Bennett Brain (no teammates)

4/12/24

Project Milestone Report

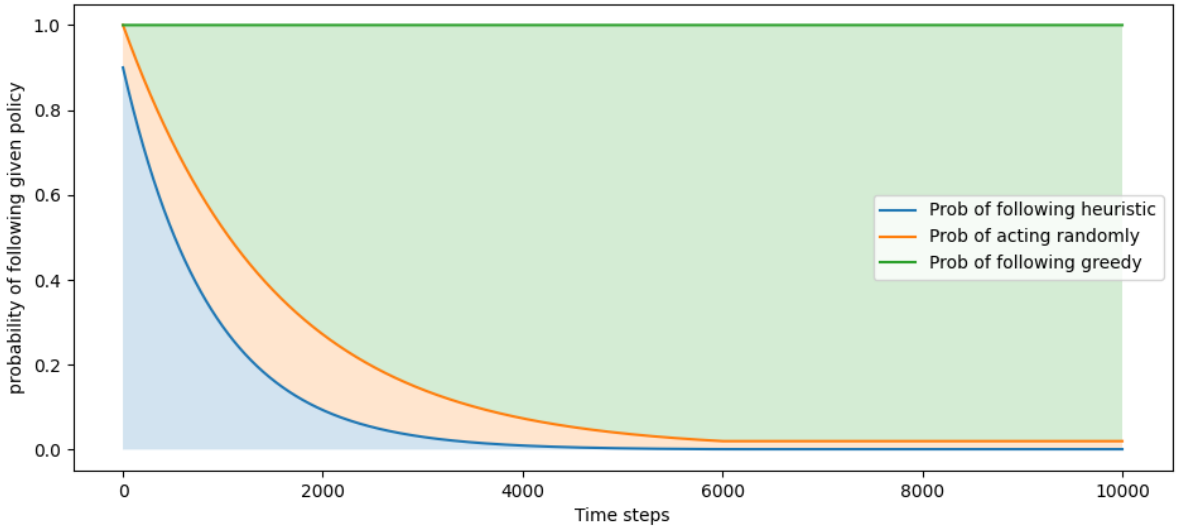
My initial project was going to be using RLang to do transfer learning- i.e. implementing heuristic policies to give RL agents some starter knowledge to see how it could improve learning speed/long-term success. The paper on RLang is here: https://arxiv.org/abs/2208.06448

However, after attempting to use RLang for a while, I learned that it wasn't well-maintained, and in order to get it working I would have to do a lot of troubleshooting and bug fixing. Doing that would probably take up most of the time I have to work on this project, and after talking with Prof. Lawson, I pivoted the project a bit. Instead of using RLang, I decided to directly implement the key ideas to test (i.e. seeding agents with heuristic strategies to see how it improves learning) myself.

My new plan was now to create "heuristic" strategies for a number of domains (windy gridworld, 4 rooms, and some of the q-learning domains we tackled in ex8), and then have agents initially follow the heuristics early on in their exploration process. I planned on testing it in tabular space (i.e. using n-step sarsa) as well as with deep RL, once the tabular domains were done.

More specifically, my plan is to use an epsilon schedule (similar to homework 8) to balance between following the heuristic, picking randomly, and following the current greedy option according to the agent's value function. I.e. there is some e1 and e2, where if a random number lands below e1 it picks randomly, between e1 and e2 it follows the heuristic, and outside of that it follows a greedy policy. And I plan on having it follow mostly the heuristic at first, then random for a bit to explore, then mostly follow the greedy policy, but leave the random exploration parameter at around .02-.05 so it's never purely greedy.

For example, the probabilities would look like this (filled-in region is the probability at a given timestep for each):



So early on, most of the probability space is taken up by heuristics, but there's always some room for random action, and as time goes on the policy acts mostly greedily.

So far, I have successfully implemented and tested heuristic-guided learning on two tabular domains- Windy Gridworld and Four Rooms.

My implementation uses the algorithms from previous homework assignments as a base. For the heuristic/random scheduling, I took the epsilon-scheduling from ex8 and then created something called a "difference schedule" which just takes a second, larger schedule as a target and fills in the difference. That allows for calculating the orange region in the previous graph.

As for heuristic policies themselves, with the above implemented, the code for them is as follows:

```python
def create_epsilon_heuristic_policy(Q: defaultdict, h_Sched: ExponentialSchedule, eps_Sched: DiffSchedule, h_pol: Callable) -> Callable:

    """Creates an epsilon soft policy from Q values and a heuristic, with exponential decay for both epsilon and a heuristic.

    A policy is represented as a function here because the policies are simple. More complex policies can be represented using classes.

    Args:
        Q (defaultdict): current Q-values
        epsilon (float): softness parameter
    Returns:
        get_action (Callable): Takes a state as input and outputs an action
    """
    # Get number of actions
    num_actions = len(Q[0])

    def get_action(state: Tuple, t_step) -> int:
        #now uses an epsilon and h-epsilon value to determine whether to pick randomly, use a heuristic, or pick argmax
        eps_Val = eps_Sched.value(t_step)
        h_val = h_Sched.value(t_step)

        r = np.random.random()
        if r < eps_Val:
            action = np.random.randint(num_actions)
        elif r < eps_Val + h_val: #in the "heuristic" range
            action = h_pol(state)
        else:
            action = argmax(Q[state]) #this should break ties randomly, reusing argmax from ex1

        return action

    return get_action
```

And the n-step SARSA algorithm calls them as such:

```
Q = defaultdict(lambda: np.zeros(env.action_space.n))
policy = create_epsilon_heuristic_policy(Q, eps_Sched=eps_sched, h_Sched=h_sched, h_pol=h_policy)
epLengths = [] #we need episode lengths for plotting
epEnds = [] #also want episode end times for plotting

timer = 0
epcount = 0
while epcount < num_episodes:

    epStartTime = timer #needed to see how long each episode is for plotting later

    S, _ = env.reset() # initialize S0
    A = policy(S, timer) # and A0
    T = np.inf
    t = 0
    tau = 0
    rewards = [0]
    states = [S] #and store them in arrays to be added to later
    actions = [A]
    done = False
```

(Highlighted lines are the ones that are different from previous implementations).

In order to test how effective using heuristic policies are, I decided to create three for each environment- one that I believed would be adversarial (i.e. cause the agent to learn *worse* by giving advice that pushes it away from the goal), one that would attempt to be helpful but would be very mediocre (i.e something as simple as walk towards the goal ignoring any walls/wind/etc), and one good one that uses more complex knowledge of the environment.

For windy gridworld, my heuristics are as follows:

```
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
```

The “good” heuristic here is far from optimal, but it does use some knowledge of the wind (i.e. moving down is helpful if we’re sorta-near the goal and at or above its altitude since the wind pushes upwards).

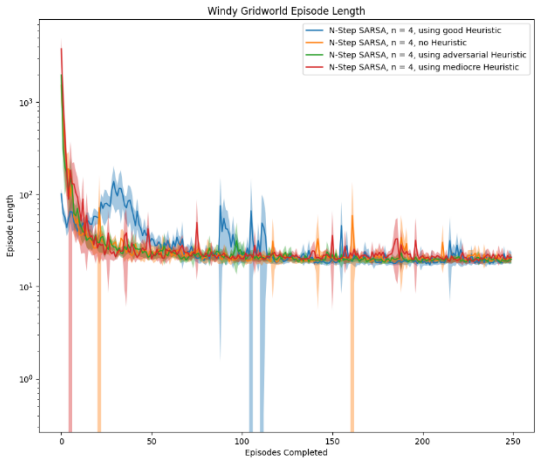
```python
def wg_heuristic_2(state):
    x = state[0]
    y = state[1]

    #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
```

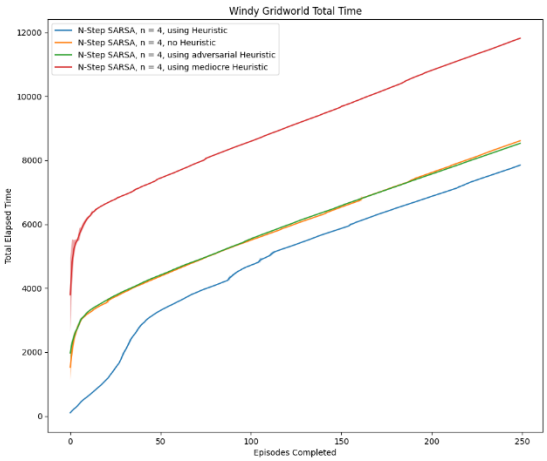
The adversarial heuristic just moves away from the goal as much as possible.

```python
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
```

The mediocre advice doesn't account for wind, just attempting to move straight towards the goal.

However, what was really interesting was how they actually performed:





(I used Log scale for episode length because some were far, far longer than others)

The “mediocre” heuristic actually performed far worse than the adversarial one- it’s possible that in a somewhat tricky environment like windy gridworld, following blind advice caused more issues. It may have circled around the goal for a while, pushed by wind, and didn’t get enough information about the rest of the environment but also didn’t learn a good path to the goal. So as the agent stopped following the heuristic, it had to learn the whole environment itself and the heuristic just wasted time essentially.

The “bad” heuristic was nearly indistinguishable from no heuristic in this case. Windy gridworld is seeded with negative rewards everywhere but the goal, so it encourages agents to explore on their own, which may be why the actual behavior of the two agents was strikingly similar. Moving away from the goal encourages exploring every nook and cranny of the grid, which is probably what using no heuristic does anyways in this environment due to the negative rewards for normal transitions.

The ”good” heuristic seemed to do its job, but did so in an interesting way. Its initial episodes were **significantly** shorter than either of the other algorithms, as it had a good guide towards the goal, but then as it followed the heuristic less and less it had some longer episodes than the others. This is because the “good” heuristic cause it to under-explore the state space, so as the agent lost its crutch it spent some time exploring. Regardless, it still spent notably less overall time to do the same number of episodes by the end. So its long term per-episode performance ended up very similar (all methods seemed to converge on a near-optimal policy by the end), but it got there faster by a good margin, wasting less time in initial episodes.

As for the four rooms environment, I made the following heuristics:

First, a necessary helper function:

```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 is just a blind move-to function where it goes from current position towards a target, not accounting for any walls or obstacles and weighting travel in x equal to travel in y.

```python
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
```

The “good” heuristic considers where the walls are by guiding the agent towards the next room exit if not in the upper-right room. If in the upper-right room, it guides the agent towards the goal. As for hallways, it just pushes the agent through them directly.

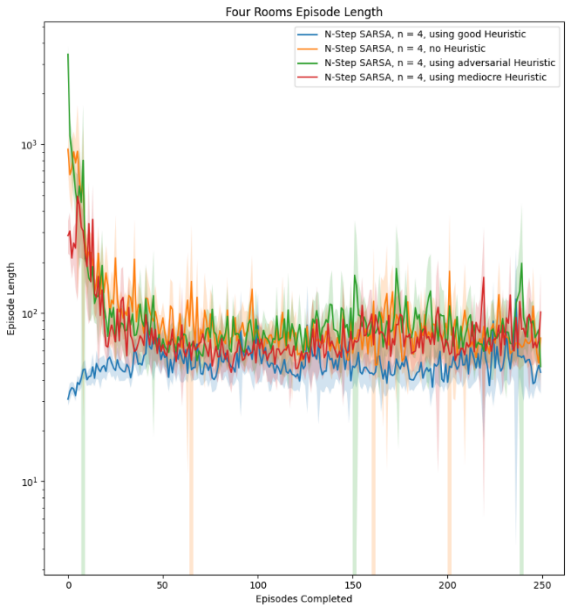
```python
def fr_heuristic_2(state): #adversarial heuristic, always moves away from goal
    r = np.random.random()
    act = None

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

    return act
```

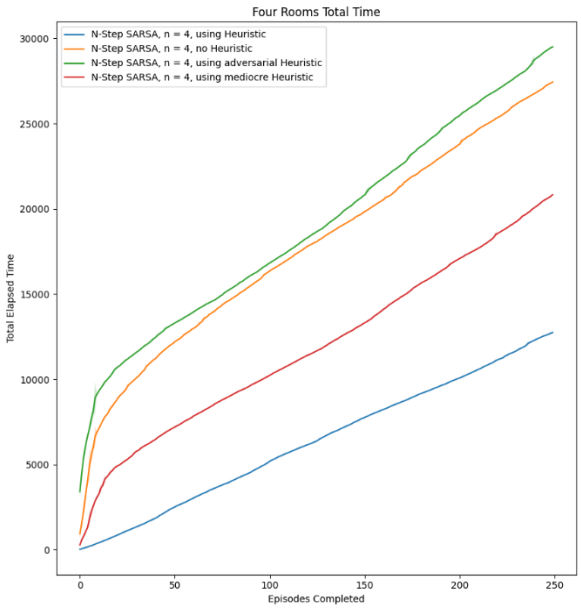
The “bad” heuristic just goes down or left, which always are away from the goal.

```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 “mediocre” heuristic uses goal knowledge but not wall knowledge. If followed by itself it would never reach the goal since it’d get caught in the corner of the first room, but since the agent will also be mixing in random and its learned greedy actions this may still be a helpful guide.

As for performance:





This is more similar to what I'd expect from using heuristics. The mediocre heuristic does better than no heuristic, but the good heuristic far outperforms any of the other methods. While the adversarial heuristic actually performs worse than using no heuristic.

As four rooms is a more difficult environment than windy gridworld, it seems that none of the methods learned a truly optimal path (interestingly the "good" heuristic still did better when relying on the heuristic than it did following greedy by the end, if only slightly), but in the long run giving a good heuristic caused the agent to learn a better path, it didn't just speed up early training.

So far, heuristics can be very environment-dependent, but can improve both early training and long-term success, with more pronounced effects as the environments are more difficult.

Additionally, some heuristics that may seem decent at a glance (such as moving towards the goal in windy gridworld) can actually slow agents down depending on certain quirks of the environment.

Going forward, I plan on implementing heuristics (with scheduling) with Deep Q-learning on a few more environments and testing the results there. I could stay in the tabular space and experiment with hyperparameters, more environments, more heuristics, etc., but in the interest of time I think applying these methods to Deep RL is the best next step.