Heuristics in Reinforcement Learning

By Bennett Brain

Introduction

Reinforcement Learning (RL) is a very powerful tool that can be adapted to nearly any problem that can be formulated with a state space representation. Many methods are mathematically proven to eventually converge on an optimal solution, while others may do so most of the time, and faster, even if convergence is not technically guaranteed. However, depending on the shape and nature of the environment, convergence can take a very long time. Even with simple toy environments like Mountain Car (explored more in detail later), arriving at any solution can take over an hour with a decently powerful computer. Thus, I considered the potential inclusion of Heuristics- i.e. whether a human can look at a problem, think of some basic guidelines to feed to an RL agent, and then hopefully that helps guide early exploration so the agent learns faster in the early stages. I implemented heuristics using an adaptation of an ε-greedy policy, and tested it thoroughly on two environments with tabular methods and did one-shot for three heuristics on each of three environments with Deep RL.

Related Work/Inspiration

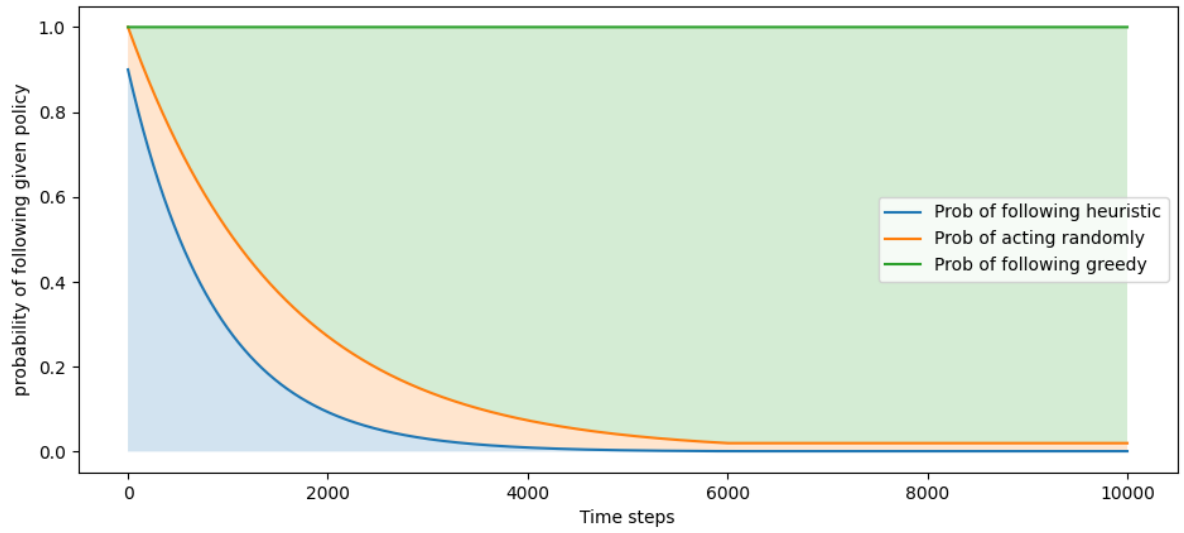
I was initially inspired to do this project by reading an RL paper about RLang (Rafael et al, 2022), which described creating a domain-specific language for RL that could translate environment knowledge into guidance for various types of RL agents. They tested on environments like a simple Gridworld with lava, a 2-d Minecraft, and the Lunar Lander environment, finding that including heuristics generally did speed up early learning while achieving the same near-optimal long-term results. I initially wanted to use RLang for my project to see if I could recreate their work and apply it to new environments, but after attempting to use it for about 10 hours it became clear that I would have to spend most of my time bug fixing RLang as opposed to designing and testing my own experiments. As such, I decided to implement heuristics and heuristic-guided RL agents myself, but I still credit the paper as a large inspiration for this project.

Methods

To create an agent that uses a heuristic to guide its learning, I iterated on the basic idea of ε-greedy. Simply put, ε-greedy works by learning state or state-action values (I used state-action, i.e. Q values) over time by acting in the environment, taking in feedback, and updating Q(S,A) by weighting the sum of the current value and some step size multiplied by the learned new value (reward + value of next state * gamma) from actions in the environment. And it chooses its actions according to some ε: a fraction of the time equal to ε it spends choosing random actions, while a fraction of the time equal to 1 - ε it spends choosing the “greedy” action, i.e. the max over A of Q(S,A). To incorporate

heuristics into this structure, I added in a second ε, which I called ε_h, representing the fraction of the time spent choosing a "heuristic" action. Thus the time choosing actions is split into three segments: ε_h of the time the agent chooses a heuristic, $\varepsilon_{explore}$ of the time the agent chooses a random action, and 1 - (ε_h + $\varepsilon_{explore}$) the agent chooses the current "greedy" action according to its learned Q values.

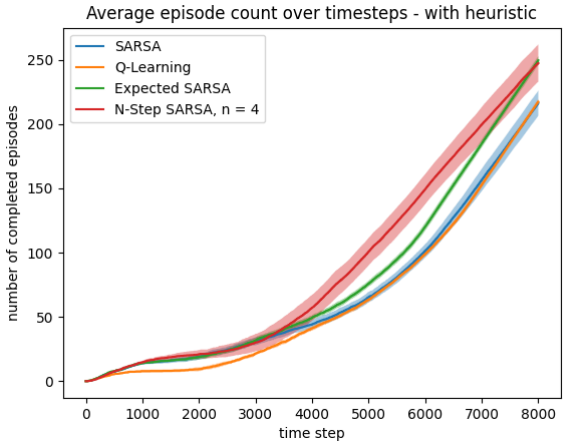
Furthermore, I wanted to ensure the agent followed mostly heuristics early on and mostly greedy actions after some initial time spent exploring, so I used an exponential decay schedule to decrement both $\varepsilon_{explore}$ and ε_h over time. This takes the form of a function where $\varepsilon = ae^{(bt)}$, where *a* is the initial value that ε takes and b is a parameter that causes ε to decay to the target final value by the target final timestep, and then the function hard-locks ε to the final value after that time step. To ensure ε_h and $\varepsilon_{explore}$ both never added up to more than 1 and decayed as a pair, I actually returned ε_h with a normal exponential decay schedule, but created a second function I called a "difference schedule" where it took in one exponential decay as an input, a second set of exponential decay parameters (that would create a strictly larger value), and returned the difference between the first and the second. I.e. you would feed in the function for ε_h and the desired function for ε_h + $\varepsilon_{epsilon}$, and the function would return $\varepsilon_{epsilon}$. I could have achieved something similar by simply ensuring the starting values of ε_h and $\varepsilon_{epsilon}$ didn't add to more than 1, but I preferred this because it caused $\varepsilon_{epsilon}$ to actually increase initially as ε_h sharply declined, but then still eventually decrease. This allows for a pattern of using mostly the heuristic very early, having a bit of room for exploration after learning from the heuristic, and then eventually relying on nearly all greedy actions. This can be seen in the following graph, where the X axis represents time and the Y axis represents the probabilities of following each action (blue region is Heuristic, orange is Explore, green is Greedy):

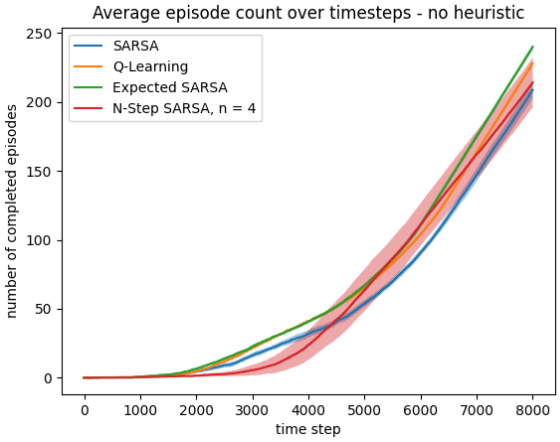


With the exploration/heuristic schedule settled (bar hyperparameter tuning), I then needed to determine what "following a heuristic" actually meant. This was fairly simple

in the general sense- I would define a heuristic policy with the state as an input and actions as an output (for example, in four rooms, a 50/50 chance to go up or right), and feed that as an input to the RL agent, who would then call it when taking an action and landing on a random number less than ε_h. The trickier part was actually designing heuristic policies for each environment. I decided to design one Good, one Mediocre, and one Adversarial heuristic for each environment. The idea was the Good heuristic would be something one could follow pretty closely and either reach the goal or get reasonably close, while the Mediocre heuristic is just generally ok advice that one could reason within a few seconds of looking at the environment, and the Adversarial heuristic is designed to stymie the agent's progress, to see if a negative effect can be achieved as well.

As for the specific RL agents used, for the tabular space I did some preliminary testing on Windy Gridworld (explained more in-depth later on) using SARSA, N-Step SARSA, Q-learning, and Expected SARSA. The results did not vary that much, so I simply picked N-Step SARSA somewhat arbitrarily for future testing (it did have slightly better results than the others for Heuristic learning, but not by much).

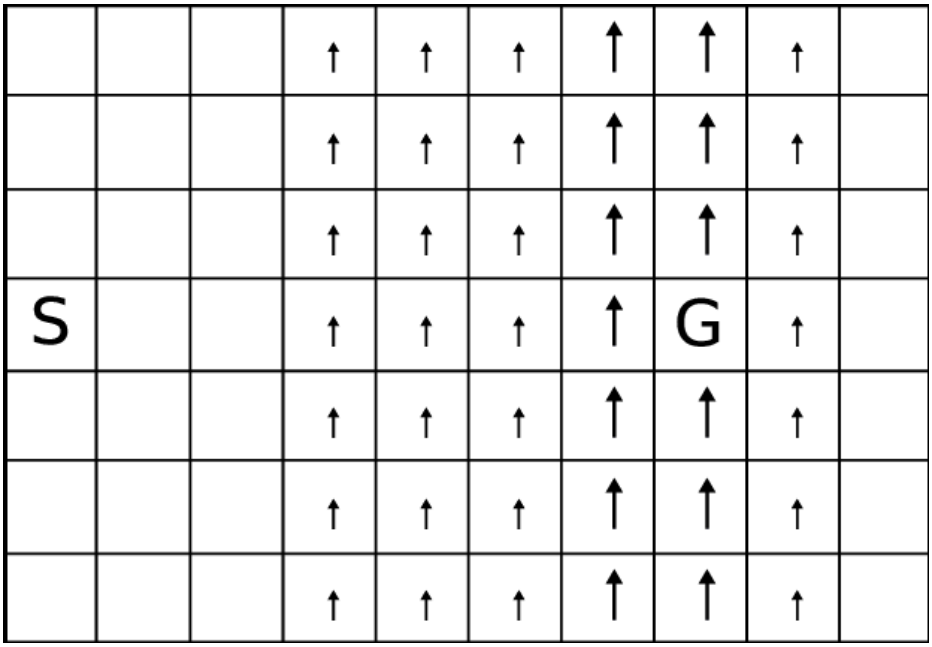




Implementation

Windy Gridworld

The first domain I tested on was Windy Gridworld. It consists of a simple 10x7 grid, with the goal at (7,3) and starting at (0,3), with wind in the middle. Strong wind pushes the agent up two squares when moving from a strongly windy square, or up one square when moving from a lightly windy square (represented by the large and small arrows in the picture). The agent receives a reward of -1 for every transition besides ending up in the goal, which returns 0 and ends the episode, and has movement options of Up, Left, Right, and Down. I chose this domain because it is a non-trivial domain that's still small enough for relatively fast early testing.



The environment itself was implemented in an earlier homework assignment, and nothing about the environment needed to change since then so I simply copied my work from that assignment.

As for the heuristics, I developed them in a one-shot intuition-based approach. I simply reasoned about the environment and developed general policies I thought would be decent, good, and adversarial. While I could have actually written out a truly optimal policy as a heuristic, that would go against the spirit of the project, which was seeing if advice gleaned on a very human level (basic reasoning about the environment) can help the agents learn faster.

The Mediocre heuristic is the easiest one to explain here. Simply choose an action in the direction of the goal relative to the agent's current position. Implementation is in the included code, but the key logic is here:

```
def wg_heuristic_3(state):
    x = state[0]
    y = state[1]

    #this is our mid-ground policy, i.e. mediocre but attempting-to-be-good advice.  Doesn't really account for wind

    if x <= 7: #left of the goal, go right
        action = Action.RIGHT
    elif x > 7: #right of the goal, go left
        action = Action.LEFT
    elif y > 3: #if we're above the goal, go down
        action = Action.DOWN
    else: #if we're above the goal, go up
        action = Action.UP

    return action
```

This heuristic does not account for the wind, which means it would not be good enough to rely on by itself, but could possibly assist early learning.

The second heuristic is the “good” heuristic- attempting to use some reasoning that the wind is a detriment to the agent’s progress, it advises moving downwards if above or equal to the goal’s height as the second priority (first is moving right if left of the goal). The full policy is:

```
def wg_heuristic_1(state): #this is our "good" heuristic
    x = state[0]
    y = state[1]

    #we know the goal is 7,3
    #since the wind always pushes up, we should have some weight on going "Down" if we're above 3 in y
    if x < 7: #if we're left of the goal, move right
        action = Action.RIGHT
    elif y >= 3: #if we're above the goal, move down to adjust
        action = Action.DOWN
    elif x > 7: #if we're right of the goal, move left
        action = Action.LEFT
    elif y < 3: #and if all else fails, move up
        action = Action.UP
    return action
```

It still may struggle to sufficiently solve the problem on its own, but should be better than the previous mediocre policy which prioritizes horizontal movement and doesn’t account for wind.

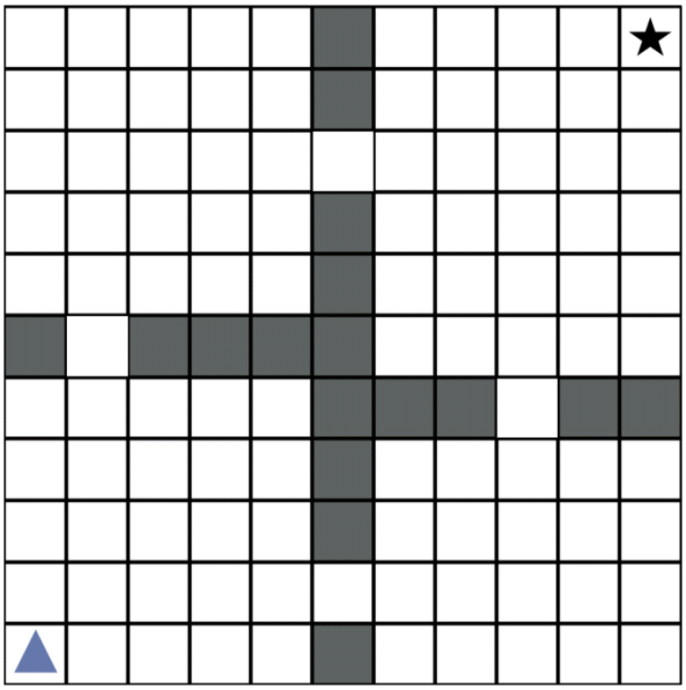
The final heuristic I used was an adversarial one – simply travel away from the goal. It can be seen below:

```
def wg_heuristic_2(state):
    x = state[0]
    y = state[1]

    #this is our adversarial policy, i.e. *bad* advice

    if x < 7: #left of the goal, go left
        action = Action.LEFT
    elif x > 7: #right of the goal, go right
        action = Action.RIGHT
    elif y >= 3: #if we can skip the goal by going up, do so
        action = Action.UP
    else: #flee to the right to make our lives harder
        action = Action.RIGHT

    return action
```

Four Rooms

The Four Rooms environment is the second environment I tested on, and the last of the ones I used tabular methods on (4-step SARSA). It is a 11x11 grid, with a number of walls that have very small 1-tile passageways through them, essentially creating the four rooms the environment is named after. The agent can only move Up, Left, Down, or Right, with a 20% chance to move adjacent to the chosen direction, adding in some

randomness to the environment. There is no reward signal for most transitions, simply a +1 for reaching the goal. This reward-sparse environment with walls can make RL agents learn slowly, as there's many obstacles in the way of randomly reaching the goal, and even then on future episodes the values learned are mostly in the top-right corner which still takes a while to return to. For this reason, I think using a good heuristic should help a lot in this environment. The environment can be seen below:



As for the heuristics, I developed a helper function for generating simple movement from point A to point B, ignoring any walls in the way. It chooses randomly between moving in X and moving in Y, unless the two points are already the same in either X or Y, as seen below:

```python
def blind_moveto(x,y,targx,targy): #helper for 4rooms - this moves WITHOUT accounting for walls
    r = np.random.random()

    act = None

    if x == targx: #if x is on target, follow by Y
        if y < targy:
            act = Action.UP
        else:
            act = Action.DOWN
    elif y == targy: #if y is on target, follow by X
        if x < targx:
            act = Action.RIGHT
        else:
            act = Action.LEFT

    elif r < .5: #if neither is on target, pick one at random
        if y < targy:
            act = Action.UP
        else:
            act = Action.DOWN

    else:
        if x < targx:
            act = Action.RIGHT
        else:
            act = Action.LEFT

    return act
```

This helper function is called by my mediocre heuristic, which just ignores walls and tries to move towards the goal. This is very simple but still provides guidance the agent otherwise does not have during early exploration:

```python
def fr_heuristic_3(state): #mediocre heuristic- doesn't account for walls but tries to go towards the goal

    x = state[0]
    y = state[1]

    act = blind_moveto(x,y,10,10)

    return act
```

The good heuristic is more complicated, essentially moving towards the nearest hallway on the route to the goal, and is thus nearly an optimal policy. Having access to such a high-quality heuristic should drastically speed up early learning, but it may cause under-exploration of other states which could have interesting effects:

```
def fr_heuristic_1(state): #this is our good heuristic
    x = state[0]
    y = state[1]

    act = None

    #cover hallways first
    if (x == 1) and (y in [4,5]): #upper hallway of r1
        act = Action.UP
    elif (y == 1) and (x in[4,5]): #right hallway of r1
        act = Action.RIGHT
    elif (y == 8) and (x in [4,5]): #right hallway of top-left room
        act = Action.RIGHT
    elif (x == 8) and (y in [3,4]): #right hallway of bottom-right room
        act = Action.UP

    #now for the rooms themselves - move to the next exit hallway
    elif (x < 5) and (y < 5): #bottom-left room
        if x > y: #bottom-right half of this room
            act = blind_moveto(x,y,4,1)
        else: #top-left half of this room
            act = blind_moveto(x,y,1,4)
    elif (x < 5) and (y > 5): #top-left room
        act = blind_moveto(x,y,4,8)
    elif (x > 5) and (y < 4): #bottom-right room
        act = blind_moveto(x,y,8,3)
    else: #in goal room
        act = blind_moveto(x,y,10,10)

    return act
```

And finally, the adversarial heuristic quite simply just moves down and left (i.e. away from the goal). This should waste a lot of time in early learning:

```
def fr_heuristic_2(state): #adversarial heuristic, always moves away from goal
    r = np.random.random()
    act = None

    if r < .5: #left and down is always away from the goal
        act = Action.LEFT
    else:
        act = Action.DOWN

    return act
```

Deep RL

In homework 8, we used ε-greedy with deep Q networks on some classic control environments, so simply modifying the policy from ε-greedy to ε-heuristic in the same way I did for the tabular space was the main change needed to test heuristics with deep RL on these environments.

For Deep RL, I extended the lengths of my exploration and heuristic schedules to fit the longer runtime in these environments. I used the following parameters for each experiment:

- 1.5 million total steps in each environment

- Replay memory with a size of 200,000 and 50,000 random prepopulate steps
- ε_h starting at 0.8 and decaying to .001 by the 1 millionth time step (2/3 of the way through training)
- $\varepsilon_{explore}$ covering the difference between ε_h and 1.0 at the start, decaying to .05 by the 1 millionth time step

Cart Pole

Cart Pole is a classic control environment, available from the Gymnasium package, and was the first of the three deep RL environments I tested heuristics on. It is the simplest of the three- there is a pole in a cart that obeys basic physics, and must be balanced by moving the cart back and forth. Moving the cart too far causes the episode to terminate early by leaving the allowed area. The reward granted is simply +1 per time step before termination, so the goal is to balance the pole for as long as possible to get long episodes and thus high reward. The agent can either push the cart to the left or right with fixed force, with no other actions available.

The heuristics here are fairly easy to understand as well. For the mediocre heuristic, I simply made the cart move in the same direction as the current angle, in order to "catch" the falling pole and thus balance it. This heuristic isn't super sophisticated, and doesn't use any parameter besides the angle, so it clearly is far from optimal but may still be helpful. See below:

```
def cp_h_2(state): #this is the "mediocre" heuristic
    x = state[0] #terminates if it leaves the (-2.4, 2.4) range
    vel = state[1]
    angle = state[2] #terminates if it leaves the (-.2095, .2095) range
    angleVel = state[3]

    if angle < 0: #if the pole is leaning left
        action = 0 #push cart left
    elif angle >= 0: #if the pole is leaning right
        action = 1 #push cart right

    return action
```

For the good heuristic, I acted somewhat similarly but also included some edge case handling, to avoid the cart moving off the edge of the allowed area. Additionally, when relatively balanced, it actually uses angular velocity instead of just angle in order to focus on keeping the pole stable in the middle when possible:

```
def cp_h_1(state): #this is the "good" heuristic
    x = state[0] #terminates if it leaves the (-2.4, 2.4) range
    vel = state[1]
    angle = state[2] #terminates if it leaves the (-.2095, .2095) range
    angleVel = state[3]

    if angle < -.05 and x > -2.2: #if the pole is leaning left and we're not about to jump off the edge
        action = 0 #push cart left
    elif angle > .05 and x < 2.2: #if the pole is leaning right and we're not about to jump off the rig
        action = 1 #push cart right
    elif x >= 2.2: #we're in danger of leaving the map to the right
        action = 0 #push cart left
    elif x <= -2.2: #we're in danger of leaving the map to the left
        action = 1 #push cart right
    else: #pole is relatively balanced and we're not close to the edge of the map, just try and offset
        if angleVel < 0: #angle is moving left
            action = 0 #move left to "catch" it
        else: #angle is moving right
            action = 1 #move right to "catch" it
    return action
```

The adversarial policy is the opposite of the mediocre one, simply trying to push the pole over as much as possible. It is worth noting how easy it is to design bad policies for so many of these environments, while good policies are far trickier:

```
def cp_h_3(state): #adversarial heuristic
    x = state[0] #terminates if it leaves the (-2.4, 2.4) range
    vel = state[1]
    angle = state[2] #terminates if it leaves the (-.2095, .2095) range
    angleVel = state[3]

    if angle < 0: #if the pole is leaning left
        action = 1 #push cart right (makes it fall faster)
    elif angle >= 0: #if the pole is leaning right
        action = 0 #push cart left (makes it fall faster)

    return action
```

Mountain Car

This is a relatively tricky environment for RL agents. There is a large valley, and a car that is attempting to reach a point atop the hill to the right of the valley. The car can only apply leftward or rightward fixed force (or no force), but does not have enough power to climb the rightward hill by itself. It can climb the leftward hill and use momentum to scale the rightward hill, but that requires moving farther from the goal and then keeping a very constant rightward force, something the agent will not usually do when exploring. Moreover, every step receives a -1 reward besides termination, thus even if the agent explores up the hill it likely will treat that as just as bad as any other space since it doesn't push all the way through towards the goal.

The mediocre heuristic I used was simply to accelerate left unless we're either moving fast or are very far to the left of the area already. This should get us up the leftward hill and hopefully gather enough momentum to finish.

```
def mc_h_2(state): #mediocre heuristic
    x = state[0]
    vel = state[1]
    if x > -.8 and vel < .03: #unless we're moving fast or very far to the left
        action = 0 #accelerate left most of the time
    else:
        action = 2 #in those specific conditions, accelerate right
    return action
```

The good heuristic uses similar principles but is hopefully a bit more precise. It has a range (x > .3) where it accelerates right even if momentum is low, because it will lose momentum as it climbs the hill and we don't want to give up so close to the end.

```
def mc_h_1(state): #good heuristic
    x = state[0]
    vel = state[1]
    if x < -.6: #decently far up the hill to the left to build momentum
        action = 2 #accelerate right
    elif x < .3 and vel < .03: #not far enough up the hill/not enough speed
        action = 0 #accelerate left to gain some height
    else:
        action = 2 #accelerate right
    return action
```

The adversarial heuristic is again trivially simple. By simply not accelerating the car will never move and just waste time.

```
def mc_h_3(state): #bad heuristic
    action = 1 #don't ever accelerate

    return action
```

Acrobot

Acrobot is an easier environment for RL agents to learn as compared to mountain car, but is far harder to determine heuristics for. It consists of two limbs attached with joints to a fixed point, with torque only able to be applied on the first limb. The goal is to get the second limb above a certain height, but to do so a decent amount of angular momentum has to be built up first by essentially "wiggling" the upper limb.

To design a mediocre heuristic, I just determined that pushing the angle further from the center would be somewhat helpful as it pushes the whole structure upwards. Obviously this is far from sufficient to solve the problem, but it may help guide some exploration.

```
def ab_h_2(state): #mediocre heuristic:
    ct1 = state[0] #cosine of theta 1, between -1 and 1
    st1 = state[1] #sine of theta 1, between -1 and 1
    ct2 = state[2]
    st2 = state[3]
    avt1 = state[4] #angular vel of theta 1, between -4pi to 4pi
    avt2 = state[5] #angular vel of theta 2, between -9pi to 9pi

    #a very simple policy could just be to keep increasing the magnitude of theta1.
    #So if it's pointing right we want to push more to the right, and vice versa for the left

    #angle to the right -> positive sine, angle to left -> negative sine
    if st1 > 0:
        action = 2
    if st1 < 0:
        action = 0

    return action
```

For the better heuristic, I went more complex, reasoning that if there's sufficient momentum we should carry through the movement, but if there's not much momentum we want to wiggle the arm to generate momentum, and if there's a medium amount of momentum we just want to carry in the same direction as the lower arm's momentum to build on it. That resulted in the following policy:

```python
def ab_h_1(state): #good heuristic:
    ct1 = state[0] #cosine of theta 1, between -1 and 1
    st1 = state[1] #sine of theta 1, between -1 and 1
    ct2 = state[2]
    st2 = state[3]
    avt1 = state[4] #angular vel of theta 1, between -4pi to 4pi
    avt2 = state[5] #angular vel of theta 2, between -9pi to 9pi

    #policy isn't as obvious here, but generally we want highly neg or highly positive velocities,
    #and want to "wiggle" until we get those

    #it's very possible this "good" hpol will be useless in the end since this environment's dynamics
    #are less obvious

    #actions can be:
    #0: apply -1 torque
    #1: apply no torque
    #2: apply +1 torque

    if avt1 < -6 and avt2 < -10: #sufficient negative momentum
        action = 0 #keep going in that direction
    elif avt1 > 6 and avt2 > 10: #sufficient positive momentum
        action = 2 #keep going in that direction
    #otherwise we will encourage wiggling behavior- if avt2 is low in magnitude then move in the opposi
    #but if it's high in magnitude then follow it through
    elif -4 < avt2 < 4: #not much momentum so we want to "wiggle" it
        if avt2 > 0:
            action = 0 #go in opposite direction
        elif avt2 < 0:
            action = 2 #go in opposite direction
    else: #larger momentum so we want to build it further by following through
        if avt2 > 0:
            action = 2 #follow through
        elif avt2 < 0:
            action = 0 #follow through

    return action
```

And finally, again, the adversarial heuristic is very simple. By never applying any torque the agent will be stuck forever:

```python
def ab_h_3(state): #bad heuristic, stay still and don't do anything

    action = 1
    return action
```

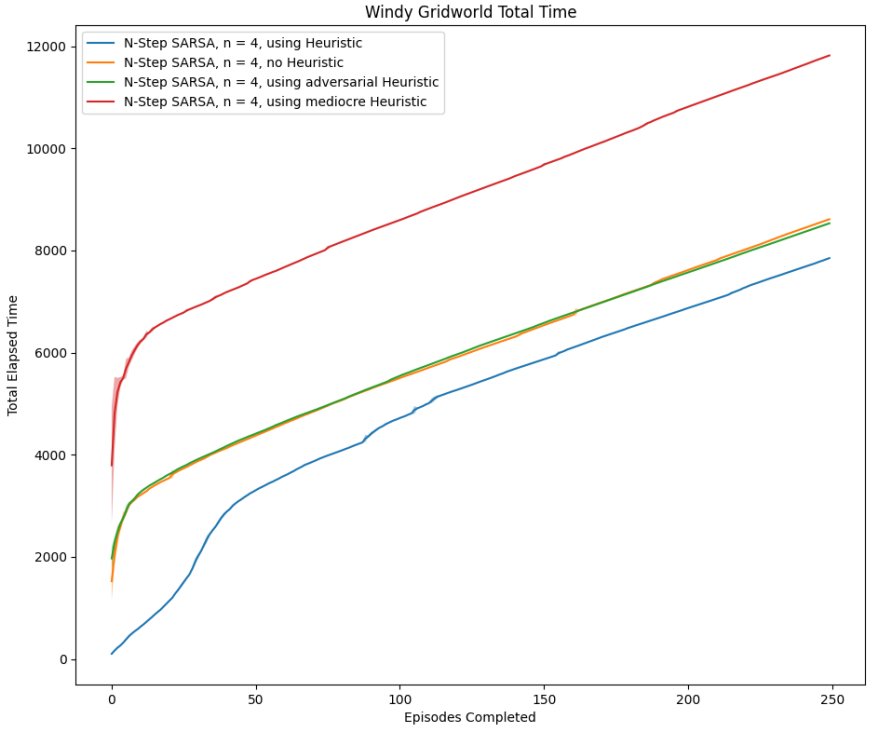
Results/Discussion

Tabular Agent Results

The Windy Gridworld and Four Rooms environments were run with tabular methods, and thus ran much faster and produced clear, crisp results. This is where I did most of the testing and tweaking of heuristic implementation. As for actual performance, I ran each heuristic-led RL agent for 20 trials and 250 episodes each trial. I compared how long each agent took to finish 250 episodes, and thus the agents that finished faster

would be considered better (as both of these environments have the goal of finishing episodes as fast as possible). The results are below:

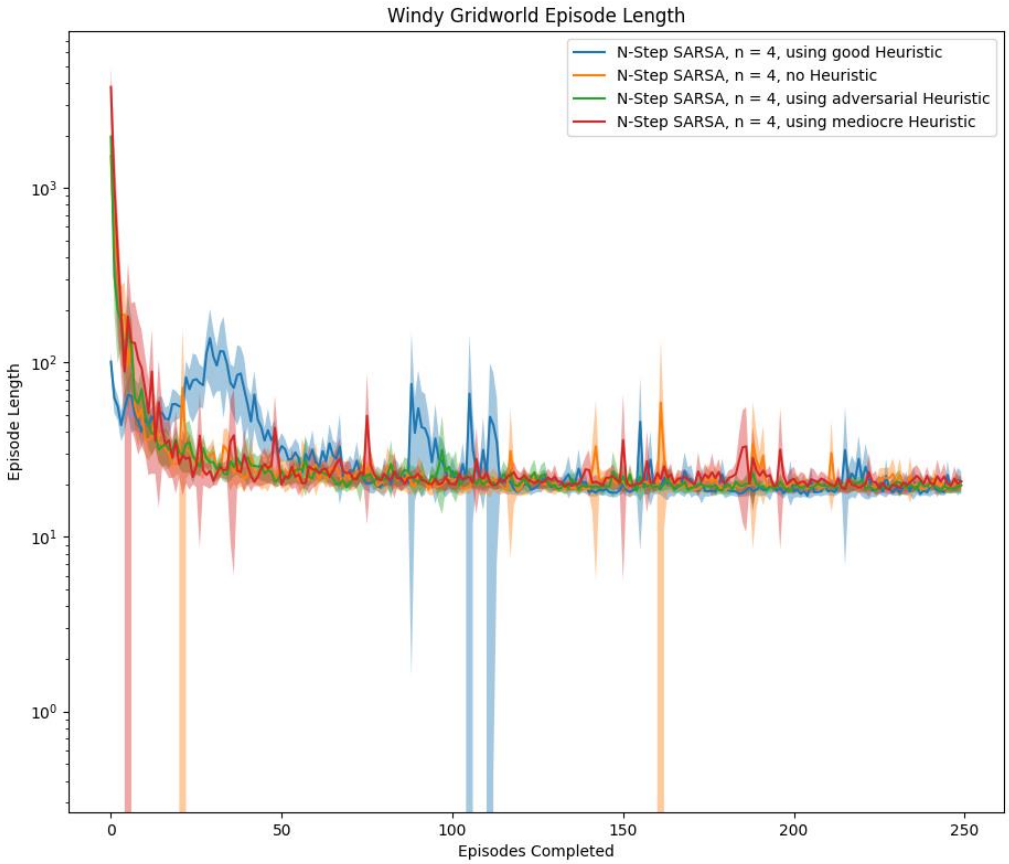
For windy gridworld:



Using a good heuristic helped a lot here. The early episodes were completed significantly faster, and while the agent did have a bit of a slow-down period after it stopped relying on the heuristic, it did recover quickly and its final performance (i.e. the slope at the end) matched the others. And its total time for 250 episodes was still notably shorter than any other method.

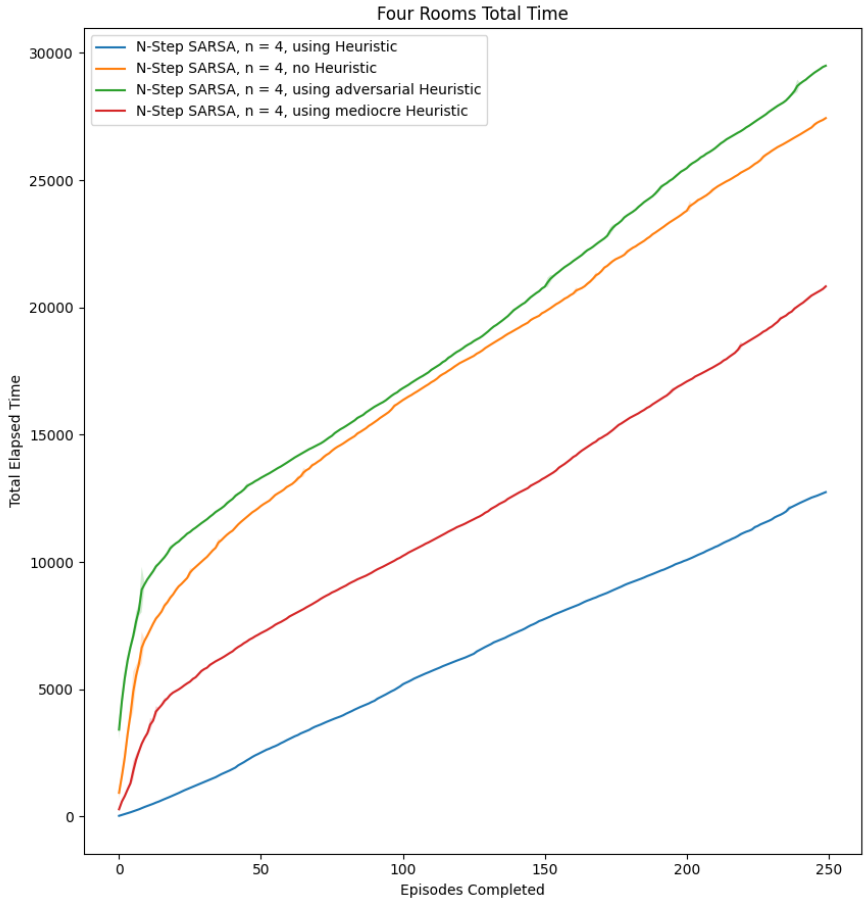
Strangely, the mediocre heuristic actually performed far worse than no heuristic, and the adversarial heuristic performed about as well as no heuristic. I believe that because windy gridworld is filled with negative rewards, the agent has to spend some time exploring every possible state to learn the negative values there before deciding to continue taking the path to the goal. Since the mediocre heuristic doesn't account for wind, it gets the agent close to the goal but not quite, getting stuck above the goal, and thus learns that the path to the goal is far more negatively valued than it actually is. If I had more time, I would try re-running this environment with a 0 reward for all transitions except +1 for the goal to see if that changes performance. As for the adversarial and no heuristic methods being nearly identical, I believe the negative rewards everywhere encourages nearly complete exploration of the state space no matter what, so a heuristic that stays away from the goal during early exploration does the same thing in practice.

Looking at episode length over time tells a similar story:



All of the methods converge to a similar speed, but while the good heuristic is *significantly* faster in early episodes (remember the y-axis is a log scale here), it does need to spend some of its later episodes chasing unexplored paths since the main one takes on negative values. Still, even this drawback is not nearly enough wasted time to let the other methods catch up, and using a good heuristic performs better in terms of overall time spent.

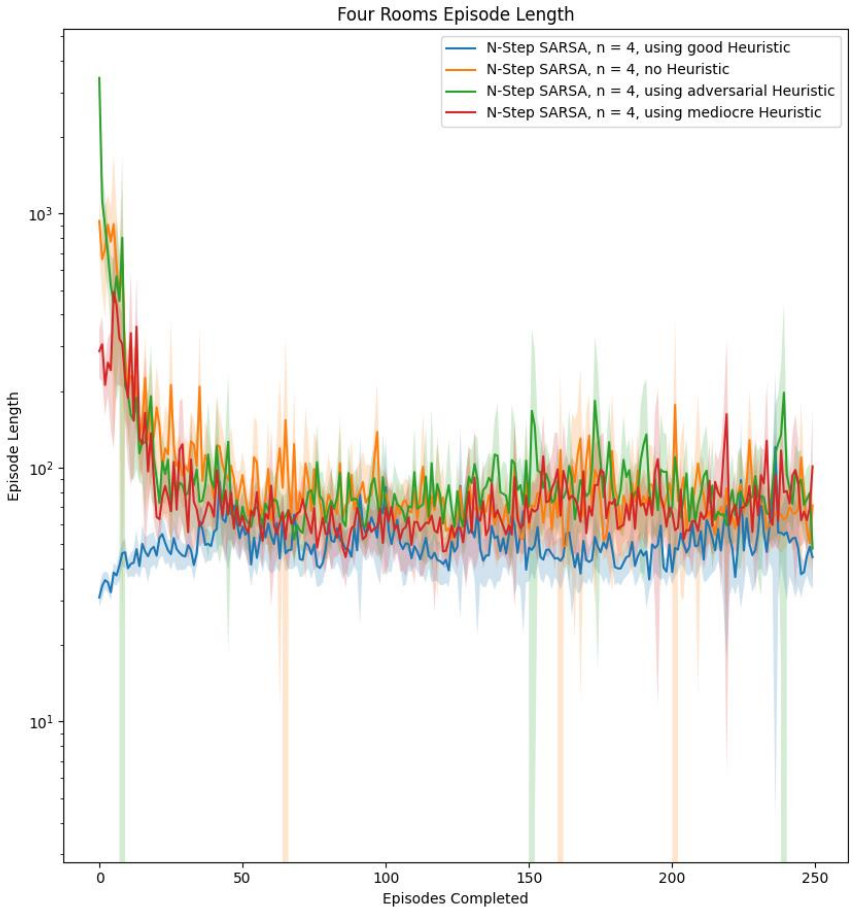
The results for Four Rooms are more in line with my initial hypothesis for how each policy would perform.



The good heuristic far, far outperformed every other method. It started strong with its near-optimal heuristic policy, but even well after ε_h decayed to near-0 it still had very fast episodes. Since four rooms has no negative rewards for normal timesteps, there's no strong exploration incentive baked into the reward function and thus once a good policy is learned the agent can keep exploiting it.

The mediocre heuristic still provided a marked improvement over no heuristic, cutting down the time wasted on early exploration by vaguely guiding the agent in the direction of the goal, even though it didn't account for walls at all.

The adversarial heuristic caused the agent to perform worse than no heuristic, but not in the long run. It just wasted extra time on early episodes, taking more than twice as long to complete the first episode. Once ε_h decayed to near-0, it essentially followed the same trajectory as no heuristic. This can be seen in the episode lengths plot below:

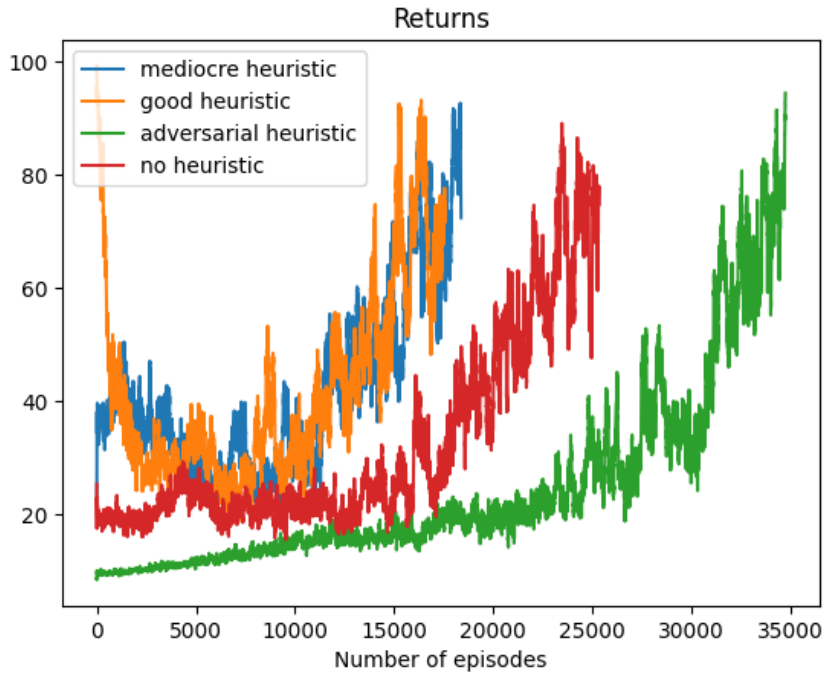


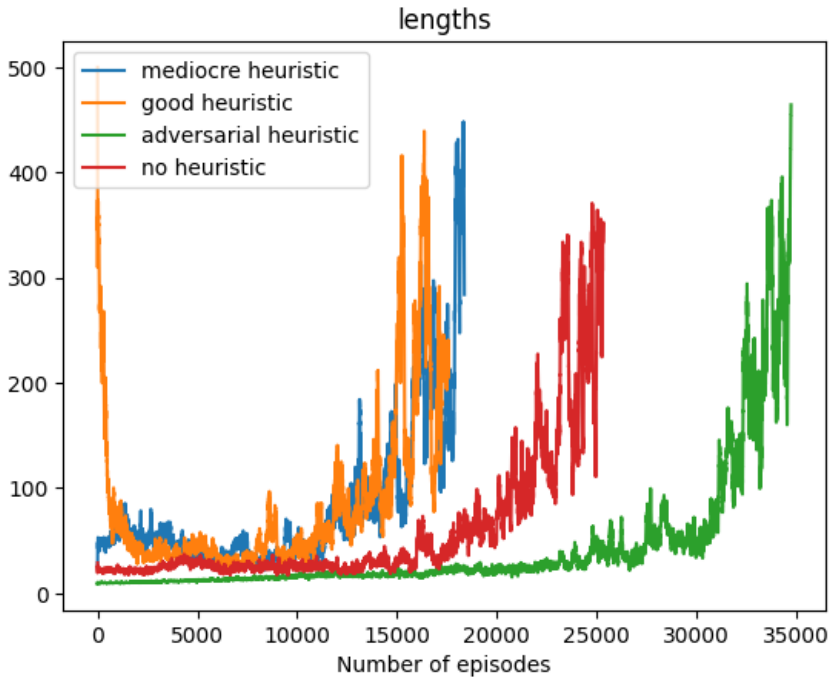
Thus, from experiments in the tabular space, it seems that using heuristics can be very helpful in RL, guiding learning early and having a lasting effect even when the heuristic policies are no longer being directly called. However, it can vary a lot based on environment, reward function, and the quality of the heuristics themselves.

Deep RL Results

For the deep RL experiments, I ran each method for 1.5 million steps, and thus the number of episodes determines how effective each method was.

In Cart Pole, fewer total episodes is better since you want each episode to last as long as possible. Thus, the graphs that are shorter in the X-Axis here are better, with the Y axis representing either the returns per episode or episode length:



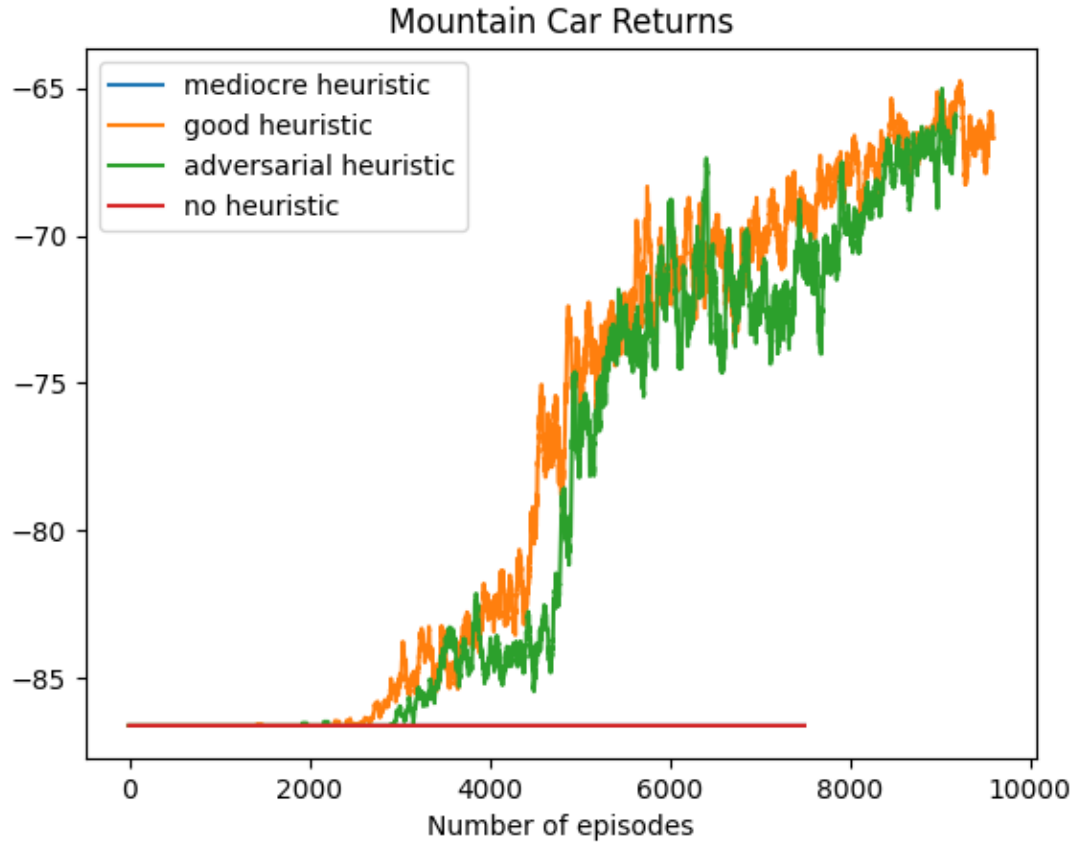


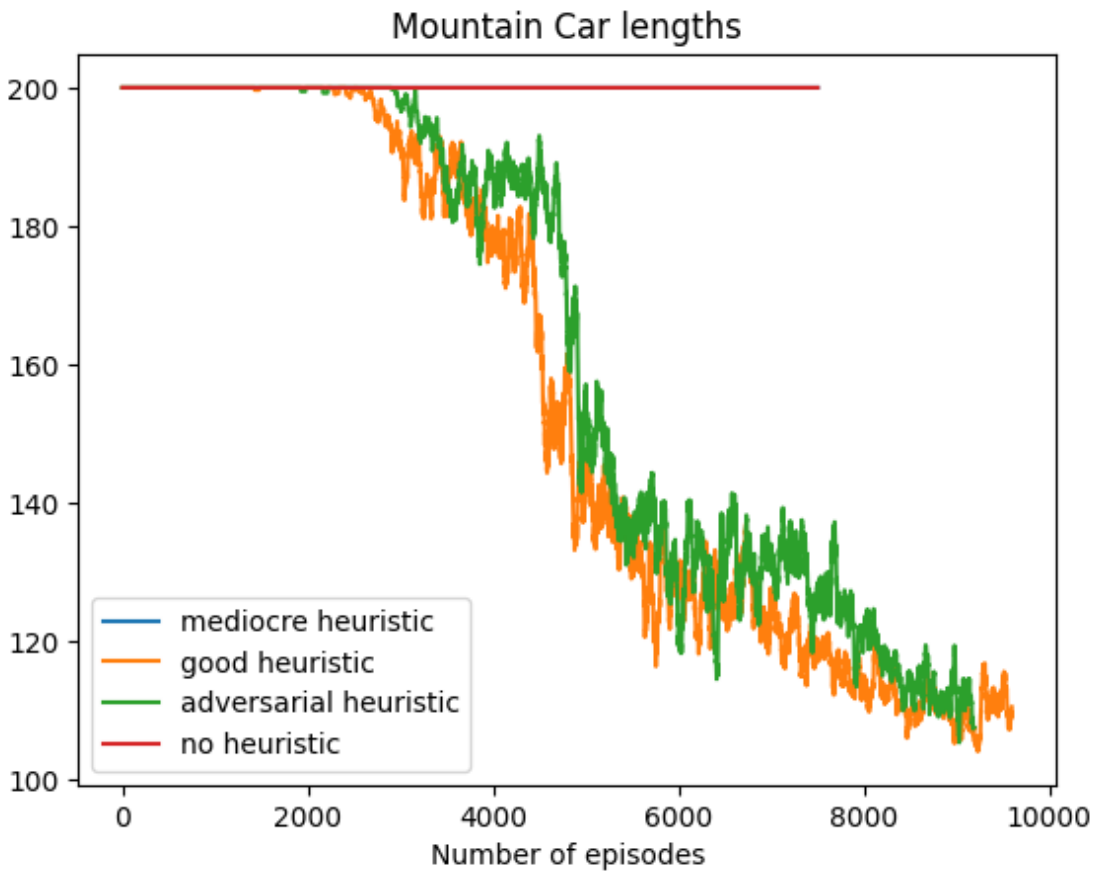
The results here do bear out the hypothesis that using a heuristic seems to drastically speed up learning. There wasn't significant difference between the "good" and "mediocre" heuristics in the long run despite the "good" heuristic doing far better at the very beginning, but both did better than no heuristic, by speeding up early learning and even achieving better performance in the long run. The adversarial heuristic performed as expected- drastically holding back learning compared to no heuristic or either of the

other methods, but its final performance still seemed on-par, it just wasted a lot of time getting there.

One interesting thing to note is the U-shaped rewards of the good heuristic. Nearly exclusively following the heuristic led to very strong early episodes, but as the rate of following the heuristic dropped even a little bit, the performance tanked and it took a while to recover that level of performance. So, the knowledge doesn't seem as "sticky" as it was in the tabular case, but even with that drop-off it still performed notably better in the long run than without a heuristic.

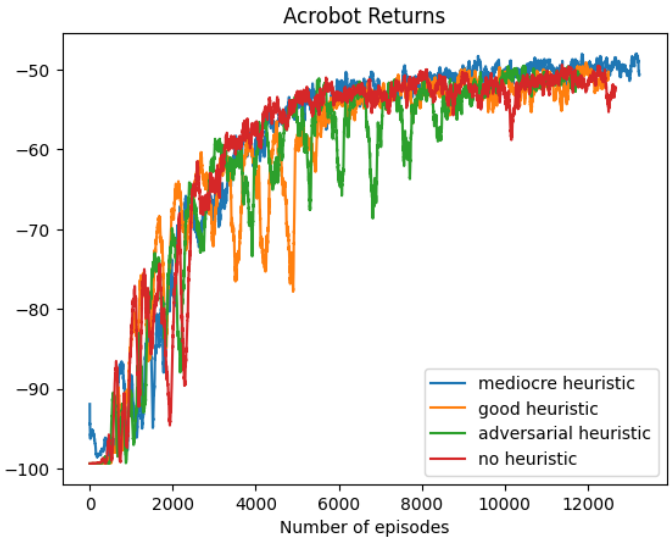
Mountain Car is perhaps the most difficult of the three environments presented. Since the goal requires very specific pathing through the state space, with a lot of possible pitfalls in the way, it's easy for an agent to get lost and never make that jump. Results can be seen below:

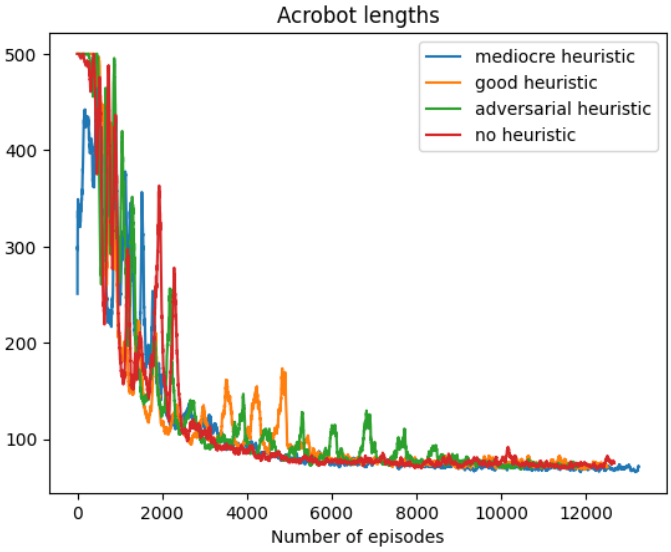




From my previous exploration with deep RL in homework 8, the agent wouldn't consistently solve this environment, sometimes completing a whole run without ever actually reaching the goal. This occurred in my experiment here as well- without a heuristic, the agent never even completed an episode in any way besides timing out. However, with heuristics, I was able to avoid this issue. While it is true that in previous trials the deep RL no-heuristic agent sometimes achieved a workable policy by the end, the inconsistency there compared to the consistency with the "good" heuristic (across 3 separate times running it) speaks to the value of a good heuristic in such a difficult environment. What is odd is that the adversarial heuristic did reach the goal while the "mediocre" heuristic did not. I think this is similar to the Windy Gridworld case, where a "close but not good enough" policy actually hurt learning more than learning nothing- by learning a lot of negative values on the route to somewhere near the goal, the agent was nudged away from the goal instead of towards it once the heuristic decayed.

For acrobot, similarly to Mountain Car, fewer episodes is more desirable since ending an episode early is the goal. Results are as follows:





It appears that the heuristics had very little impact on performance here, with every method achieving pretty similar long-term and even short-term performance. The "good" and adversarial heuristics both performed slightly worse than the mediocre/no heuristic cases, which either could imply that the "good" heuristic I tried was actually a bad one for the environment, or that similarly to cart-pole it just skewed early learning in a direction that's close-but-not-quite when compared to the optimal policy. Weirdly the "mediocre" heuristic did best here, which either implies it's a much better heuristic than I initially thought, or some quirk of stochasticity or early weights for the deep Q network rolled out in its favor. These results aren't too surprising, though, as the relative complexity of the different state variables made it far more difficult to design policies, and I wasn't as confident in the relative strength of each of my policies as I was in previous environments.

For Deep RL in general, even the best heuristics can create U-shaped performance, where a good heuristic can improve the speed of early episodes but the network has trouble carrying the experience from those early episodes forward. It is possible this issue could be solved with hyperparameter tuning, or it might be more innate to the way that Deep RL functions- its learned value functions might simply be too intricate to easily carry forward learned policies from guided exploration. Even with that said, the presence of heuristics did provide significant performance improvements in at least the Cart Pole and Mountain Car environments. More generally, Deep RL is still a pretty new field and needs a lot more exploration before its methods are more locked-in and reliable (Irpan, 2018), and thus it's not as surprising that the results are more concrete in tabular space as opposed to Deep RL.

Conclusion and Future Work

Heuristics are an interesting space to explore in RL. Early results here are promising, showing that they can drastically speed up early learning without sacrificing long-term optimality if given good heuristics. Additionally, adversarial heuristics (or even attempted good heuristics that don't properly account for the quirks of the environment) can easily make learning worse, proving that the "advice" must be good in order to be helpful, and it can sometimes be unclear what constitutes "good" advice. However, as with many things in RL, heuristic learning can be very temperamental, affected strongly by the idiosyncrasies of a reward function or hyperparameters. And the results do seem notably more pronounced in the tabular environments as opposed to deep RL, but the deep RL results are still significant. If I were to continue this work further, I would experiment with slight alterations of the environments I tested on, such as altering Windy Gridworld to have a +1 reward in the goal and -1 everywhere else. And for the Deep RL space, I would experiment more with tuning hyperparameters, trying more heuristics, perhaps only using heuristics for the prepopulate steps for the replay memory, and possibly testing on more environments.

Citations

1. Rafael Rodriguez-Sanchez, Benjamin A. Spiegel, Jennifer Wang, Roma Patel, Stefanie Tellex, George Konidaris: "RLang: A Declarative Language for Describing Partial World Knowledge to Reinforcement Learning Agents", 2022; arXiv:2208.06448.
2. Irpan, A. (2018). *Deep Reinforcement Learning Doesn't Work Yet*. https://www.alexirpan.com/2018/02/14/rl-hard.html