Risk-averse Batch
Active Inverse Reward Design

Overview

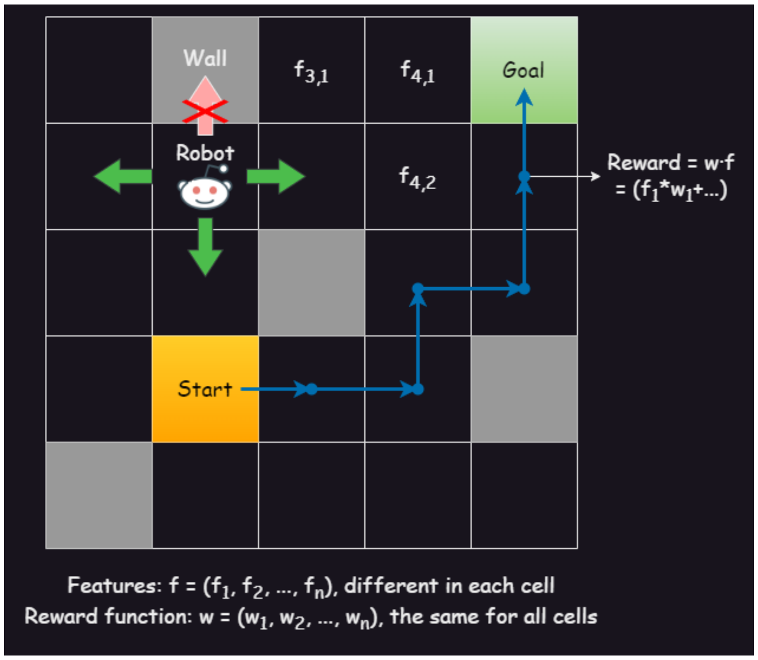
The project is an improved version of Inverse Reward Design and Active Inverse Reward Design, that computes the probability distribution over the true reward function in batches of test data and a risk-averse policy based on it. It tries to counteract the problems of goal misgeneralization and reward misspecification, and increase the safety of AI systems, by implementing the ability to learn from real-life environments, not only in training, make the decisions that are most certain using the information it has gained, and learning from the behavior that humans want it to have.

I used and modified part of the AIRD code. The GitHub repository for the code of my project is: RBAIRD.

Terminology/Setup

The environment I used is a gridworld, which is a grid with dimensions 12x12, and it contains:

- A robot, which can move up, down, right, and left in adjacent cells.
- A start state, from which the robot starts moving.
- Some goal states, which when the robot reaches it stops moving.
- Some walls, from which the robot cannot pass through.
- All the other cells, in which the robot moves.


Wall
$f_{3,1}$
$f_{4,1}$
Goal
Robot
$f_{4,2}$
Reward = w·f
= (f_1*w_1+...)
Start
Features: f = (f_1, f_2, ..., f_n), different in each cell
Reward function: w = (w_1, w_2, ..., w_n), the same for all cells

All the cells contain a vector of *features* ($f_1, f_2, ..., f_n$), which are used in calculating the reward in that state.

The *reward* is calculated using a *reward function*, which is a vector of *weights* ($w_1, w_2, ..., w_n$), which is the same along all states.

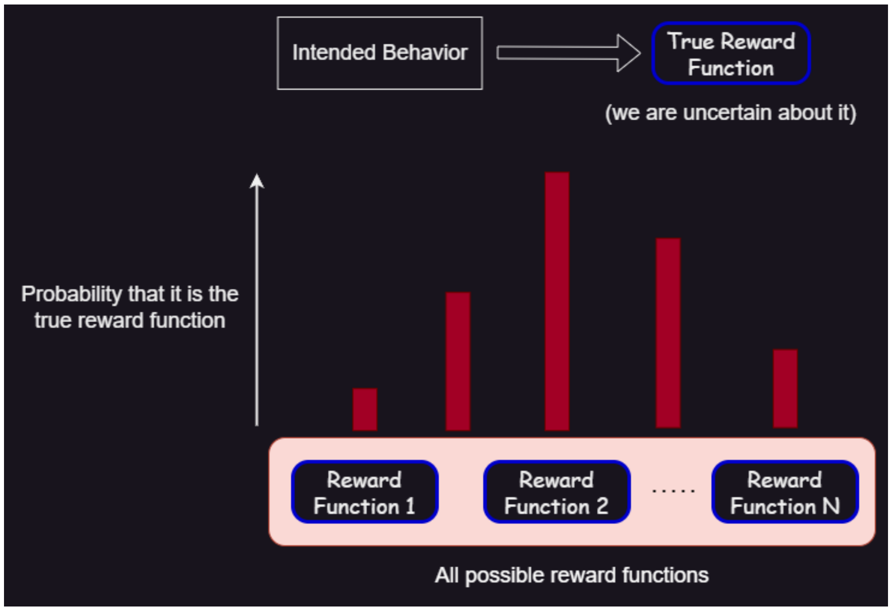
The reward in a state with features $f = (f_1, f_2, ..., f_n)$ and weights $w = (w_1, w_2, ..., w_n)$ is their dot product $f \cdot w = (f_1*w_1 + f_2*w_2 + ... + f_n*w_n)$. We also have a *living reward*, that is used to incentivize shorter routes, so we subtract it from the dot product.

A *policy* is a map from the states (x, y) to the action (north, south, east, west) in the environment. An *agent* controls the robot and moves it in specific directions, using a predetermined policy, in order to maximize the total reward in a trajectory of the robot (the trajectory is the set of states the robot has visited in chronological order until we stopped it or it reached a goal)

In both papers and my project, we try to find the reward function that best represents the intended behavior of the agent, which we call the *true reward function*. This function is an element of a big set that is called the *true reward space*, which contains all the possible true reward functions.

However, because we are unsure of that perfect reward function, in IRD we start with a human-made estimation which is a *proxy reward function*, which is an element of the *proxy reward space* (in AIRD we only have the proxy reward space).

The goal of the papers and the project is to find a probability distribution over all the rewards in true reward space: for each element of it, we have the probability that it is the true reward function, based on the behavior they incentivize in the training environment.


Intended Behavior
True Reward Function
(we are uncertain about it)
Probability that it is the true reward function
Reward Function 1
Reward Function 2
.....
Reward Function N
All possible reward functions

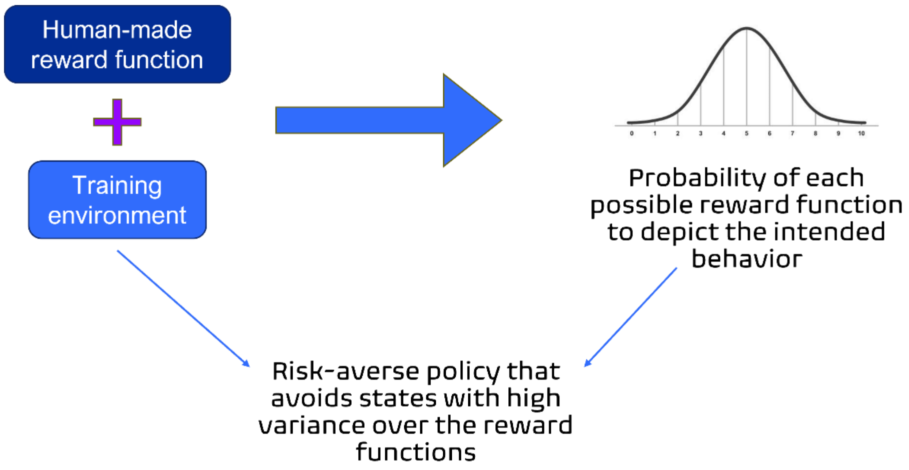
The *feature expectations*, given a reward function and an environment, is the expected sum of the features in a trajectory derived from an optimal policy given that reward function.

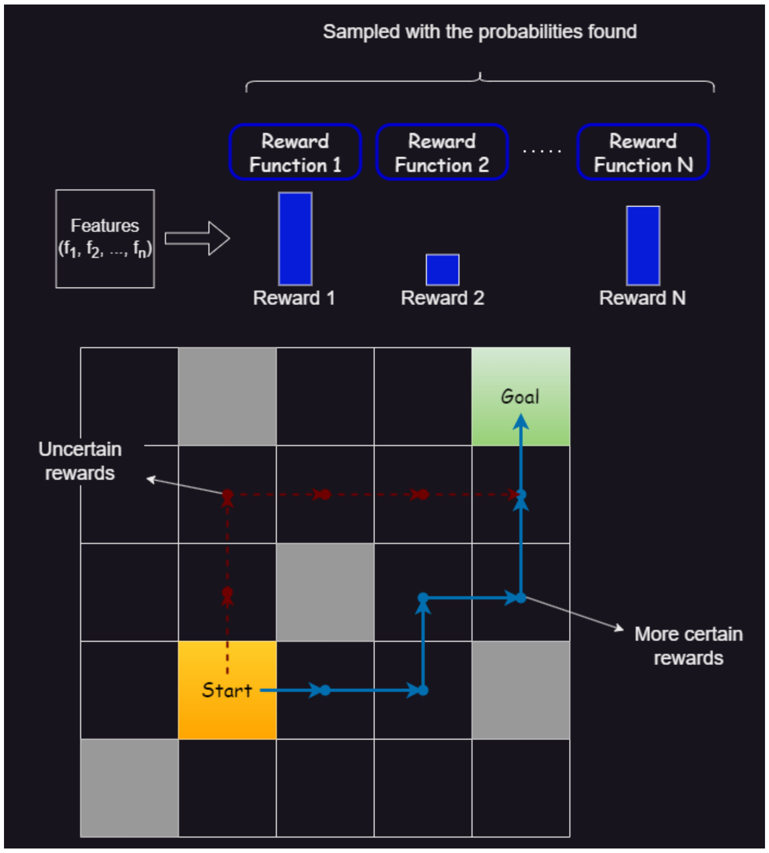
In both the total trajectory reward and feature expectations, we apply a discount γ (it may be 1), such that the next feature or reward is first multiplied by γ^i, where i increases by 1 each time the robot moves.

Existing approaches

Inverse Reward Design (Hadfield-Menell et al., 2020)

Given the true reward space and a proxy reward function, it approximately computes (using Bayesian inference) the probability distribution over the true reward function.


Human-made reward function
Training environment
Probability of each possible reward function to depict the intended behavior
Risk-averse policy that avoids states with high variance over the reward functions



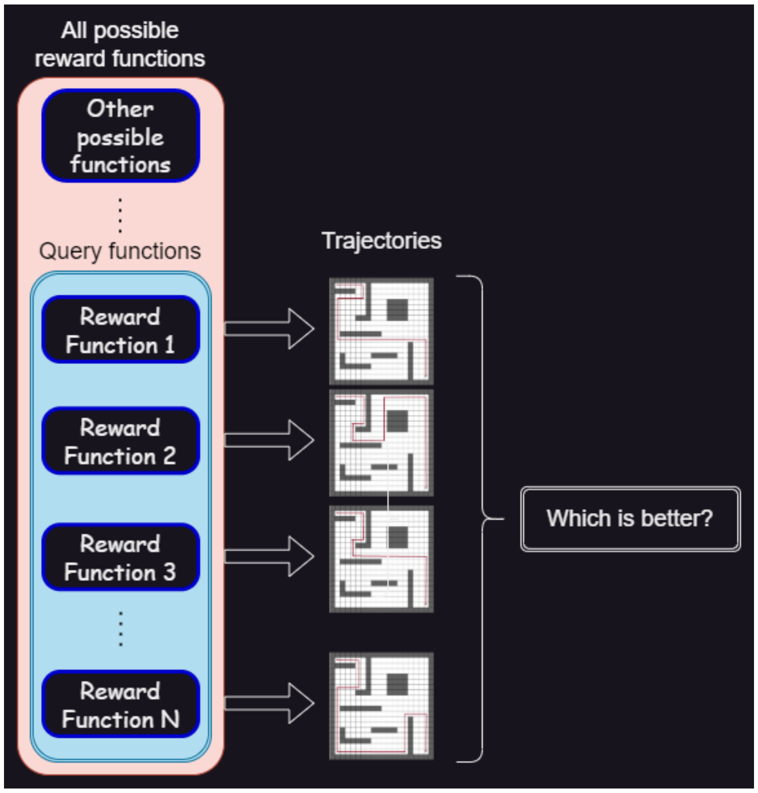
It then computes a risk-averse policy, that takes actions so that the distribution of the rewards in that state, using a set of weights sampled with the precomputed probabilities, and the features of that state, has low variance (the reward function distribution is very certain about that state). The risk-averse policy is computed in various ways:

- Maximizing the worst-case reward, per state or trajectory.
- Comparing the reward of each state with the reward of some baseline features used as a reference point.

Active Inverse Reward Design (Mindermann et al., 2019)

It is given the true reward space, and a proxy reward space with some proxy reward functions (they may be the same set). It starts with setting the wanted probability distribution (for the true reward function) as a uniform distribution (all the functions are equally probable since we don't know anything about the true reward function).

Then, it continuously asks queries to the human, in order to update that probability distribution and make it more certain about the true reward. A *query* is defined as a small subset of the proxy reward space. The answer to the query is a single element of that subset, which the human believes incentivizes the best behavior, compared to the other elements of the query (it compares suboptimal behaviors, not the optimal one).

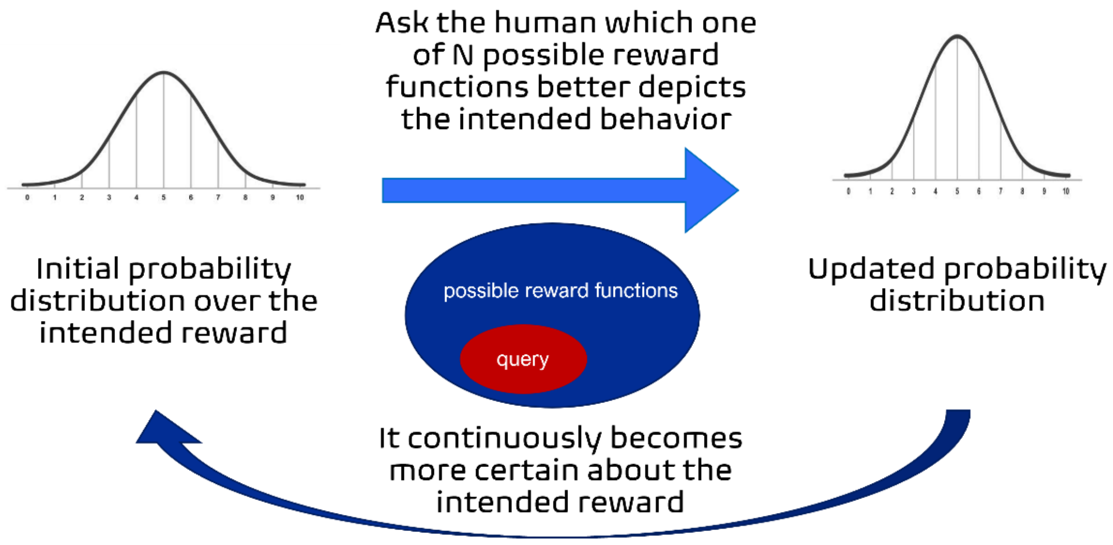


After each query, it uses Bayesian inference to update the probability distribution based on the answer to that query. To do that, it uses a [Q learning] planner that optimizes trajectories, in the training environment, given each element of the query as the reward function. It then computes the feature expectations of these trajectories and uses these and the answer to the query to update the probabilities.

The queries are chosen such that the expected information gain from the answer to the query is maximal. The information gain is measured using various metrics, one of which is the entropy of the probability distribution over the true reward function.

There are multiple ways this selection can be done, but the one I used on my project, as it is more efficient and less time-consuming, is the following: as long as the query size is less than a predetermined constant (initially it is empty), we take a random vector of weights, and then take gradient descent steps such that the information gain when these weights are the answer to a query is maximal.

After each query, the performance is evaluated by running the Q learning planner, using the true reward function and the average of the probability distribution computed, and measuring their difference in the training environment and some test environments.


Ask the human which one of N possible reward functions better depicts the intended behavior
Initial probability distribution over the intended reward
possible reward functions
query
Updated probability distribution
It continuously becomes more certain about the intended reward
0 1 2 3 4 5 6 7 8 9 10

My approach

We start with the same given data as AIRD: the true reward space and the proxy reward space.

I also define a query, and update the probabilities using Bayesian inference, the way AIRD does it (I used the AIRD code for query selection and Bayesian inferences).

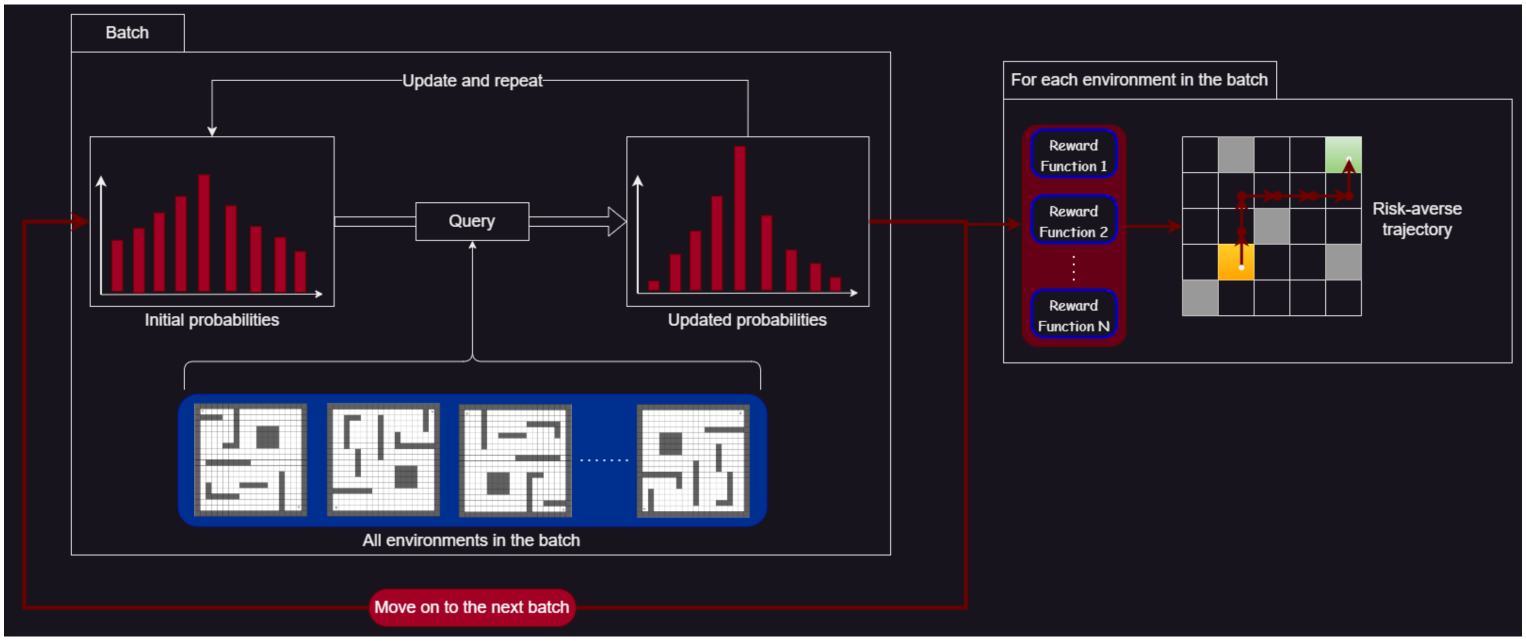
There are some batches, with a specific number of environments in each batch. There is a constant (big) set of test environments for evaluation of the performance. I also keep track of the probability distribution over the true reward function, which initially is a uniform distribution.

I also made two planners, using Q learning, that have as an input a set of weights:

- The non-risk-averse (unsafe) one, which has as the reward the average of the rewards on the state with each weight sample.
- The risk-averse one, which penalizes the variance of the rewards computed using the weight sample and the state's features, in two ways:
    - By taking the worst-case reward
    - By subtracting the variance multiplied by a coefficient

For each batch, I do the following:

1. Repeat (for a constant number of iterations) the query process of AIRD: I find the query from which the information gain of the probability distribution is maximal. The difference with the previous query process is that for each reward function in the query, for each environment in the batch:
    a. I take the initial probability distribution.
    b. I answer the query to that specific environment.
    c. I apply the reward function to the unsafe planner and get the feature expectations.
    d. Use the feature expectations to update the probability distribution.
    e. Update the initial probability distribution using this inference and move on to the next environment in the batch.
2. Update the initial probability distribution, so it will be transferred to the next batch.
3. Sample a set of weights from the distribution over the reward function, and compute a risk-averse policy using the respective planner.



Evaluation

After each batch, for each test environment, I computed the total reward of the risk-averse planner and that of the unsafe one. I also computed the optimal reward by giving the planner the exact true reward function.

Then, I computed the following metrics:

- Test regret = optimal planner – unsafe planner
- Risk-averse regret = optimal planner – risk-averse planner
- Test and risk-averse variance = sum of variances of the rewards in trajectory computed using the unsafe and risk-averse planner

I then took the average of the above metrics over the test environments.

I also plotted the trajectories of both planners in each environment of the batch.

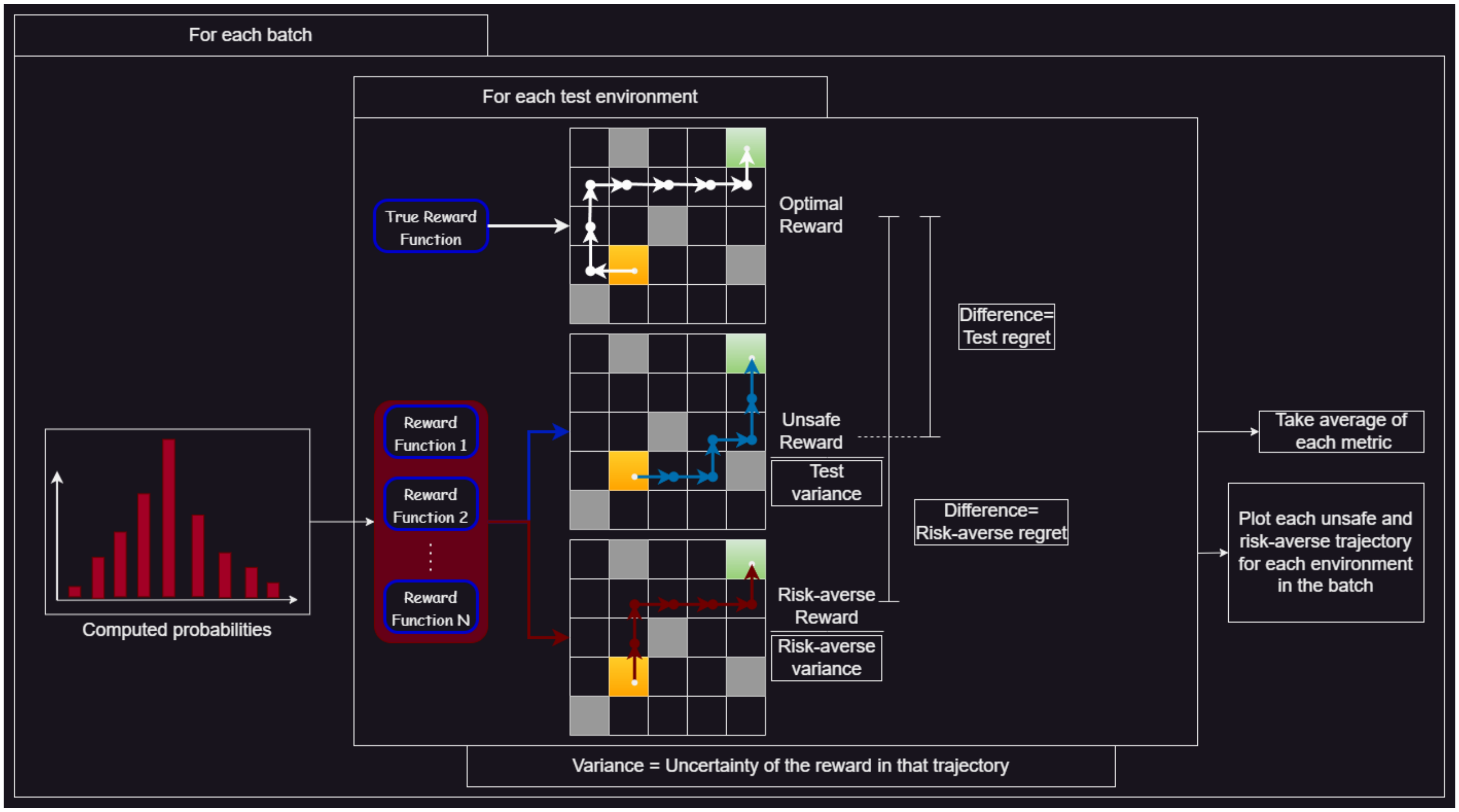
The X-axis of the graphs is the number of total Bayesian inferences (updates of the reward probabilities), which is the number of batches*number of queries for each batch*number of environments in each batch.

I performed experiments, by varying:

- Number of batches
- Number of environments in each batch
- Number of queries for each batch
- The method used for risk-averse planning:
    - Subtracting the variance with coefficient 1

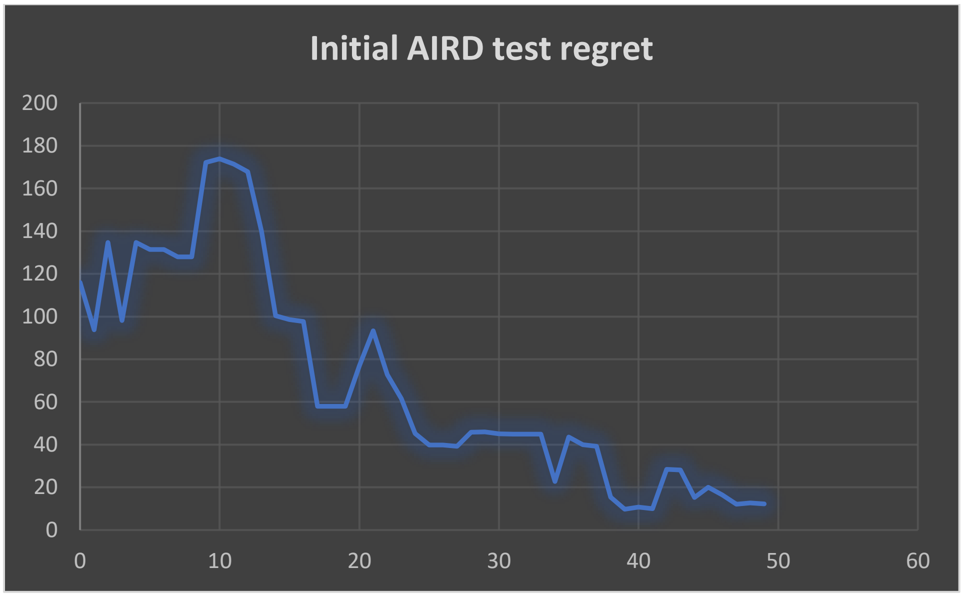
- Subtracting the variance with coefficient 100
- Worst-case with 10 reward samples
- Worst-case with 100 samples

I also collected data on the AIRD paper’s method, for comparison.


For each batch
For each test environment
True Reward Function
Optimal Reward
Difference= Test regret
Computed probabilities
Reward Function 1
Reward Function 2
Reward Function N
Unsafe Reward
Test variance
Difference= Risk-averse regret
Risk-averse Reward
Risk-averse variance
Take average of each metric
Plot each unsafe and risk-averse trajectory for each environment in the batch
Variance = Uncertainty of the reward in that trajectory

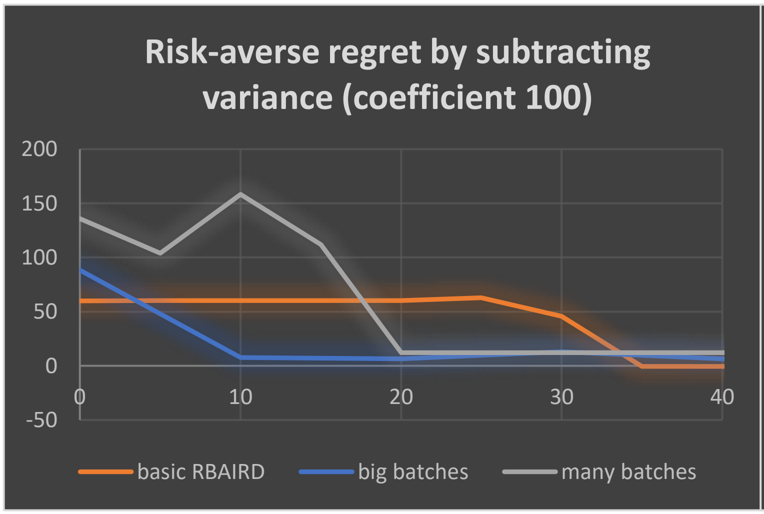
Results

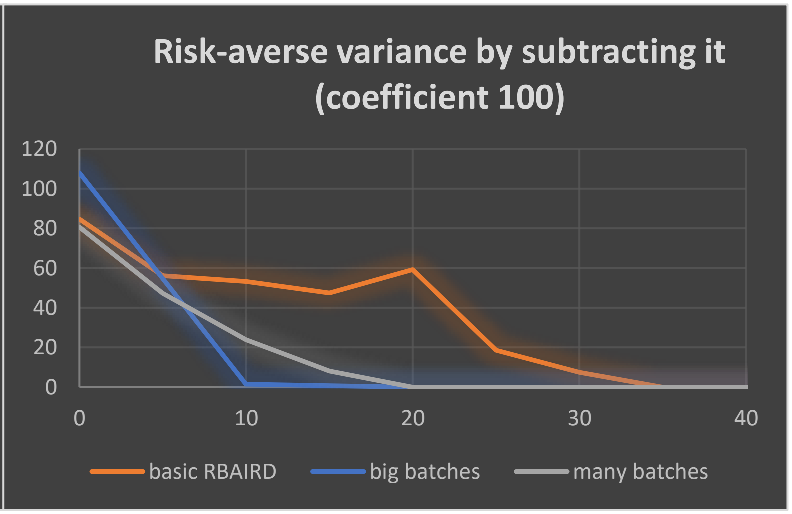
Initial AIRD



In the initial AIRD paper, using the same query-choosing method as I did, the performance approaches optimal after ~50 queries (but it never becomes optimal, a single environment isn't enough to understand the true reward function to its fullest)

RBAIRD performance

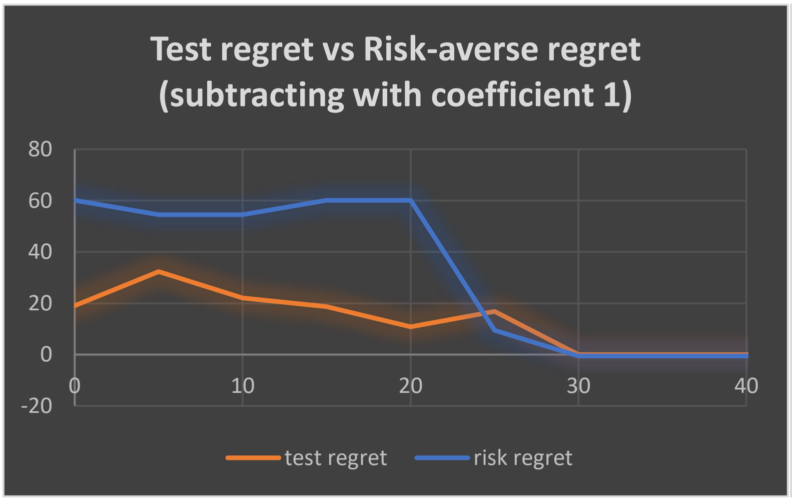


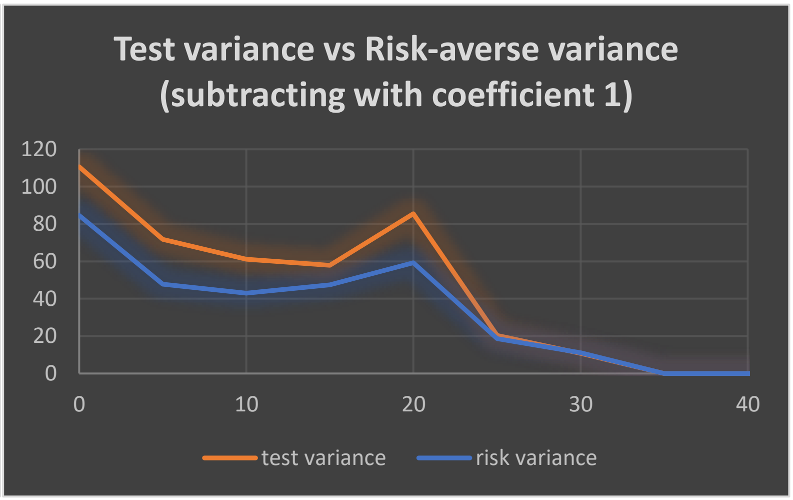


Using RBAIRD, the total number of inferences is lower than AIRD (~30), and it almost achieves optimal performance. This shows that there is much bigger information gain when combining behavior from many environments instead of many queries on just one environment (each environment can highlight different aspects of the policy of a specific reward function)

The number of queries can be even lower with big batches (~1-2 queries if we have 10 environments in each batch), so less human intervention is needed (even with answering the query to each environment, we have at most 15-20 answers needed, but gain perfect performance)

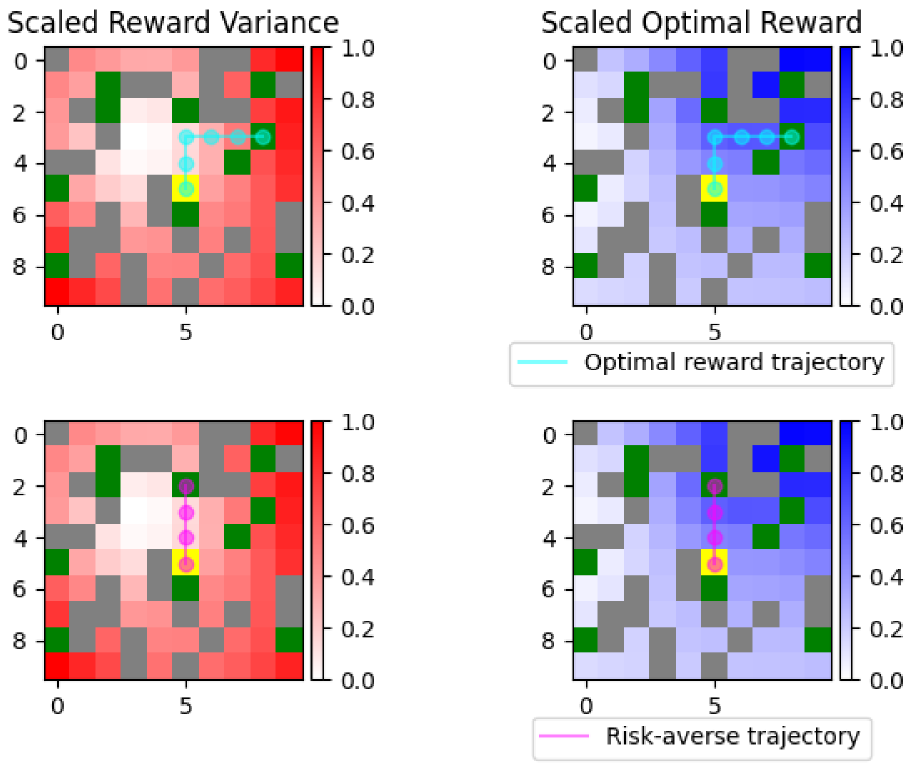
Risk-averse vs unsafe performance





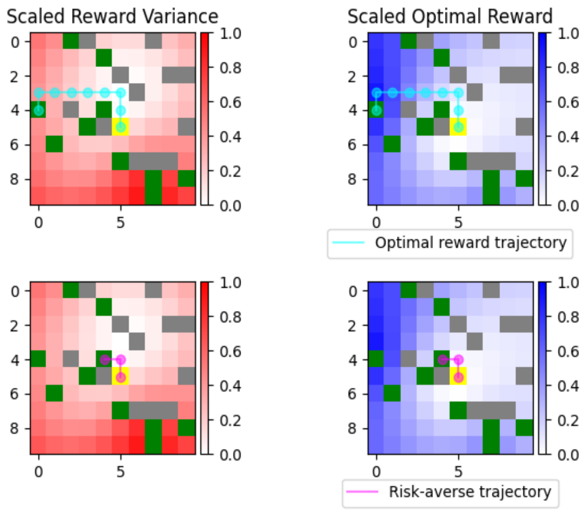
When we are still uncertain about the true reward function, risk-averse performance is worse than the unsafe one. However, the risk-averse planner has a constantly lower variance than the non-risk-averse one. Both performances become optimal at the same time, when the true reward function is found.

Trajectory Comparison

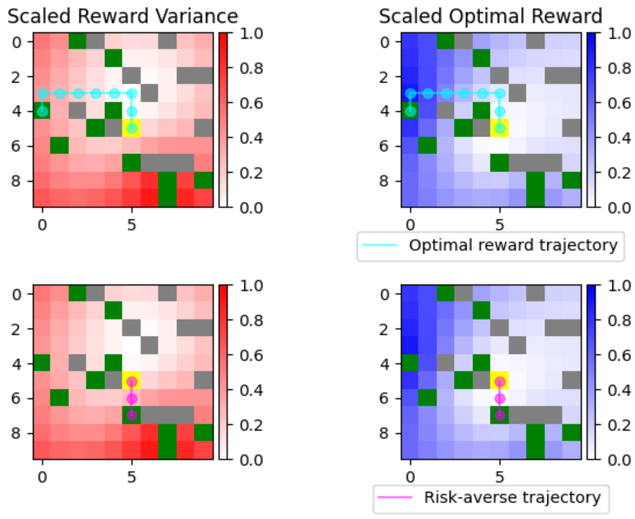


The risk-averse planner chooses a trajectory that is safer than the non-risk-averse one, while still reaching a goal and collecting as big of a reward as possible.

Here is a comparison of different risk-averse methods in the same environment:



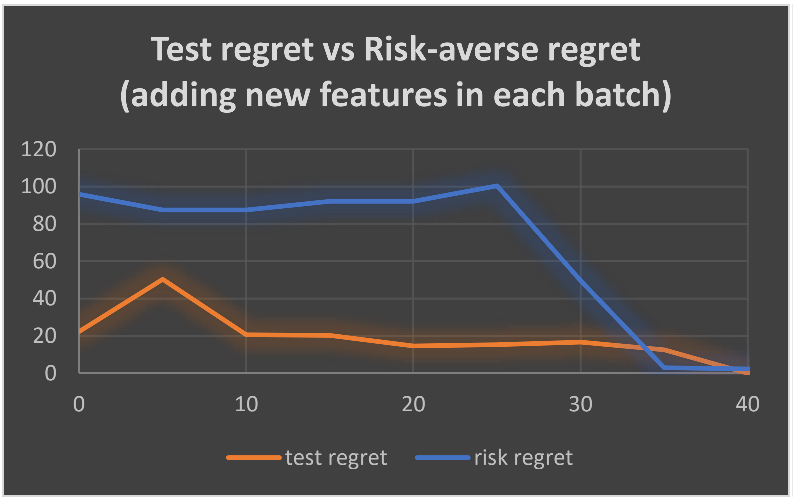
Subtracting with coefficient 100

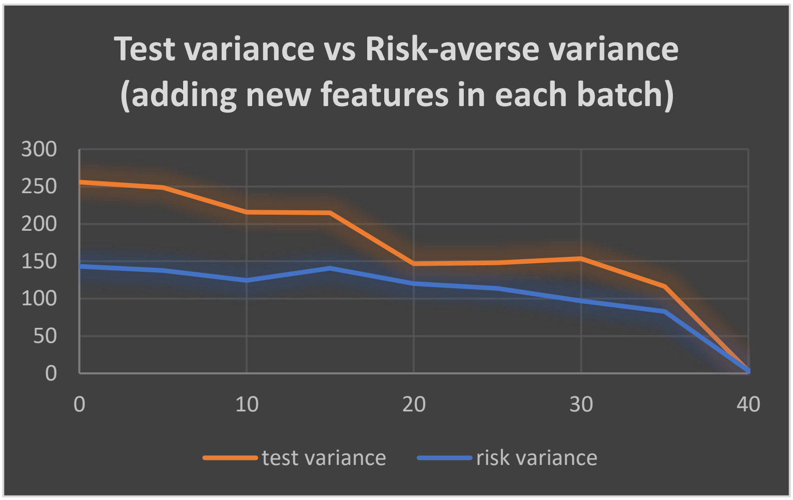


Worst-case with 100 samples

Adapting to new environments

I performed an experiment where, instead of having all the features available in the training environments from start, I added a number of them, that were 0 in all the environments of the previous batches, to each new batch.

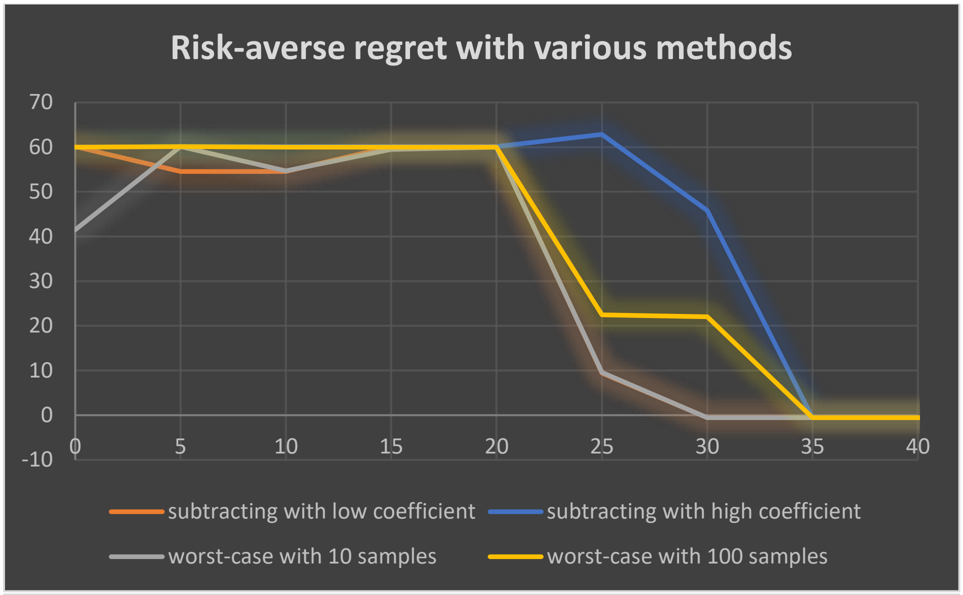


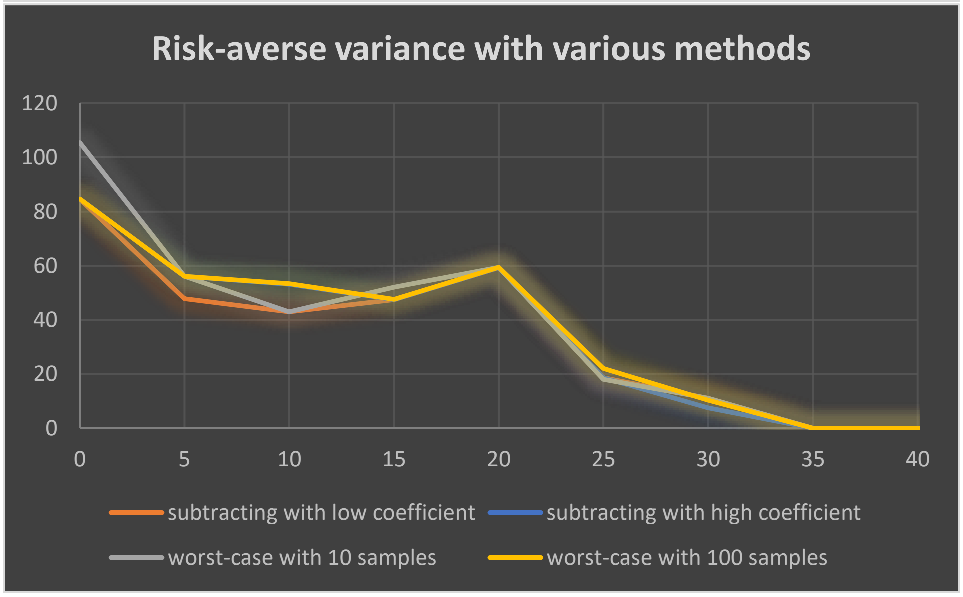


The total number of inferences was still ~40, similar to when we had all the features available from the beginning, and lower than AIRD, and with even only 2 queries per batch. This shows that RBAIRD is able to adapt to unforeseen scenarios quickly. Also, the risk-averse planner had about half the variance of the unsafe one, noting its importance on new environments and the safety it offers on them. AIRD didn't have the capability to adapt to unknown environments, since it was only trained in one environment, and it ignored the possibility of new features appearing, often making risky decisions.

Comparing risk-averse methods

Here I plot the risk-averse regret and variance using different reward methods (subtracting with low coefficient and high coefficient means subtracting the variance with coefficients 1 and 100 respectively):





It seems subtracting with coefficient 1 is the most efficient method, both regarding the regret and the variance (this is without comparison to other more sophisticated methods).

Next steps

Other query selection methods

In this project, I only used one query selection method from those that are used in the AIRD paper, the one where the query is increased by size one each time, after randomly sampling some weights and optimizing them. I also only used queries of size 5.

However, other query methods are more efficient in the original paper, but more computationally expensive, so I wasn't able to run them on my PC. I will try to optimize them and integrate them into RBAIRD (my approach), in order to compare their efficiency and performance, and maybe achieve better results.

More efficient risk-averse planning method

Until now, I have only tried using some simple, per-state, risk-averse reward functions, that simply take the worst-case scenario or penalize the variance in some way. However, they lead to the so-called *blindness to success*, and there are more efficient, but more complicated, methods that improve that aspect and gain performance-wise, and possibly reduce the expected variance even more. I will try to implement these, and evaluate their performance, regarding the maximum reward and the uncertainty of the actions.

Answering each query in multiple environments at once

What the code currently does, is it makes a single query in each iteration, that is optimized along all environments, but then gets an answer for each environment (total number of answers = number of environments in each batch, but we have the same query in each environment). This answer depicts which function performs better in that specific environment, relative to the true reward function.

What would be more efficient, is to answer each query once for all environments, showing which reward function shows better behavior in general, not specific performance to that environment. This would greatly reduce the number of query answers needed, and maybe make it less computationally expensive. However, then the human has a more difficult task to do, and the selection is a bit ambiguous, since there is no clear measure of "better behavior" about a reward function, since each one incentivizes different behaviors to a different extent.

Making an interactive query-answer environment

Currently, the program does not actively ask for a human to answer each query, but it is given the true reward function and predicts the expected answer based on that. Something that would demonstrate how a human would be able to judge and compare different reward functions, is to make an interactive environment where the human can answer the query, given various types of information.

It could show the trajectories that are computed based on each reward function, in a single or multiple environments (maybe choose the environments where their behavior would differ a lot, or where a single feature is prevalent in each environment). Also, it could provide various metrics about the performance of the reward function regarding various features, or some other high-level patterns that are observed, but this is a bit ambiguous and related to mechanistic interpretability. It could also provide the risk-averse and the unsafe performance of the planner in various steps of the query process, in order for the human to be able to judge what behavior the agent adopts (as a safety measure)

Related work/References

Casper, S., Davies, X., Shi, C., Gilbert, T. K., Scheurer, J., Rando, J., Freedman, R., Korbak, T., Lindner, D., Freire, P., Wang, T., Marks, S., Segerie, C.-R., Carroll, M., Peng, A., Christoffersen, P., Damani, M., Slocum, S., Anwar, U., … Hadfield-Menell, D. (2023). *Open problems and fundamental limitations of reinforcement learning from human feedback*. arXiv. https://doi.org/10.48550/arXiv.2307.15217

Hadfield-Menell, D., Milli, S., Abbeel, P., Russell, S., & Dragan, A. (2020). *Inverse reward design*. arXiv. https://doi.org/10.48550/arXiv.1711.02827

Langosco, L., Koch, J., Sharkey, L., Pfau, J., Orseau, L., & Krueger, D. (2023). *Goal misgeneralization in deep reinforcement learning*. arXiv. https://doi.org/10.48550/arXiv.2105.14111

Mindermann, S., Shah, R., Gleave, A., & Hadfield-Menell, D. (2019). *Active inverse reward design*. arXiv. https://doi.org/10.48550/arXiv.1809.03060

Specification gaming: The flip side of AI ingenuity. (n.d.). Retrieved August 22, 2023, from https://www.deepmind.com/blog/specification-gaming-the-flip-side-of-ai-ingenuity