on Autolab.

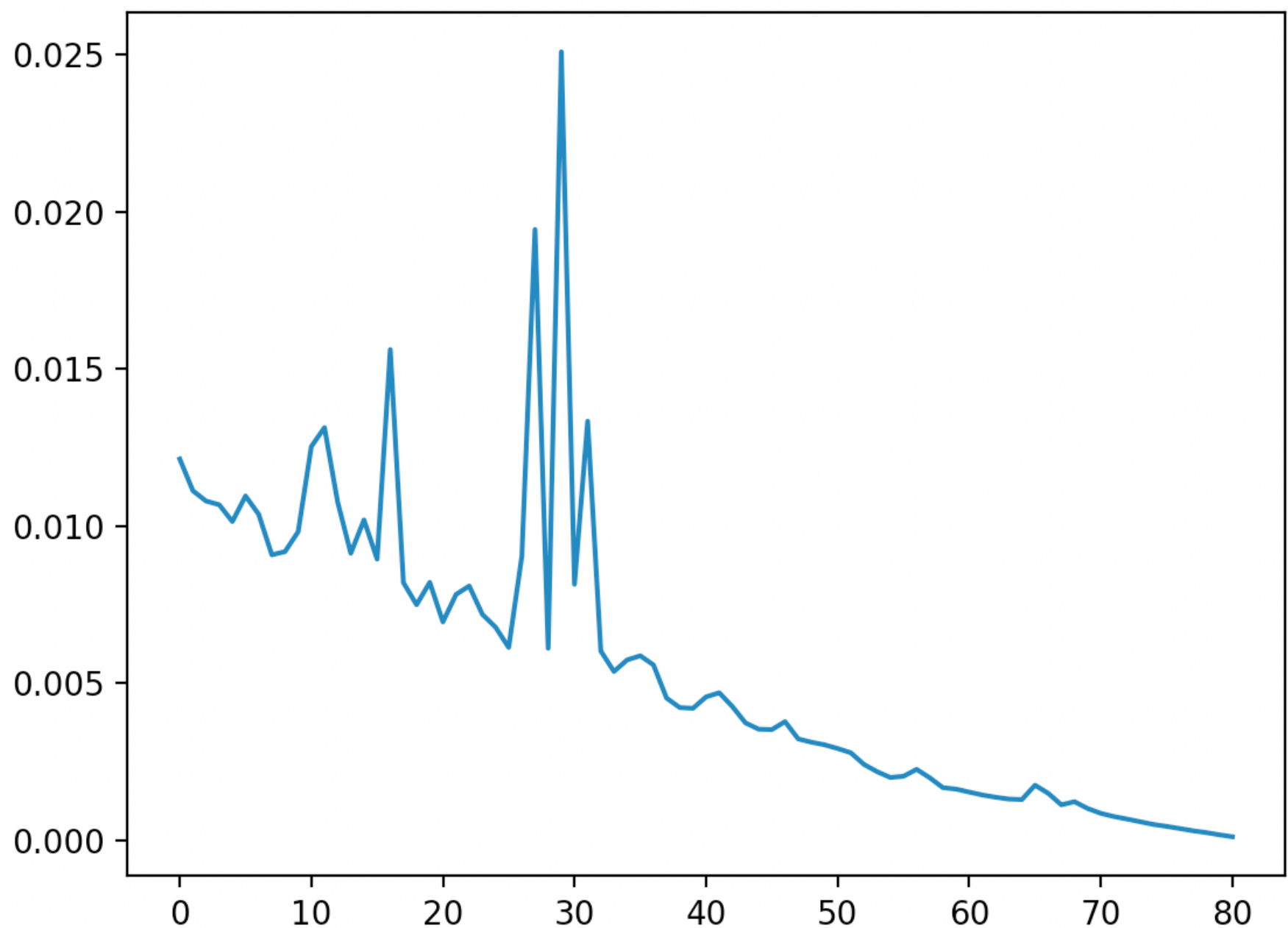
D: Algorithm Description [5 pt.]

For my algorithm, I implemented backtracking search with some additional heuristics and caching methods for faster solutions. In particular, I kept track of the remaining possible assignments for every free variable and focused on constraining the most constrained ones first. This helped the speed because more of the search tree could be paired down, and having nodes with low branch numbers higher in a tree is generally better for the overall tree size. Since solutions to hard Sudoku problems are rather sparse, the methods used for N-Queens (local search) would not have worked. Thus, backtracking search with the "most constrained first" heuristic was a good balance between working well and being reasonably quick to implement.

E: Difficulty of Sudoku [8 pt.]

Intuitively, more unassigned cells would necessitate a larger search tree to find a solution. However, in a completely unconstrained sudoku board, it would be fairly easy to come in with a strategy for assigning variables that makes the problem easier. Therefore, I would intuitively guess that difficulty peaks somewhere in the middle and is relatively easy with many knowns or many unknowns. This would also fit with what we had learned in class about constraint satisfaction problems in general.

This plot was generated from 10 averaged experiments on each number of unknowns. The x-axis shows the number of pre-constrained variables while the y-axis shows the amount of time it took for my algorithm to solve the problem. While there is a general downward trend, there is a clear peak in the middle.



Question 2: Sales Bot POMDP [35 pt.]

A: Belief-Space MDP Model [5 pt.]

Belief space model: We can create and update the belief state as follows:

$$b'(s') = \alpha P(e|s') \sum_s P(s'|s, a) b(s)$$

- Reward: $\rho(b) = \sum_s b(s) R(s)$
- Uninformed transition probability: $P(b'|b, a) = \sum_e P(b'|e, a, b) P(e|a, b)$
- Prediction of observation: $P(e|a, b) = \sum_{s'} p(e|s') \sum_s P(s'|s, a) b(s)$

Questions:

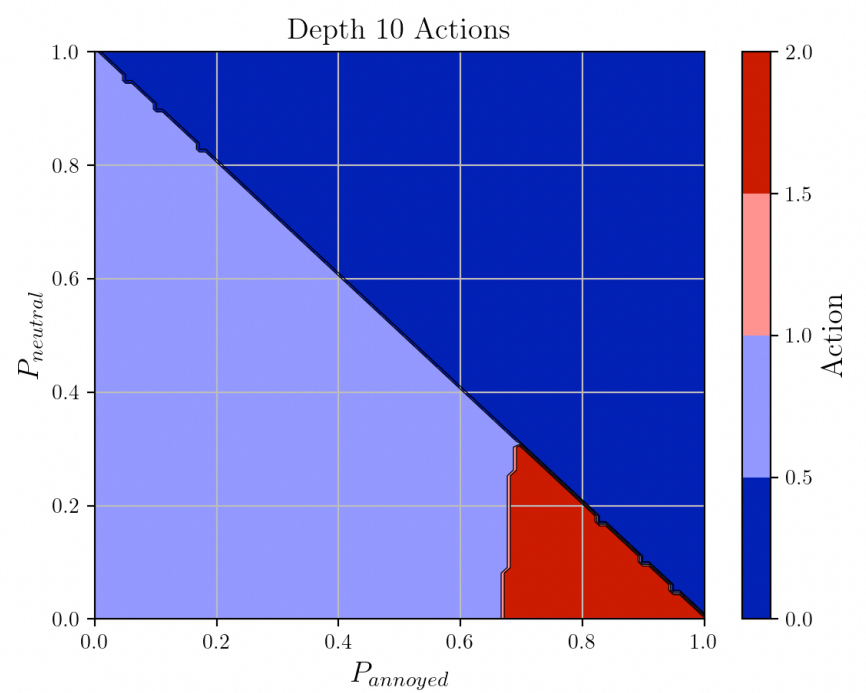
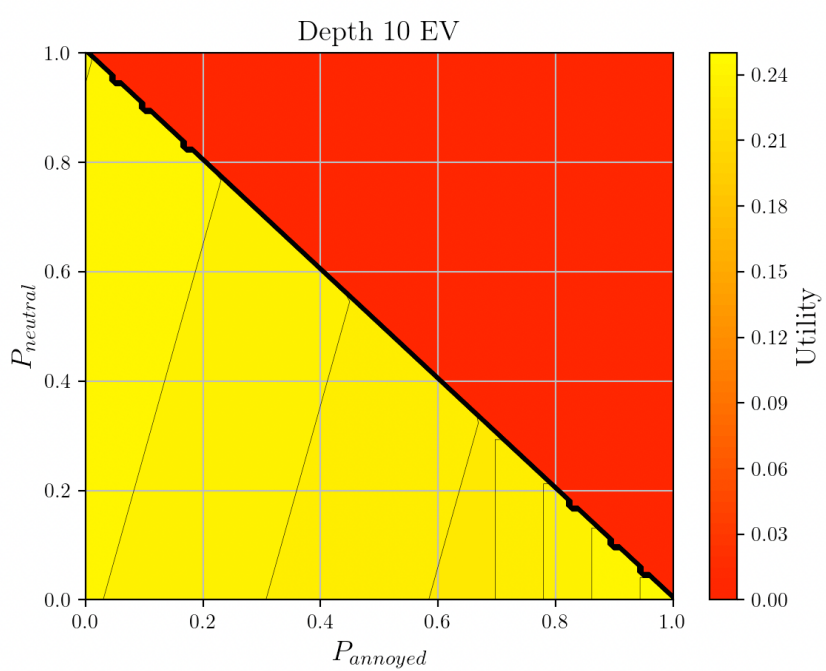
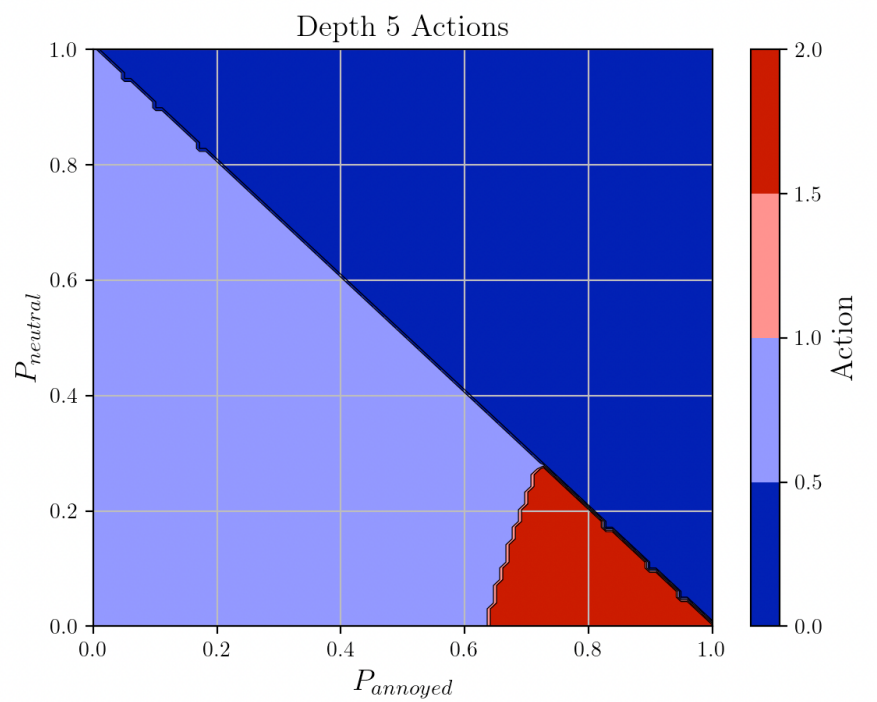
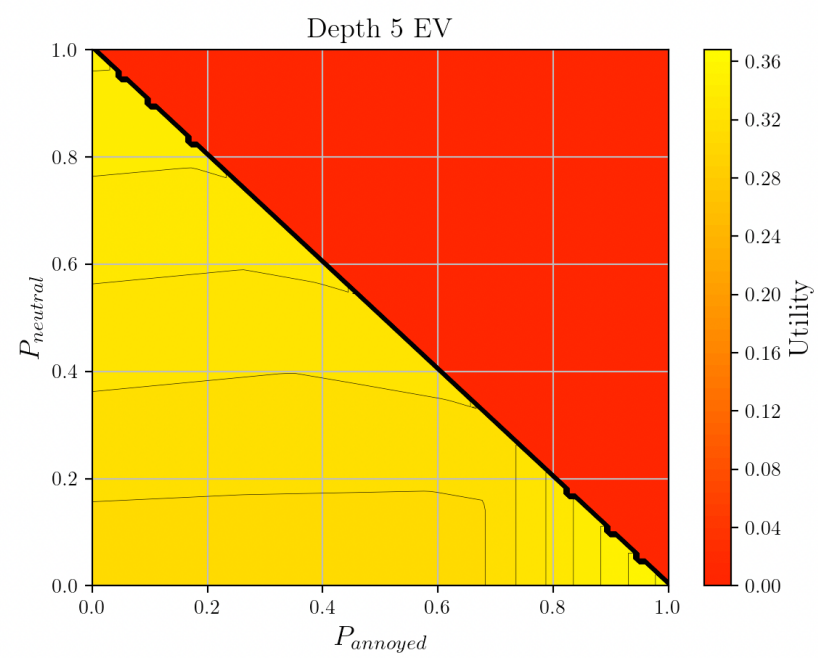
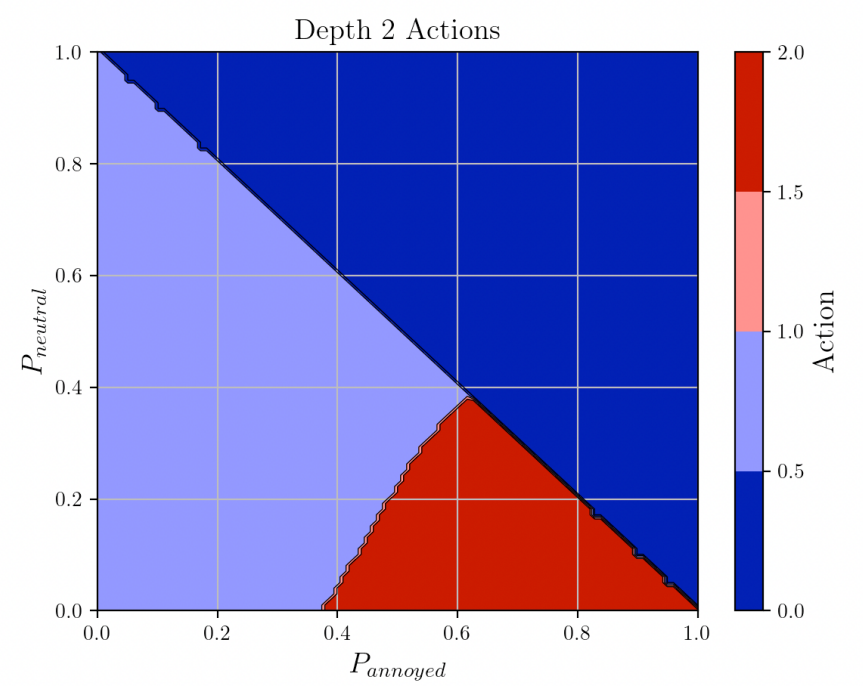
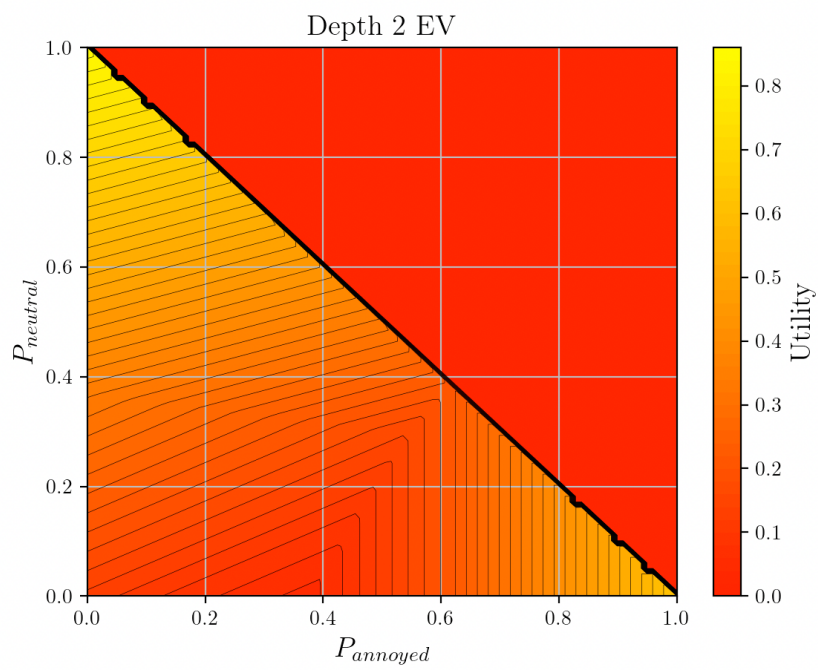
$$\begin{aligned}
2a) \quad 1) P(e=\text{hangry} \mid a=\text{agg}, b=\text{annoy}) &= \sum_{s'} p(e|s') \sum_s P(s'|s, a) b(s) \quad \rightarrow 1 \text{ iff } s=\text{annoy} \\
&= \sum_{s'} p(e|s') P(s' \mid s=\text{annoy}, a=\text{agg}) \\
&= \sum_{s'} p(e=\text{hangry} \mid s') P(s' \mid s=\text{annoy}, a=\text{agg}) \\
&= \sum_{s' \in [s]} P[0, s'] T[1, 1, s'] \\
&= 0.1
\end{aligned}$$

$$\begin{aligned}
2) P(e=1 \mid a=1, b=\begin{Bmatrix} 0.5:1 \\ 0.5:2 \end{Bmatrix}) &= \sum_{s'} p(e|s') \sum_s P(s'|s, a) b(s) \quad \rightarrow 1 \text{ iff } s=\text{annoy} \\
&= \sum_{s' \in [s]} P[1, s'] (0.5 P(s' \mid 1, 1) + 0.5 P(s' \mid 2, 1)) \\
&= \sum_{s' \in [s]} P[1, s'] (0.5 T[1, 1, s'] + 0.5 T[2, 1, s']) \\
&= \text{sum} \left(\begin{bmatrix} 0 \\ 0.4 \\ 0.2 \\ 0.2 \\ 0 \end{bmatrix} \left(0.5 \begin{bmatrix} 0.1 \\ 0.1 \\ 0.8 \\ 0 \\ 0 \end{bmatrix} + 0.5 \begin{bmatrix} 0 \\ 0.1 \\ 0.2 \\ 0.7 \\ 0 \end{bmatrix} \right) \right)
\end{aligned}$$

$$\begin{aligned}
&= \text{sum} \left(\begin{bmatrix} 0 \\ 0.06 \\ 0.1 \\ 0.97 \\ 0 \end{bmatrix} \right) \\
&= 0.23
\end{aligned}$$

$$\begin{aligned}
3) P(e=\text{neutral} \mid a=\text{apt}, b=\begin{Bmatrix} \frac{1}{3}:1 \\ \frac{1}{3}:2 \\ \frac{1}{3}:3 \end{Bmatrix}) &= \sum_{s'} p(e|s') \sum_s P(s'|s, a) b(s) \quad \rightarrow 1 \text{ iff } s=\text{annoy} \\
&= \sum_{s'} P(2|s') \sum_{s \in [3]} P(s' \mid s, 2) \frac{1}{3} \\
&= \sum_{s'} P[2, s'] \sum_{s=1,2,3} T[s, 2, s'] \cdot \frac{1}{3} \\
&= \text{sum} \left(\begin{bmatrix} 0 \\ 0.2 \\ 0.6 \\ 0.2 \\ 0 \end{bmatrix} \cdot \left(\frac{1}{3} \left\{ \begin{bmatrix} 0 \\ 0 \\ 0.5 \\ 0.5 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.2 \\ 0.5 \\ 0.3 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0.2 \\ 0.5 \\ 0.3 \\ 0 \\ 0 \end{bmatrix} \right\} \right) \right) \\
&= \text{sum} \left(\begin{bmatrix} 0 \\ 0.2 \\ 0.6 \\ 0.2 \\ 0 \end{bmatrix} \begin{bmatrix} 0.13 \\ 0.3 \\ 0.37 \\ 0.16 \\ 0 \end{bmatrix} \right) \\
&= 0.32
\end{aligned}$$

B: Visualizing Utility and Policies [15 pt.]



Trends:

- The amount of gradation in expected value for each belief space configuration decreases as the number of lookahead steps increases. This makes sense because, given enough steps, they would all converge to a steady state value.

- For some reason, the depth 2 expected value is low for high probability of "engaged". This could be an error with my implementation, but I have very closely examined it and was unable to find the source of this error.
- "Aggressive" action is universally not taken. This could be an error with my implementation, but I have very closely examined it and was unable to find the source of this error. It is possible that the uncertainty in measurement makes the aggressive tactics unwise.
- At each lookahead depth, it appears that being sure about one's state yielded the best results (i.e., a "concentrated" belief state). That said, there was significant diffusion as depth grew, resulting in a fairly uniform utility for all reachable belief states for depth 10.

C: Autolab [10 pt.]

Truthfully, I was unable to find why my code resulted in poor results. I re-wrote it several times and spent many hours on it. I hope that this effort is evident.

D: Algorithm Description [5 pt.]

I used the standard value-iteration scheme for POMDP's in continuous belief space. I recursively generated conditional plans. For plan p , $\alpha_p(s)$ represented the expected utility of executing the plan starting in state s . As the plans were recursively grown, these α_p values were updated as follows.

Base Case: For a single-move plan,

$$\alpha_p(s) = R(s) + \gamma \left(\sum_{s'} [P(s'|s, a) R(s')] \right)$$

Recursive Definition: Action p adds first action a . $p.e$ is the depth $d - 1$ plan that is executed if emission e is observed.

$$\alpha_p(s) = R(s) + \gamma \left[\sum_{s'} P(s'|s, a) \sum_e P(e|s') \alpha_{p.e}(s) \right]$$

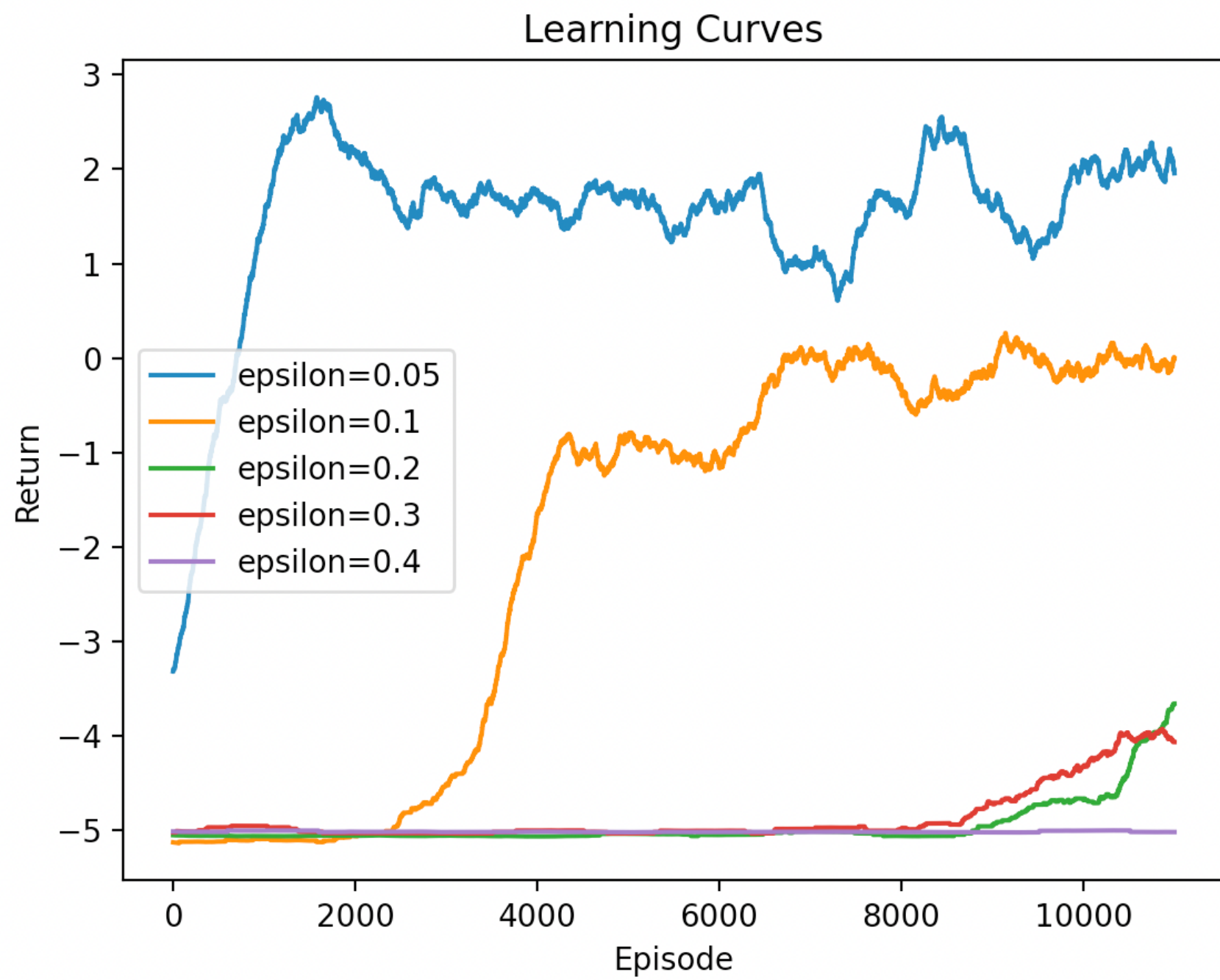
Since the number of conditional policies grow incredibly fast with depth, I used an approximate sampling method as a substitute for the linear programming method commonly used for eliminating dominated strategies. Since the ranges for each index of the belief vector are well defined (positive, simplex constraint), I took a fine sampling (step size = 0.01) through this region on each axis. I kept track of which entries in U were NOT observed via `argmax(A @ b)` where A is a matrix composed of row vectors $\alpha_p(s)$ and b is the iterated belief vector. These entries not observed in the argmax were very likely dominated by the rest and were therefore removed because they never dominated any others in the relatively fine sampling. This approximate method did an excellent job of reducing the computational complexity of the task.

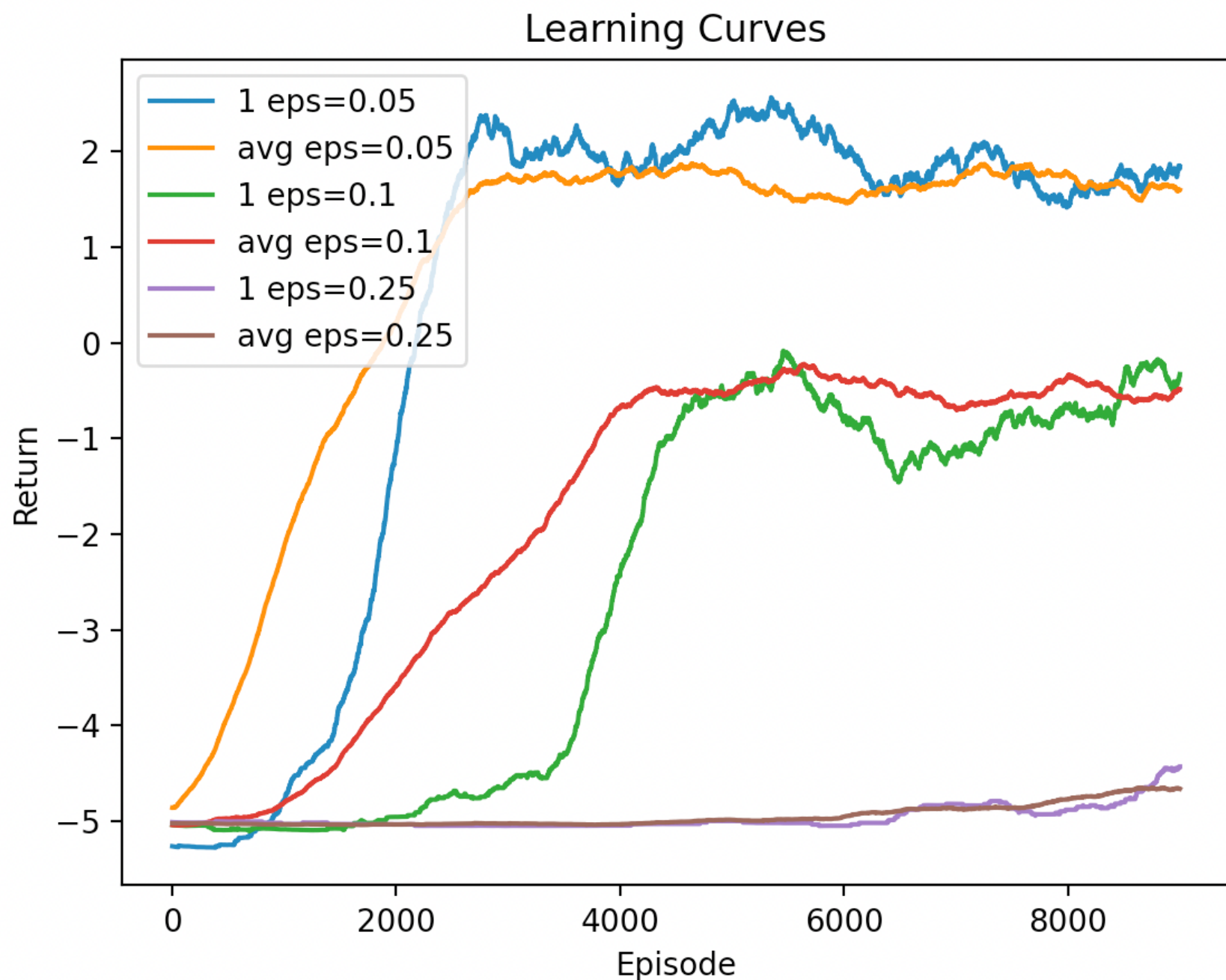
Once the desired depth was reached, the conditional policy was selected again based on $\max_p \alpha_p \cdot b(s)$, an operation easily implemented as above in a matrix multiplication. From this maximizing conditional policy p , the next action was selected. This was repeated at each time step.

While this algorithm did not seem to achieve the desired results, I would point out that it was written in a rather efficient manner.

Question 3: Learning to Drive [30 pt.]

A: Reinforcement Learning Algorithm [10 pt.]





Note that `avg eps` labelled items had the average taken on 10 separate runs of the experiment for that value of epsilon.

Trends: Lower epsilon seems to do much better in terms of both rate of convergence and steady state value. This makes sense, because small deviations toward an obstacle yield high risk with this MDP. Reducing that risk by having a small epsilon value makes a lot of sense. Otherwise, they all look like they have roughly the same noise/variance and appear to follow the average for the epsilon value reasonably closely.

Any epsilon above 0.2 tended to work extremely poorly. This intuitively makes sense because it corresponds with "reckless driving". Even if the optimal policy were somehow deduced, there is a 20% chance of the driver taking a random move on top of the existing stochasticity in the Markov process.

B: Autolab [15 pt.]

15/15 marks on Autolab.

C: Algorithm Description [5 pt.]

I implemented a standard Q-learning scheme with an ϵ -greedy approach. Results from part A heavily impacted my decisions for hyperparameters, particularly with respect to the proper value of ϵ . Since $\epsilon = 0.05$ yielded faster convergence to a higher return value, it made the most sense to choose, especially for a dynamic environment. As I experimented with the autograder, it turned out that $\epsilon = 0.01$ worked even better (15/15).


I chose not to implement a decaying exploration probability because of the fact that the map would change over time with new obstacles. Evidence from local testing also suggested that this problem was an exercise in exploitation (given how extremely low ϵ values performed better). This intuitively makes sense — once you "learn to drive" in this simplified world, you should probably not explore too much around unless you are in danger of crashing. Plus, the Markov process already introduces some quasi-exploration. Overall, it makes the most sense to learn quickly from mistakes/changes in the environment and not explore too much. Based on evidence from part A and performance in part B, this is achieved well with low epsilon.

My other parameters $\alpha = 0.1$ and $\gamma = 0.9$ were fairly straight forward and required little to no tuning.

I also chose to use Q-learning because of my research experience in this domain. I actually trained a Q-learning based algorithm on EEG data to incentivize certain brain states last year! You can check out the paper here:

A Novel Approach to EEG Neurofeedback via Reinforcement Learning

Since the invention of EEG brain scanning technology, cognitive response modeling and brain state optimization has been a topic of great interest and value. In particular, applications for improving musical therapy via neurofeedback have shown promise for brain state optimization.

 <https://ieeexplore.ieee.org/document/9278871>

IEEE Xplore®

Thanks a lot for a great year!