Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2

Table of Contents

Data and Plots

To see plots of my warmup exercises, see:
https://public.tableau.com/views/PS2Warmup/WarmupAThroughputvs_Delay?:embed=y&:display_count=yes

To see plots of my contest submission, see:
https://public.tableau.com/views/PS2PartD/Scores?:embed=y&:display_count=yes

For my Warmup A data, see:
https://www.dropbox.com/s/ovd8ph9dpkhgbh4/window_sizes.csv?dl=0

For my Warmup B data, see:
https://www.dropbox.com/s/o3kwa6r1pipreoq/aimd_consts.csv?dl=0

For my Warmup C delay threshold data, see:
https://www.dropbox.com/s/nwfvjlsc3ixbutp/delay.csv?dl=0

For my Warmup C AIMD constants data, see:
https://www.dropbox.com/s/zwv3v7jdqm522x6/aimd_delay.csv?dl=0

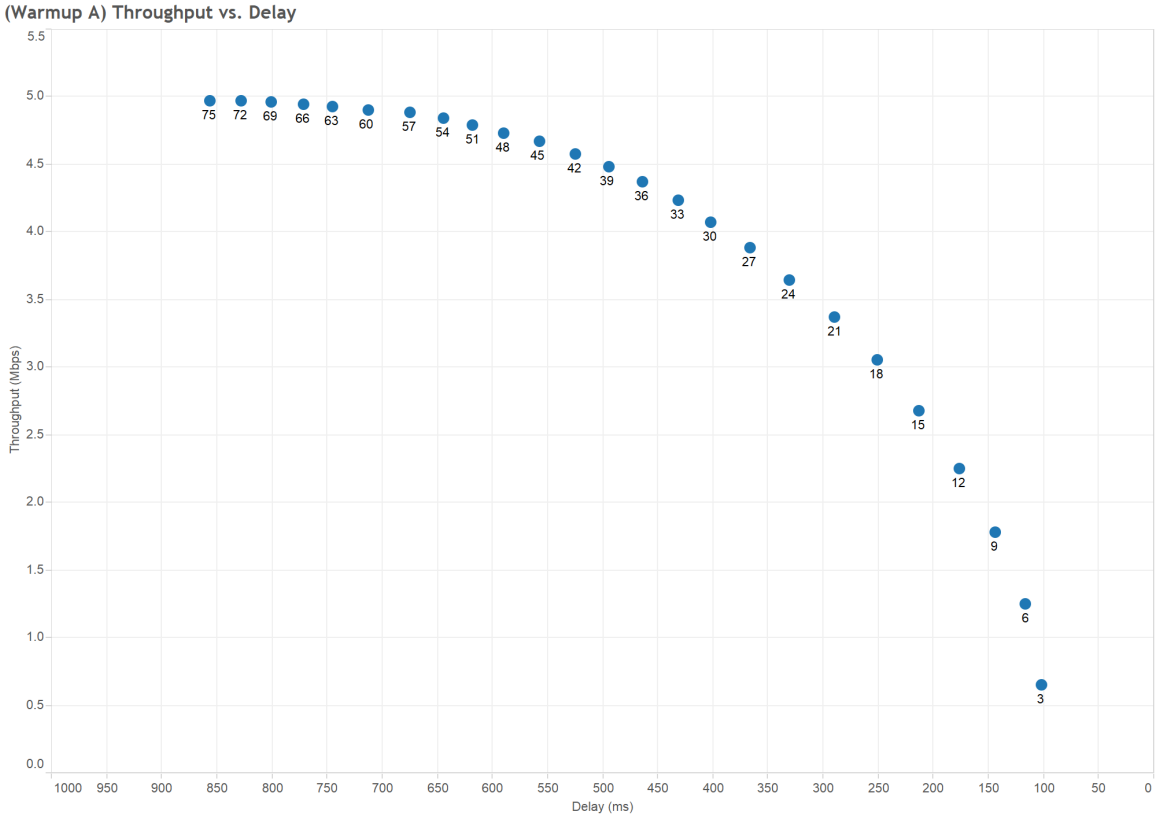
For my Exercise D data, see:
https://www.dropbox.com/s/45ac8q377t6mj5e/best_algo.csv?dl=0

Warmup Exercise A

I tried fixed window sizes ranging from 3 to 75 in increments of 3. For each window size, I ran the contest simulation three times. The results were repeatable - throughput and signal delay did not vary much between runs. Since each window size has three (throughput, 95th percentile signal delay) pairs, I took the median of the three throughputs and median of the three 95th percentile signal delays to use in my plot below.

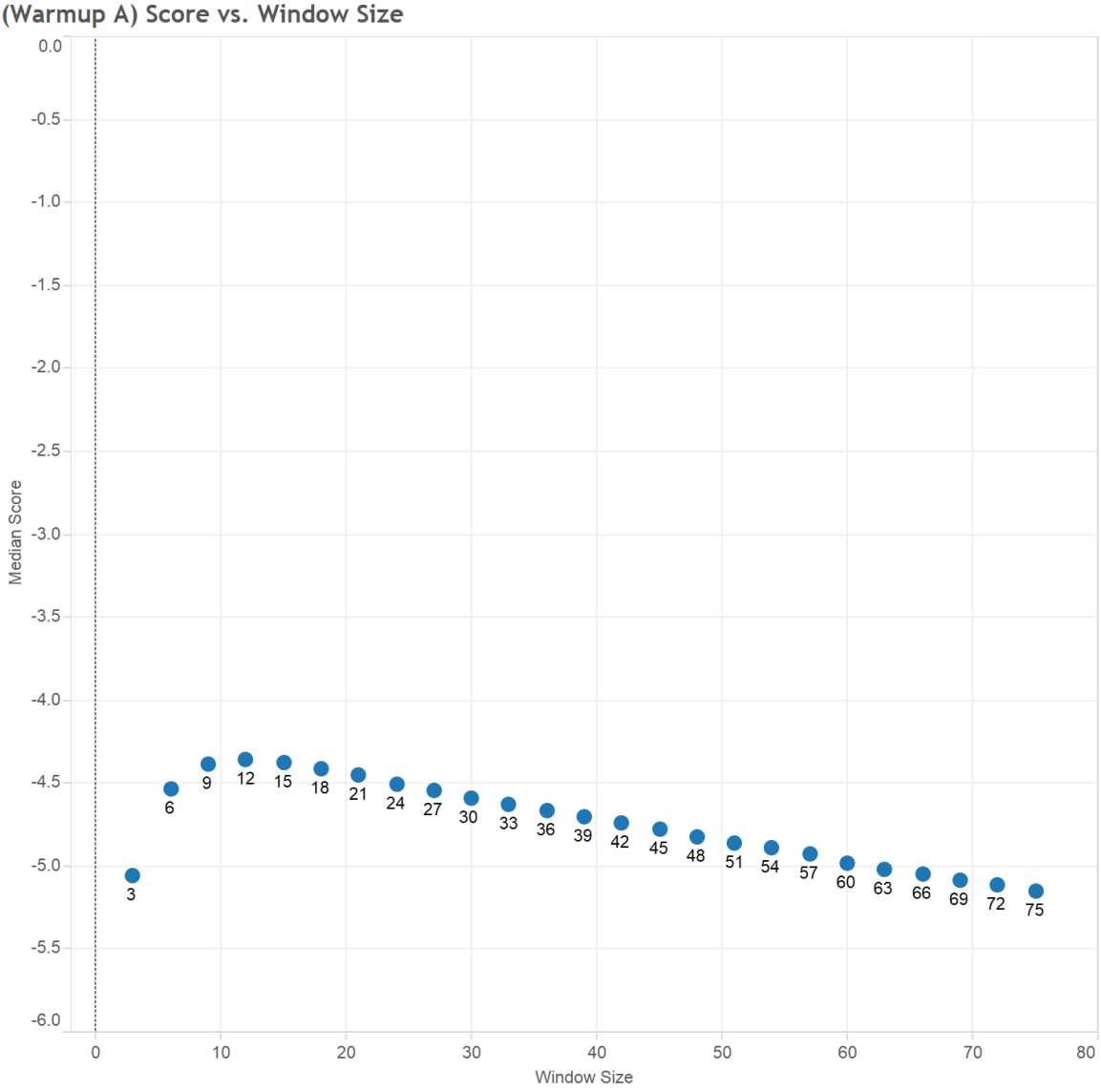
Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2

Below is a plot of throughput vs. 95th percentile signal delay for each window size. Notice that the delay axis is reversed. The window sizes are displayed along with their points.



Based on the data I collected, I found that my best fixed window size was window size **12**, which achieved a score of approximately −4.3596. The chart below plots score vs. window size, where the labels are the window sizes.

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2



Warmup Exercise B

I implemented AIMD in `controller_aimd.cc`, `controller_aimd.hh`, and `sender_aimd.cc`. I detect timeout using these two methods (using just one yielded a lower score):

1. Check if I get `PollResult::Timeout` in `sender.cc`
2. In `ack_received` in `controller_aimd.cc`, check if `timestamp_ack_received - send_timestamp_acked < timeout_ms()`

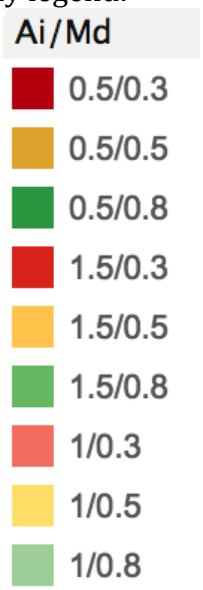
In both the cases above, I perform a multiplicative decrease. If they second case does not hold, then I perform an additive increase when I receive an ACK.

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2
Specifically, the update is:

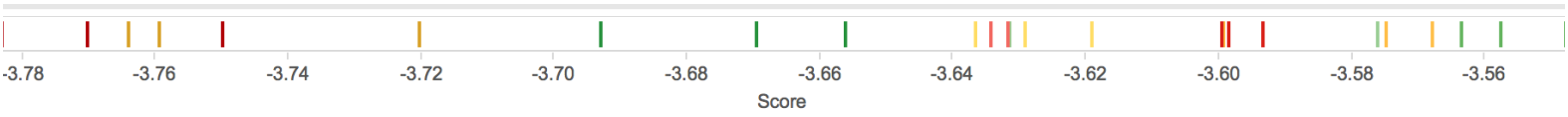
$$cwnd \leftarrow \begin{cases} MD\_CONST \times cwnd & \text{timeout} \\ cnwd + \dfrac{AI\_CONST}{cwnd} & \text{successful packet ACK} \end{cases}$$

I chose `MD_CONST` from $\{0.3, 0.5, 0.8\}$ and `AI_CONST` from $\{0.5, 1, 1.5\}$. I took the Cartesian product of these two sets as my search space. I tried a few choices for `timeout_ms()`, and found that 90 milliseconds worked well. I chose this because many of the high scoring algorithms on the leaderboard had a 95th percentile signal delay around that value. I ran three simulations for each pair of constants.

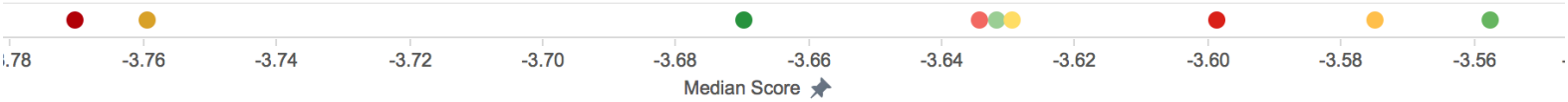
Before I display the plots, here is my legend:



Here are the scores:

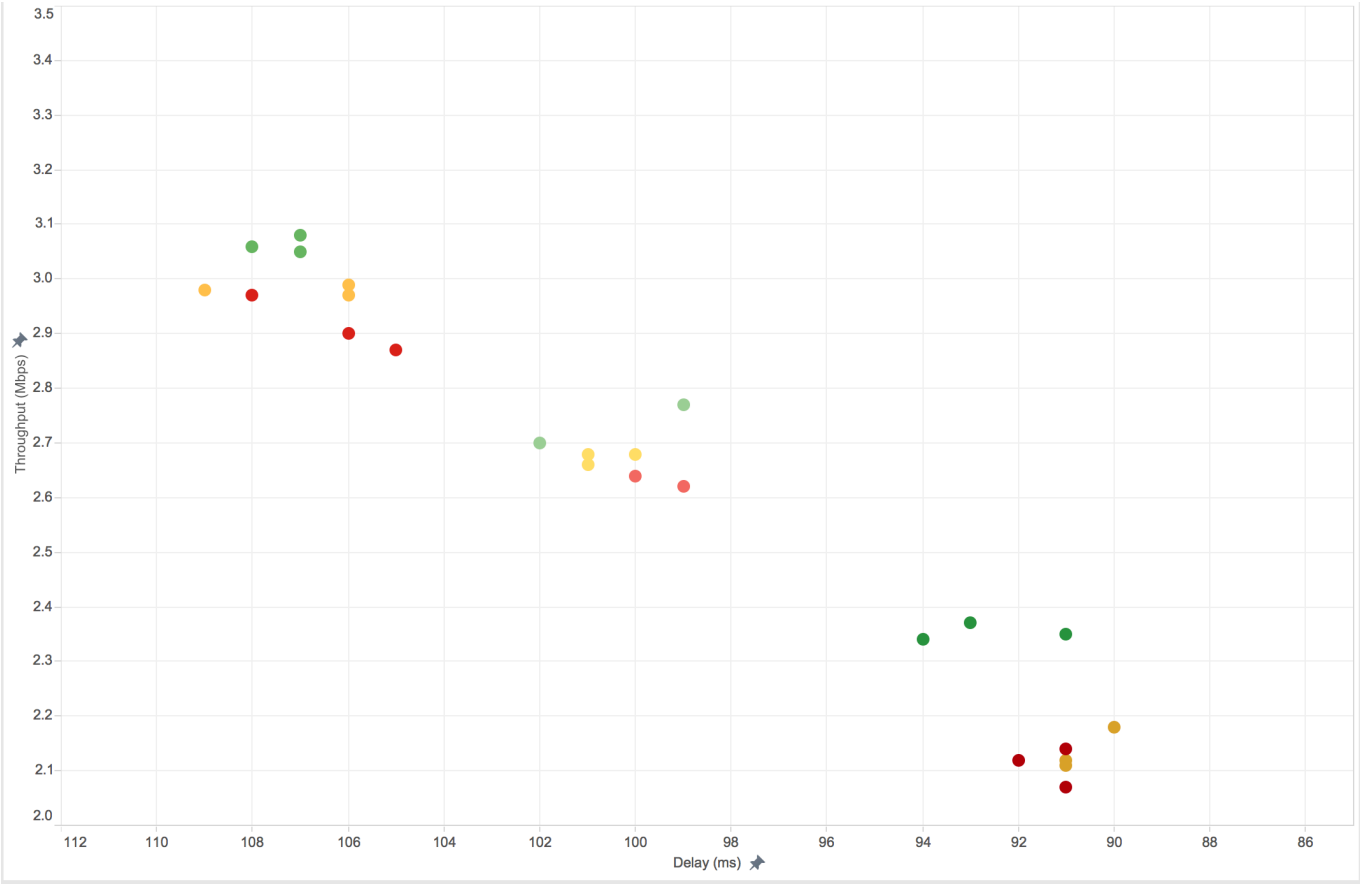


Taking the median scores gives:



So $(AI\_CONST, MD\_CONST) = (1.5, 0.8), (1.5, 05)$ are the first and second best, respectively. The throughput vs. delay plot looks like this (after zooming in)

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2



Warmup Exercise C

I implemented a delay-triggered scheme in `controller_delay.cc`, `controller_delay.hh`, and `sender_delay.cc`.

On `ack_received` in `controller_delay.cc`, I estimate the latest RTT using `timestamp_ack_received - send_timestamp_acked`. Then, I use an EWMA update to incorporate the latest RTT into the overall RTT estimate.

Then, if the RTT is below a threshold, then I use additive increase and if it's above the threshold then I do multiplicative decrease. I also use the technique from Exercise D where I require at least `MD_BUFFER_SIZE` (set to value = 500) milliseconds elapse between multiplicative decreases (see the `mdbf` optimization from Exercise D).

To choose a good delay-triggered algorithm, I could have performed a grid search where I varied scheme (AIMD, MIMD, MIAD, AIAID), scheme constants (e.g. the `AI_CONST` parameter), threshold, `MD_BUFFER_SIZE`, and more. However, I

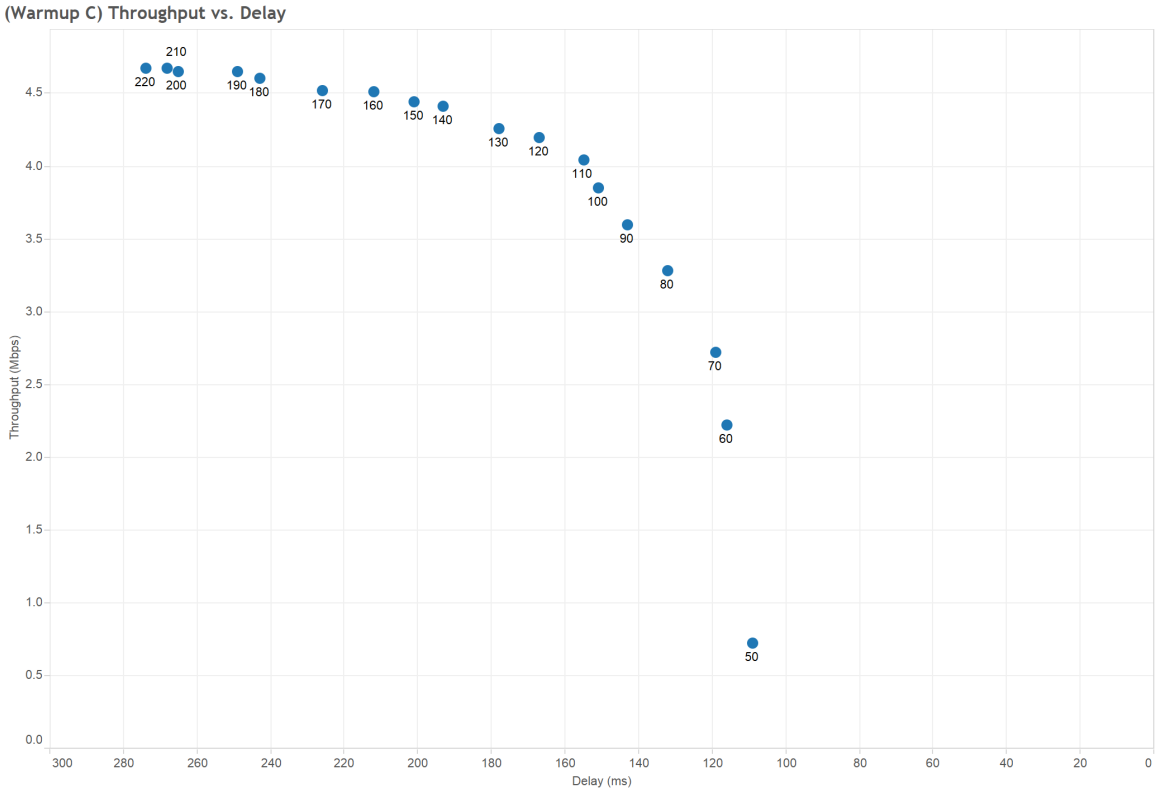
Harihar Subramanyam
hsubrama@mit.edu

decided not to do this because the search space would have been very large and would have taken a very long time to search it.

Thus, I decided to simplify the search process with three decisions:

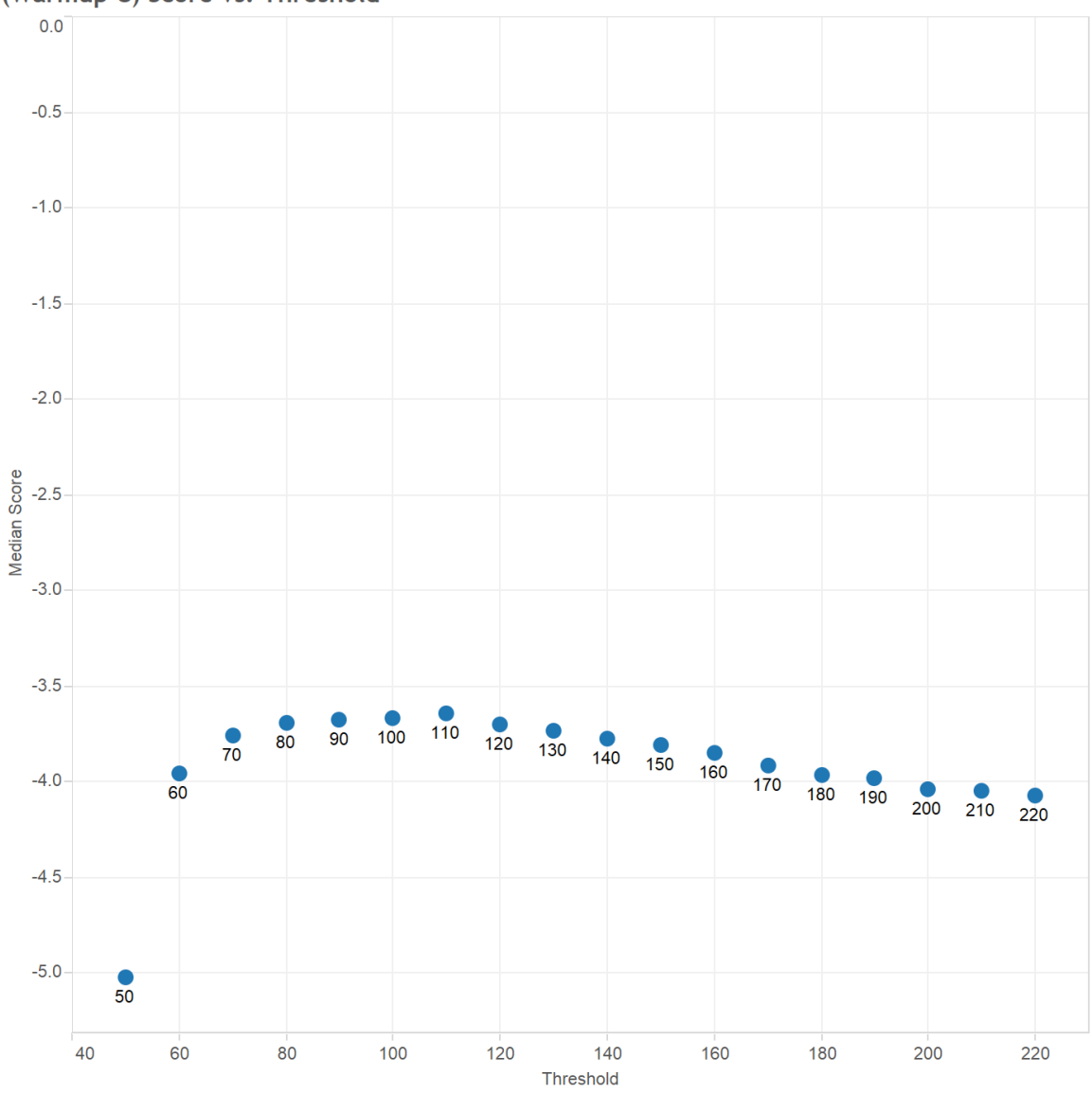
1. I fixed the `MD_BUFFER_SIZE` at 500 milliseconds, minimum window size at 1, and timeout at 1000 milliseconds, EWMA weight at 0.1 (seemed to work well)
2. After playing around with a few delay-triggered MIMD, AIMD, MIAD, and AIMD schemes, I found that delay-triggered AIMD outperformed the other schemes. So, I decided that I would use an AIMD scheme.
3. I decided to fix the AIMD constants at the standard $MD\_CONST = \frac{1}{2}$ and $AI\_CONST = 1$ and vary the threshold to find the optimal threshold. Then, I decided to fix that optimal threshold and vary the AIMD constants.

I varied the threshold from 50 to 220 in increments of 10. For each threshold, I ran the simulation three times and took the median throughput and median 95th percentile signal delay. My resulting plot is shown below (the threshold values are indicated alongside their points).



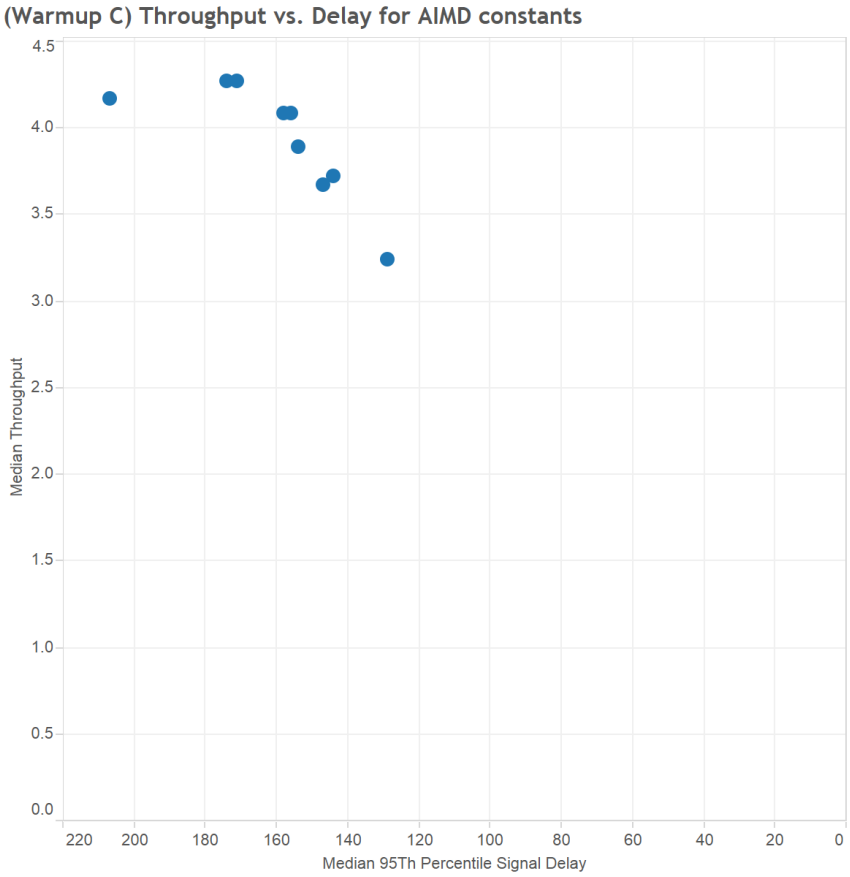
The best choice of threshold was 110, with a score of -3.646. The plot of score vs. threshold is shown below, where points are labeled with their threshold value.

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2



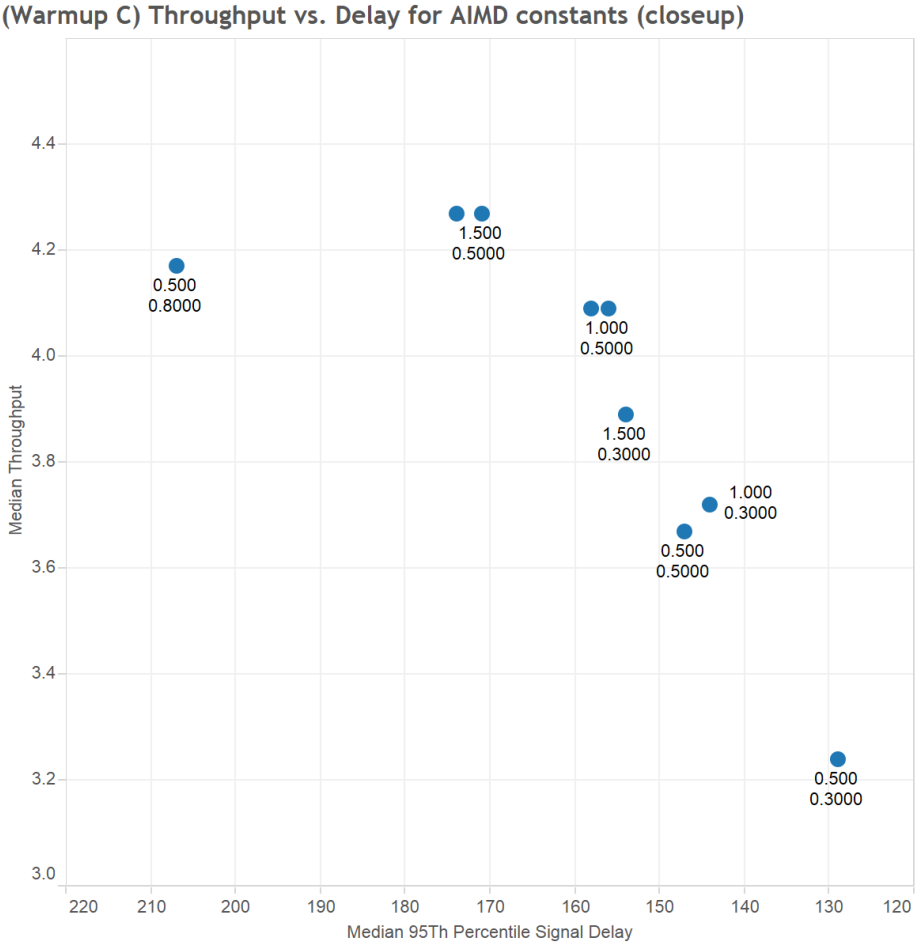
Taking the threshold fixed at 110, I decided to vary the AI constant along the set $\{0.5, 1, 1.5\}$ and varied the MD constant along the set $\{0.3, 0.5, 0.8\}$. The Cartesian product of these two sets contains nine elements. I tried each of these configurations and ran the simulation three times, taking the median throughput and median 95th percentile signal delay. The resulting plot is shown below, where each point corresponds with one AIMD pair.

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2



Here is a close-up of those points, with their AI constant displayed above their MD constant.

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2



Now looking at the score for each AI/MD constant pair gives.

(Warmup C) Score for AIMD constants

	md		
ai	**0.3**	**0.5**	**0.8**
0.5	-3.6842	-3.6957	-3.8976
1	-3.6561	-3.6398	-3.6516
1.5	-3.6837	-3.6900	-3.7121

The highest score is achieved by an additive increase of 1 every RTT and a multiplicative decrease of 0.5 (i.e. halving the window).

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2

Exercise D

I started with AIMD used a number of optimizations to improve its performance. My code is in `sender_custom.cc`, `controller_custom.cc`, and `controller_custom.hh`.

Denote the vanilla AIMD algorithm as **`aimd`**. This vanilla algorithm does the following:

$$cwnd \leftarrow \begin{cases} \frac{1}{2} cwnd & \text{timeout} \\ cnwd + \frac{1}{cwnd} & \text{sucessful packet ACK} \end{cases}$$

I say that a timeout has occurred when either 1) an AKC for a packet arrives with RTT > `TIMEOUT` (90 milliseconds) or when I have gone `TIMEOUT` milliseconds without receiving any ACKs.

Here are my optimizations, with their name in parenthesis:

Buffer time between multiplicative decreases (`mdbf`):
I noticed that packets sometimes came in groups and had similar RTTs. Thus, many multiplicative decreases can happen in a short time, which can cut the window size too aggressively. So, I use a constant `MD_BUFFER_TIME` so that two multiplicative decreases do not happen within `MD_BUFFER_TIME` milliseconds of each other. This prevents clustering of multiplicative decreases.

Ratio based multiplicative decrease (`rat`):
Using a fixed multiplicative decrease constant led to the congestion window being cut too aggressively sometimes (wasted capacity) and not aggressively enough other times (high delay). So, I decided to make the multiplicative decrease depend on the RTT of the packet. Specifically, on packet timeout, I compute the value `ratio` by taking the packet RTT as a fraction of the timeout, multiplying by a scaling constant, and constraining the ratio to be at least 1 (to avoid increasing the congestion window size).

```
ratio = min(1, rtt_packet/timeout * MD_RATIO_SCALER)
```

Then, my multiplicative decrease step is:

```
cwnd = cwnd/ratio
```

This decreases the congestion window aggressively when the packet RTT is huge and decreases it conservatively when the packet RTT is close to the timeout. I found that `MD_RATIO_SCALER` = 1.5 worked well.

Growing additive increase (`grw`):

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2

Looking at the traces, I found that, sometimes, the channel capacity suddenly increases by a large amount. Using a fixed additive increase constant can cause the congestion window to grow too slowly and thus waste capacity. So, I modified the additive increase step from:

```
cwnd += AI_CONST/cwnd
```

to

```
ai += AI_GROWTH_RATE

cwnd += ai/cwnd
```

I set `cwnd` to `AI_CONST` on start-up and on multiplicative decrease.

This change allows the congestion window to grow more rapidly when it has been a while since the last multiplicative decrease. This makes better use of channel capacity.

Tracking packets in flight (`fly`):

Initially, I performed multiplicative decrease on receiving an ACK if the packet's RTT was above my timeout. However, rather than waiting for a packet to return before I react to its RTT, I instead chose to keep a table that maps packet sequence number to the timestamp at which the packet was sent. Then, I periodically scan this table to if there are any packets that have been gone for a long time (i.e. current time - send time > threshold value). If so, then I perform a multiplicative decrease. By using this approach, I don't need to wait for the packet to return in order to perform a multiplicative decrease; I can react sooner to falling channel capacity. This results in lower delay.

Don't start with cwnd = 1 (`wind`):

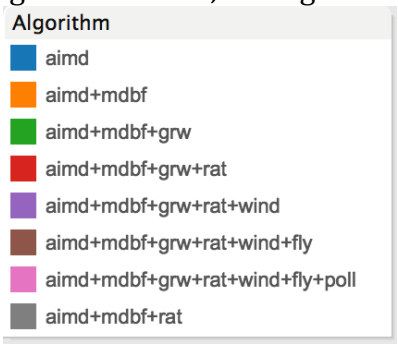
What congestion window should the controller start with? Obviously, I do not want to use a huge value like 50. At the same time, I do not want to be too conservative and pick an initial window size of 1. After all, it is reasonable to think that when the controller first starts sending packets that the link capacity will be neither amazing nor awful. In my analysis of fixed window size, I found that the best fixed window size was 12. So, I decided to cut that roughly in half and start with a window size of 5.

Poll faster than timeout (`poll`):

In the original code, `sender.cc` calls the poll method with an argument of `controller_.timeout_ms()`. However, I'd like the system to be quicker to react. So, I separated the packet timeout and the poll interval. Now, the poll interval is smaller, which allows the controller to perform the `fly` optimization more frequently.

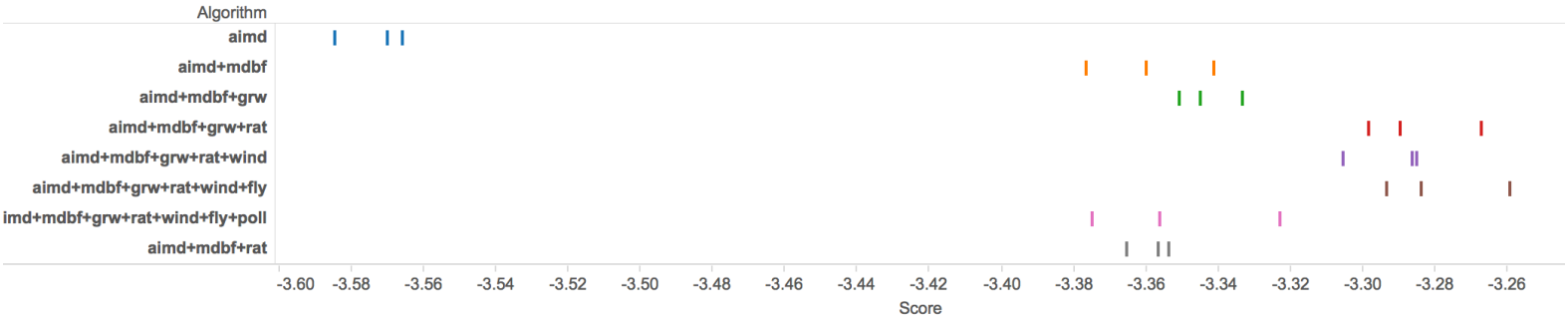
Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2

Here are the results of my algorithms. First, the legend for my plots is as follows:

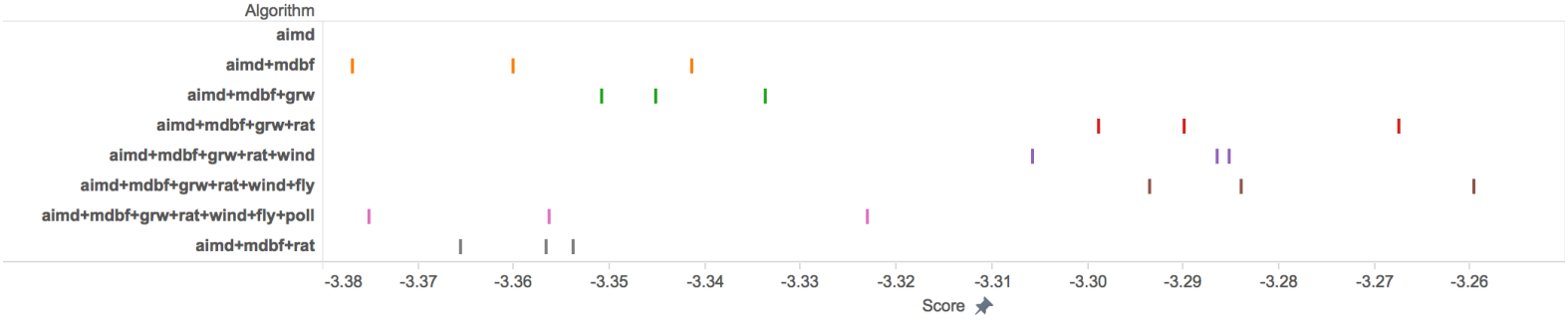


For example `aimd+mdbf+rat` is vanilla AIMD augmented with the `mdbf` and `rat` optimizations.

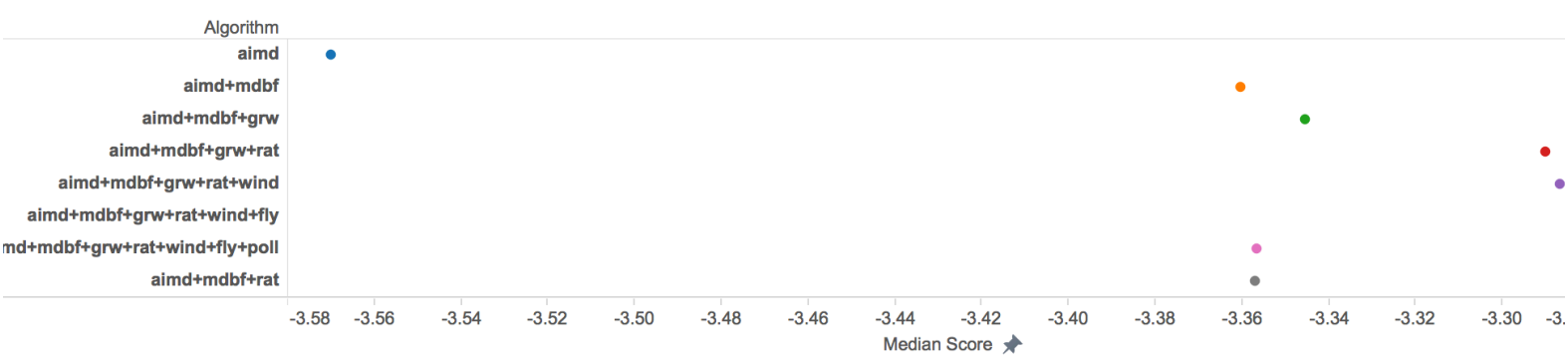
After running each of algorithms in the legend three times each, I plotted their scores:



As you can see, vanilla AIMD performs very poorly. Zooming in on the better algorithms gives:

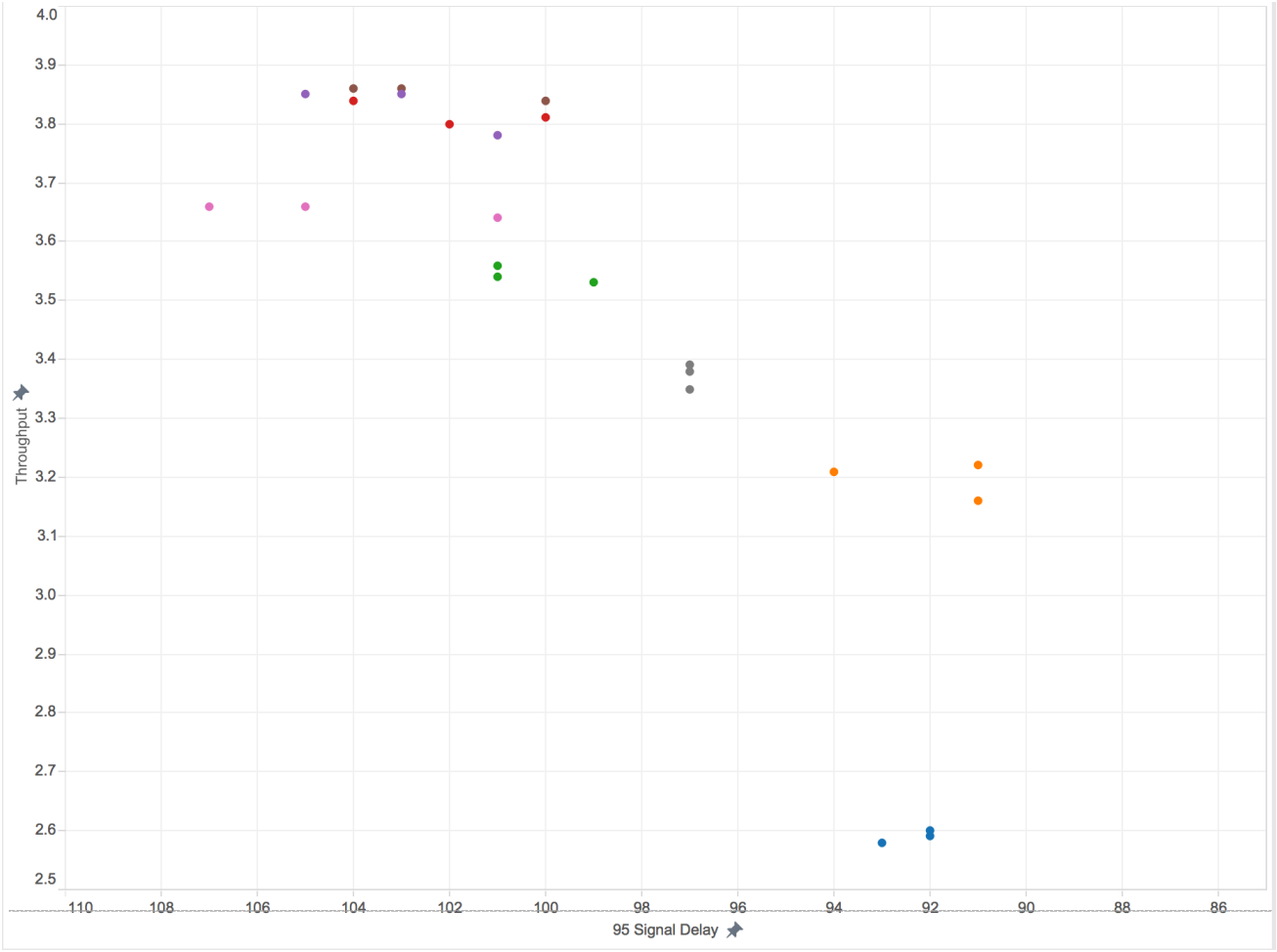


Plotting the median score instead of each of the three scores gives:



Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2

So, the best performing algorithm was `aimd+mdbf+grw+rat+wind+fly`. In other words, all my optimizations except the `poll` optimization were helpful in improving performance. The `mdbf` optimization was the single most impactful optimization. To understand why, let's take a look at the throughput vs. delay plot (I have zoomed in on my algorithms):



So, the `aimd+mdbf` algorithm has roughly the same delay as the `aimd` algorithm, but it has much higher throughput. This is because `aimd+mdbf` does not cut the window size as aggressively as `aimd` does (recall that `mdbf` requires a minimum time elapsed between multiplicative decreases).

The `aimd+mdbf+grw+rat+wind+fly` algorithm did much better than the `aimd+mdbf+grw+rat+wind+fly+poll` algorithm. The `poll` optimization did little to change delay, but severely hurt throughput. Recall that the `poll` optimization involved increasing the frequency at which the table of outstanding packets is scanned and multiplicative decreases performed. Evidently, this was done so frequently that the congestion window was too small, which reduced throughput.

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2
Overall, at peak performance, my best algorithm outperformed Sprout on the leaderboard and achieved a throughput of 3.88 Mbps, a 95th percentile signal delay of 103 ms, and a power score of 37.67.

Exercise D': Reinforcement Learning (didn't work)

I tried approaching the problem using reinforcement learning, but I didn't get very far. The performance was bad and the algorithm was difficult to tune. That being said, I want to include my approach in case the reader finds it interesting.

Before I did any reinforcement learning, I first wanted a way to estimate throughput and delay. I decided to approximate delay as $\frac{1}{2}$RTT.

To estimate the RTT, I computed the `timestamp_ack_received - send_timestamp_acked` and used it in an EWMA update.

To estimate the throughput, I broke time into chunks of length `EPOCH` milliseconds. Then, within each epoch, I counted the number of received ACKs and estimated the throughput, in packets per second, using `num_packets_in_epoch_ / EPOCH`. I used EWMA here as well.

Now, let's get to the reinforcement learning. Reinforcement learning is applicable in situations where an **agent** observes the **state** of its **environment** and chooses an **action**, which yields a **reward** from the environment. The agent aims to learn a **policy**, which maps states to actions, that maximizes its **value**, or long-run cumulative reward. This closely matches the situation in this problem set:

- **Agent** = controller
- **Environment** = network
- **Action** = congestion window size
- **Reward** = computed from previous throughput and delay
- **State** = throughput and delay

I chose to use the **Sarsa** reinforcement learning algorithm, which is simple to implement and requires very little storage and computational resources. The controller maintains a table $Q(S, A)$, called the **action-value function**, which approximates the **value** if the controller takes action A when in state S. The controller uses an ***ϵ*-greedy policy** π that maps state to action. The ϵ-greedy means that when the agent in state S, it will pick a random action with probability ϵ and will otherwise greedily pick the action that maximizes the value:

$$A^* = \arg\max_A Q(S, A)$$

When the controller chooses action A in state S and ends up in state S' with reward R, the controller updates its estimate of the value for the state-action pair.

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q\left(S', \pi(S')\right) - Q(S,A)\right]$$

for constants α and γ. Notice that this update takes the form:

$$New\ Value \leftarrow Old\ Value + (Step\ Size)[Estimated\ Value\ after\ Taking\ Action - Old\ Value]$$

The step size $0 < \alpha \leq 1$ weights how much the latest experience affects the estimated value for the state-action pair, $Q(S,A)$.

The $0 < \gamma \leq 1$ represents the **discount rate**. This codifies the assumption that a reward now is better than a reward in the future. Specifically, a reward of 1 in the next time step is worth the same as a reward of γ now.

For tabular Sarsa, which I used, I needed to answer the following questions:

1. How will I discretize time?

Since the throughput is estimated every **`EPOCH` milliseconds**, I decided to make the controller act (i.e. update the congestion window, update $Q(S,A)$) every epoch.

2. How will I discretize the state space?

I decided that the current state should be the (throughput, delay) pair. I needed to discretize this so I first bucketed throughput into LOW, MED, and HIGH buckets. Then, I did the same for delay. Finally, I took the two bucketings and produced a number from 0-8 that represented the state.

3. How will I discretize the action space?

I decided there would be 9 possible actions. These actions would pick a congestion window size linearly from the range $[0, 15]$ and took the maximum of the chosen value and a minimum congestion window size of 1.

4. How will I compute reward?

For each of $|\{LOW, MED, HIGH\} \times \{LOW, MED, HIGH\}| = 9$ states, I manually selected a reward. I gave the highest reward to the high throughput, low delay state and gave the lowest reward to the low throughput, high delay state. Another option I tried was to compute the reward as $\ln \frac{\text{throughput}}{delay}$.

5. How will I pick the discount rate?

I chose a discount rate of $\gamma = 0.9$. This was arbitrary.

Harihar Subramanyam
hsubrama@mit.edu
6.829 PS 2

6. How will I pick the step size?

I chose a step size of $\alpha = 0.4$. This is quite large, so it weights new data heavily. I chose a large value because the link capacity changes a lot and changes often, so I wanted the controller to be quick to adapt.

7. How will I pick ϵ?

I chose $\epsilon = 0.05$. I initially tried 0.1 and that was poor. Then I tried 0.01 and that was also poor. So, I chose 0.05 and it performed a little better.

I didn't spend a lot of time with the reinforcement learning approach because of four key problems.

First, as you can see above, there are a lot of parameters to tune and it's not clear how to tune them. I did not have time to run a massive grid search cross validation to tweak the parameters.

Second, discretizing states, actions, and rewards involved looking at delay and throughput at a very coarse granularity. A better approach would have been to learn a supervised regression model (e.g. linear regression, deep neural network) to represent the $Q(S, A)$ function. However, this makes the algorithm considerably more complicated and compute-intensive.

Third, reinforcement learning requires a lot of training data, and I did not have time to run the simulation thousands or millions of times.

Fourth, my ad-hoc algorithm from Exercise D performed much better, was easier to understand, and easier to improve. So, I decided to devote more time to that instead.

Ultimately, the reinforcement learning algorithm did not perform well, with a throughput of roughly 1.5 Mbits/sec and a delay of 109 ms. However, it may be possible to improve on my approach and create a highly performant algorithm that views the problem through the lens of reinforcement learning.