MTL Constraints with Abstract MDP in Value Iteration

Michael Yuen

April 27 2020

1 Introduction

A Markov Decision Process (MDP) is a structured way of formulating a problem, which is initialized with a state space, action space, transition probabilities, and reward function. Value Iteration is one method of solving such a problem, which calculates a utility per state. These utilities can then be formulated into a policy which determines the best action per state. On a large state space however, many utilities need to be calculated, and many probabilities would exist between states, hence the state explosion problem exists in many practical MDPs. This is where the abstract MDP comes into play. By making the MDP a graph formulation and splitting the graph through a graph partitioning algorithm, Value Iteration would need to run for less iterations based on a smaller abstract state space, which in turn requires a smaller set of transition probabilities. Another issue with Value Iteration is that the utilities and policy found are not accurate- this requires using Metric Temporal Logic (MTL), which is a similar logic framework to Linear Temporal Logic. Metric Temporal Logic uses time-constrained operators with intervals to set boundaries on values; for example, utilities should not be too far apart from one another, causing issues between policies if some value is calculated very far off from the correct value in the Abstract MDP. Metric Temporal Logic has an open source package that has each operator implemented, adding more than just Linear Temporal Logic's operators that can be used for solving an MDP in a specified way. Creating an Abstract MDP and using MTL constraints allow greater speed and accuracy in Value Iteration.

2 Technical Approach

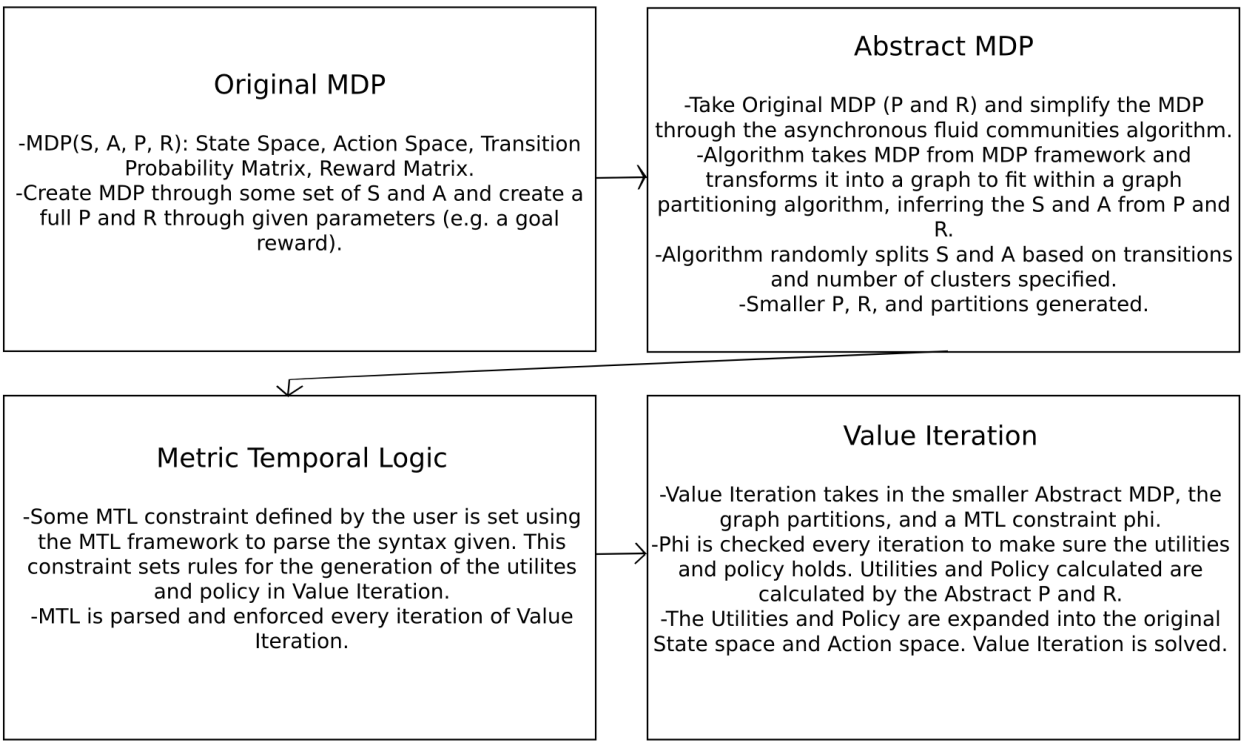

Figure 1: MDP Pipeline

The MDP pipeline is as follows: input an MDP with a given State Space (S), Action Space (A), and computed Probability Matrix (P) and Reward Matrix (R). The P and R are calculated based on the given S and A, along with additional parameters; such as the size of the State Space, the probabilities of each Action from State 1 to State 2, the start and goal states, states that give additional reward, and other necessary parameters to generate an implicit S and A through the P and R. P is a size A * S * S matrix that contains a State Space * State Space sized matrix per Action. Each Action holds the probability of going to a particular State from a given State; for example, State 1 can have 0.1 probability of going to State 2 (e.g. going right in a grid world environment), have 0.8 probability going back to State 1 (e.g. going left and up from State 1), and 0.1 probability going to State 5 (e.g. going down). R is a size S * A matrix, which each State has an array the size of the Action Space. The numbers in the array of R show the reward gained from an Action at that particular State. For example, in State 1, there could be a -1 step cost reward for taking the Up Action in a grid world, which would end up in the same State, and 10 reward for taking the Down Action.

The next step is to feed the original MDP specification into the Abstract MDP generator, which takes the P and R and simplifies these two arrays into a smaller P and R. The State Space is made smaller by the number of clusters specified by the user. The MDP is formulated into a graph by creating an edge from each Transition. The vertices are States from the State Space, and each edge has edge information on estimated Reward gain and Transition Probability of taking that Action between two States. The Graph is fed into the asynchronous fluid communities algorithm, which randomly partitions a graph based on the random seed given and the number of clusters to split the graph into. The clusters must allow the array of P to maintain stochastic properties in order to run Value Iteration on the MDP- P and R must still be an MDP based on the number of clusters the graph has been split into. If the graph is asked to be split into more clusters than it could possibly split into, the algorithm will fail. The algorithm initializes a number of communities based on this number of clusters, iterating randomly until each vertex is added to a community evenly, in which each community has a density distributing the vertices contained. Once no vertex changes communities, the algorithm converges, resulting in the partitioning of the graph. The partitions are still in graph form however, so translation of these partitions from vertices to a smaller P and R is necessary. This is done through taking the respective vertices and combining P and R based on the vertex communities, checking each value of P and R to simplify each set of States into singular rows. An image of the original MDP of a simple Forest Fire simulation is shown below, translated to an abstract MDP.

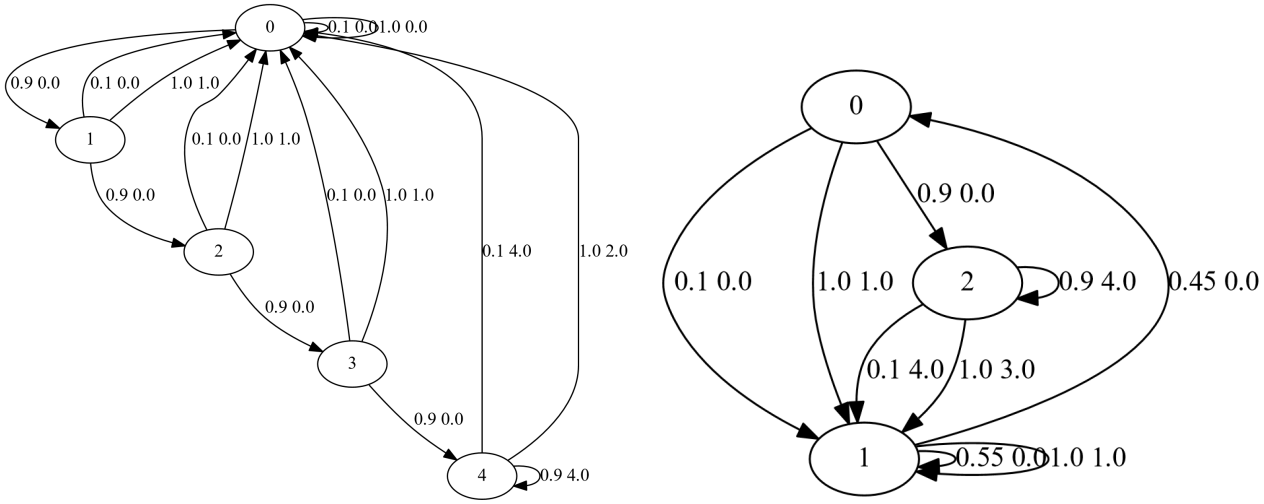

Figure 2: Original MDP (left) vs Abstract MDP (right)

The Forest Fire Management simulation is an MDP problem in which the State Space is of different States of a forest, from a newly grown forest to an older forest. At any State, the agent can choose to cut or wait, either cutting the forest back to the initial newly grown forest, or waiting until the forest grows old enough, generally for more value, in this case a reward of 2 rather than 1, before cutting. Constantly waiting also gives a reward of 4 at the oldest State, unless the forest burns. When waiting, there is a chance of 0.1 that the forest could catch fire, resetting the State to the initial State. The Original MDP on the left shows a Forest Fire simulation with 5 States, with the initial State being State 0 and the final oldest State being State 4. The Abstract MDP represents this exact MDP by placing States 0 and 1 in State 1, States 3 and 4 in State 2, and State 2 in State 0. As stated above, constantly waiting gives a reward of 4, as shown by the self loop in State 2 along with the slight chance of being reset to State 1. State 1 only has a 0.45 chance of moving on to

State 0, which is representative of State 2 in the Original MDP, as States 0 and 1 from the Original MDP are represented in State 1. If the agent makes the decision to wait in the Original State 0, it may end up in the Original State 0 or 1, which both are considered State 1 in this Abstract MDP, hence the self loop exists to show this transition. This example highlights the effects of the Original MDP conversion to the Abstract MDP: the asynchronous fluid communities algorithm condenses the Original MDP into this smaller Abstract MDP, making Value Iteration run much quicker and only slightly less accurate, which can be fixable with proper use of MTL constraints to force the calculated policy to be optimal.

Metric Temporal Logic is used to enforce precision in the policy by enforcing certain constraints. This is done through a user inputted formula phi, which is parsed through a set of syntax, checking for situations such as whether the policy does a certain Action at a certain State, whether the policy has Actions that lead to non-terminal States repeatedly, or whether the policy has Actions that are optimal and reach the terminal State quickly with few Actions. The policy phi is fed into Value Iteration, in which every iteration would take this formula and make sure the policy enforces the formula. This formula is checked on a dictionary with keys as each State and values as a tuple of the State that that particular State goes to when the State takes an Action, which is the result of the policy. The metric used for the time based logic is per state, in which moving between States is considered the metric for time, and users can then input a formula phi to check on whether a particular action from the policy is a good action.

Value Iteration is then run in three ways, with just the Abstract MDP, with just the Metric Temporal Logic, and with both additions. For the Abstract MDP, Value Iteration is directly ran on the Abstract MDP and creates a smaller list of utilities and policy. These smaller lists are mapped based on the graph partitions of the Original MDP. Each State from the Abstract MDP is split back to the Original MDP, in which the policy is duplicated across the States in the Original MDP depending on which partition each State was in. Metric Temporal Logic parses the syntax of the formula phi and feeds it into Value Iteration; this formula is checked based on the way the policy keeps this logic formula true when the formula must be held. If the formula needs to be held consistently throughout every run of Value Iteration, the formula is checked every single Iteration, and if the formula only needs to be checked during certain intervals, the policy is checked to see if the formula holds during those intervals. This operation is done through creating a dictionary to hold states connected to the action taken in that State from the policy, creating a metric through each State. The formula phi is run on this set of data, which until phi is true and the algorithm reaches max iterations, Value Iteration doesn't end. Value Iteration can do both of these operations at once, as Abstract MDP's translation is only done after Value Iteration stops iterating, and the MTL constraint is checked every single iteration, depending on what the constraint is.

3 Software Used and Written

The software is written in Python, hence all packages are from Python. The basic packages of Numpy, Matplotlib, Scipy, and built-in Python packages were used for array manipulation, timing different parts of the code, and for visualization windows. Mdptoolbox was used as an MDP solver with Value Iteration, along with an easy way to create a streamlined way of creating MDP examples and scenarios. Mdptoolbox has the pipeline of creating arrays P and R to represent the States and Actions, which the P and R are fed into Value Iteration to solve for the utilities and policy. Mdptoolbox has some baseline MDPs, such as the Forest Fire Management example and a simpler MDP example that has a handful of States and Actions. Graphviz is used to visualize MDPs by creating a graph with States and Actions from the P and R arrays. The Original and Abstract MDP are drawn with this python package. Networkx is used in order to use the implemented asychronous fluid communities algorithm, along with trying out other graph partitioning algorithms such as bi-partitioning and k-clique partitioning. With some trial and error on simple MDPs, asynchronous fluid communities worked on multi directional graphs and connected communities based on edges, making communities that made sense. However, to use this package, the P and R arrays must be converted to an actual graph with edges and vertices, rather than an implicit graph built between transitions. The last package used is py-metric-temporal-logic, which is a library that parses MTL operators, such as until, next, and finally into a model checker, which these parsed formulas can then check data for whether the formula holds appropriately. This can be used on the policy and utility values to make sure that these formulas are held by the policy found by Value Iteration.

Software that I wrote include a Simple Grid World MDP, which generates an MDP from a given goal with a reward and a step cost. Actions include going up, down, left, and right, in which there is a 0.7 probability to go one direction and 0.1 probability to go in any other direction. The agent's goal is to move from the start State to the goal State. The more advanced Grid World MDP with flags is generated by considering a fixed reward per flag. The MDP is generated through a number of specified flags with their specified rewards, along

with the goal state, reward of the goal, and step cost. Probabilities, Action space, and State space are all the same, but the rewards are different to allow for more specific MTL formulas and a more complicated optimal policy. The agent's goal is to go from the start to the goal State while collecting these flags in a particular order or to get the best reward based on these flags. The Rock Collection MDP is an MDP example used in similar papers about MDP research. The example is similar to both Grid World examples, in which there are a set of good rocks and bad rocks giving a good reward or bad reward respectively. The Probability, Action, and State space are all the same, and the goal is to get from the start State to goal State, in which the agent gathers as many good rocks as possible without gathering bad rocks. Just like the flag based Grid World, this allows for many types of MTL constraints to instruct the policy to get the right number of good rocks, to avoid bad rocks at all costs, or to avoid all rocks and rush for the goal. Besides MDP examples, I had written the connection in the graph visualizer to parse graphs from the mdptoolbox framework to the graphviz framework. This allows for graphs to be visualized through just inputting the generated P and R from the MDP. I had also written the abstract MDP algorithm which parses graphs from the mdptoolbox framework to the networkx framework in order to fit the original MDP into the asynchronous fluid communities algorithm. The algorithm returns a networkx graph structure that isn't the same as mdptoolbox's MDP structure, so I had also written the mapping of the partitions into the Original MDP, generating the Abstract MDP. This algorithm has been partially vectorized, but lacks vectorization of the partitions into the Abstract MDP. I had written additional code in the Value Iteration code in order to convert the Abstract MDP's policy to the original sized policy. I had also written code in Value Iteration to build a mapping of State per policy index to a singular tuple containing the next State the current policy's action should send the current State to. This allows for the user to check this mapping with a formula phi in order to determine whether the mapping works in favor of the time constraint, using States as a metric. Lastly, I wrote a large number of tests and experiments to play around with different parameters, MTL formulas, and stress testing on large State Spaces.

4 Experiments and Evaluation

Test	**Parameters**	**Seconds**
Original MDP 5x5 Simple Grid World	N/A	0.9310
Abstract MDP 5x5 Simple Grid World	12 Clusters	0.4205
Original MDP 5 State Forest Management	N/A	0.7144
Abstract MDP 5 State Forest Management	3 Clusters	0.2158
Original MDP 50x50 Simple Grid World	N/A	1.6156
Abstract MDP 50x50 Simple Grid World	100 Clusters	8.2717
Original MDP 5x5 Flagged Grid World	4 Flags with Positive Reward	0.9330
Abstract MDP 5x5 Flagged Grid World	12 Clusters; 4 Flags with Positive Reward	0.4136
Original MDP 5x5 Rock Collection	2 Good rocks and 3 Bad rocks	0.8961
Abstract MDP 5x5 Rock Collection	12 Clusters; 2 Good rocks and 3 Bad rocks	0.4140
Original MDP 250x250 Simple Grid World	N/A	Cannot Finish
Abstract MDP 250x250 Simple Grid World	100 Clusters	Cannot Finish

By N/A, the chart states that there are no important parameters that might affect time.

The results of comparing the Abstract MDP Value Iteration versus the Original MDP Value Iteration are shown above. Each of the Grid World applications in a 5x5 Grid seemed to hold the same values, with Abstract MDP being faster than the Original MDP. However, the 50x50 Grid has the Abstract MDP algorithm taking much longer to solve Value Iteration- this is because the mapping of the values from the Original MDP's partitions to the Abstract MDP is not optimized or vectorized in code. The smaller Grid World examples work fine, as this portion of generating the Abstract MDP doesn't take significant time for smaller MDPs, and Value Iteration's time taken is much less with lesser States. With the largest MDP test of a 250x250 grid, both the

Original Value Iteration and Abstract Value Iteration cannot finish due to memory constraints. The Original MDP is still being constructed in the Abstract MDP's construction, as the Abstract MDP requires the Original MDP's construction to exist first before abstracting this MDP structure. 12 Clusters were used for every 5x5 Grid World, and 3 Clusters were used in the Forest Management example, since there are only 5 States versus 25 States in the 5x5 Grid World. The differences between each Grid World example are negligible, as there are minor numerical changes when generating the utilities, but the State Space and Action Space do not change. Something to note is that the Value Iteration part of the Abstract MDP always takes less time than the Original MDP's Value Iteration, and the asynchronous fluid communities algorithm also takes less time than the unchanged Value Iteration. Overall, more testing needs to be done after vectorizing the code in the Abstract MDP generation.

Test	**Parameters**	**Constraints**	**Seconds**
Original MDP 5 State Forest Management	N/A	State 0 implies moving to State 2 in 2 Actions	0.2157
Abstract MDP 5 State Forest Management	3 Clusters	State 0 implies moving to State 2 in 2 Actions	0.2082
Original MDP 50x50 Simple Grid World	(4,4) goal	State (4,3) implies moving to State (4,4) in 1 Action	9.5187
Abstract MDP 50x50 Simple Grid World	100 Clusters; (4,4) goal	State (4,3) implies moving to State (4,4) in 1 Action	7.8436
Original MDP 5x5 Flagged Grid World	4 Flags with Positive Reward	State (1,1) implies moving to State (1,0) in 1 Action (Flag at (1,0))	1.2206
Abstract MDP 5x5 Flagged Grid World	12 Clusters; 4 Flags with Positive Reward	State (1,1) implies moving to State (1,0) in 1 Action (Flag at (1,0))	0.8481
Original MDP 5x5 Rock Collection	2 Good rocks and 3 Bad rocks	State (1,1) implies moving to State (1,0) in 1 Action (Good Rock at (1,0), Avoiding Bad Rock at (0,1))	0.8149
Abstract MDP 5x5 Rock Collection	12 Clusters; 2 Good rocks and 3 Bad rocks	State (1,1) implies moving to State (1,0) in 1 Action (Good Rock at (1,0), Avoiding Bad Rock at (0,1))	0.4396

The results above show that Abstracting the MDP speeds up the algorithm in comparison to just doing MTL constraints. MTL constraints are tested for every iteration, slowing down the Value Iteration algorithm if the policy doesn't meet the requirement quickly. The Abstract MDP allows for the State Space to be much smaller in order to have the requirement be met faster than the larger State Space. Different constraints can delay the algorithm more than others; checking the Forest Management States take much less time since there are only a couple States to check the constraints in and generate the mapping of States and Actions. Another example like checking whether a State is reached in one Action when there are more Actions and States in a Grid World example. These constraints are met in a longer amount of time. With some visual checking of policies, with the Abstract MDP and MTL constraints, the policy is slightly closer to the original Value Iteration results compared to just the Abstract MDP. Overall, optimization would clean up the algorithm's runtime and more testing is required to ensure that the accuracy of the policy is stronger with MTL constraints over the original Value Iteration.

5 Conclusion

Value Iteration solves for approximate solutions to MDP formulated problems through finding both utilities to estimate rewards and a policy to estimate the best Action per State. With the State explosion problem, there are more rewards and transitions, making P and R too large for Value Iteration. This makes Value Iteration slower and more memory intensive, requiring the Abstract MDP to simplify the P and R for Value Iteration. The Abstract MDP is created through graph partitioning using the asynchronous fluid communities algorithm

to equally split the graph based on the number of partitions set. This theoretically takes less time, but due to the different framework connections and lack of vectorization, the results show that larger MDPs take longer to map Original States to partitions created by the algorithm. Value Iteration takes around half the time afterwards with the Abstract MDP, though the time taken to generate the Abstract MDP is longer with larger MDPs. On smaller MDPs, the time taken is extremely short, since Value Iteration on a smaller MDP doesn't require any time on mapping the State space. The Abstract MDP does make Value Iteration less accurate, so MTL is necessary to force the policy to be more accurate while having a theoretically quicker run time. Metric Temporal Logic formulas allow for the user to specify how the policy should work and how Value Iteration can find the optimal policy. This is done with the MTL framework which requires a data structure to hold (time, value) tuples for the phi formula to determine whether the time constraints are satisfied. If the time constraints represented by States in the State Space were satisfied, Value Iteration just needs to reach the bound of change to end the algorithm. With a bit more time, the Abstract MDP can be vectorized to be made faster.

In regards to reward hacking, MTL can be set in a way to determine whether the policy runs in a way that promotes reward hacking. If the policy does, changing parameters of the MDP, such as reward values, allows for Value Iteration to fix issues such as reward hacking. However, in regards to automatically generating the reward function, an MDP specification requires user input in order to find the optimal reward, such as the inputs to how much reward an agent gains upon reaching the goal State. This problem is still not solved through the Abstract MDP and MTL constraints; the Abstract MDP and MTL constraints on Value Iteration allow Value Iteration to follow specifications that the user provides. The user still needs to input the complete MDP specification to generate the Abstract MDP. Future work can be done to try to generate the entire MDP itself, especially the rewards and transition probabilities in order to create a perfect policy every time with Value Iteration.

6 Appendix

Progress was slowed down significantly due to less than ideal working conditions (personal issues in family). As I had stated before in piazza posts, no matter what happens, I should still be able to finish my work, and if I am unable to (I deem that some of the work here is not finished or optimized yet), then I take full responsibility no matter the conditions.

Edit- 5/17/20 The report and progress should be fine now since I've spent a couple more days to make up for the lack of progress earlier.

Progress made every day since April 27th:

- 4/27/20 - Reviewed papers I had found along with figured out what are the best frameworks to use for the project.
- 4/28/20 - Played around with these frameworks (mdptoolbox, networkx).
- 4/29/20 - Built simple MDP example of gridworld. Read a bit to see similar metrics in papers, found Rock Collection example and Forest Fire Management example.
- 4/30/20 -Built flag MDP example of gridworld, pretty much built depending on where rewards are. Most MDP and RL oriented scenarios used do not change State or Action Space.
- 5/1/20 - Started document, wrote intro. Continued writing utility scripts and tests.
- 5/2/20 - Built Rock Collection and Forest Fire Management examples. Built graph visualization tool using graphviz after finding networkx couldn't support multi dimensional directed graphs.
- 5/3/20 - Spent day looking through graph partitioning and graph analysis tools. Wrote some of the pipeline section and diagram of pipeline.
- 5/4/20 - Built initial graph partitioning algorithm and found it extremely slow (only faster when the MDP is so small that it doesn't matter).
- 5/5/20 - Optimized the algorithm significantly, but still not fully vectorized and still slower than original MDP to value iteration (which takes 1 second on almost every case unless the case overflows memory).
- 5/6/20 - Practiced MTL constraints and tried using the py-metric-temporal-logic library.

- 5/7/20 - 5/10/20 - Lost lots of time due to family issues (parents were fighting, bit of violence). Was not able to get progress done since a lot of it involved me. Lost power for the day on 5/10/20 and most progress is on desktop. (Update: I am fine (physically) now as of 5/13/20 and I don't think this will be consistent, most likely just quarantine being tiring for the family.)
- 5/11/20 - Wrote software used, written, and finished Technical Approach, took pictures of graphviz MDPs.
- 5/12/20 - Wrote experiments and results, ran a few simple experiments to find abstract MDP still very slow, but nowhere near as bad as before.
- 5/13/20 - Wrap up of the paper, running more experiments and results. Tried to optimize the code for better results, not much gain.
- 5/14/20 - Proofread and submission.
- 5/15/20 - Brief planning of what to do next with MTL in Value Iteration.
- 5/16/20 - Playing around more with MTL constraints. Planned data structure to run the MTL constraint on in Value Iteration.
- 5/17/20 - Coded up the idea and added more to the report. Final version for now.