RL in Continuous State Spaces with Sparse Reward

Discretization for Model-Based RL

Students: Amit Tsvieli & Yazn Willy
Supervisor: Chen Tessler
Summer 2020

Control Robotics and Machine Learning Laboratory

THE ANDREW & ERNA VITERBI
FACULTY OF
ELECTRICAL
ENGINEERING

TECHNION
Israel Institute
of Technology

Table of Contents

Abstract

In continuous state spaces tabular methods cannot be applied directly. Thus, approximation approaches, such as function approximation methods, are used. These approaches often lack in exploration compared to classical tabular RL algorithms and thus perform badly in sparse reward domains. A way to apply classic discrete algorithms to continuous state-action spaces is by performing a discretization of the space.

As usual, a trade-off is present when performing discretization. A too fine discretization will produce an over complicated and large MDP, which will be computationally hard to solve. A too thick discretization will produce a MDP which does not correspond well enough to the underlying original dynamics.

We explored different discretization methods for different settings. Our goal was to find a method which balances the trade-off mentioned above. Upon a given discretized new space we acted according to R-max, which has optimal exploration.

We explored and present three setting-method pairs:

1. Solving a continuous space with known dimensions with uniform offline discretization.
2. Solving a continuous space with known dimensions with adaptive online discretization.
3. Solving a continuous space with unknown, possibly infinite, dimensions with adaptive online discretization.

Main results:

1. We implemented a solution for the first problem which converged.
2. We changed the method proposed in [1] to use R-Max and implemented other improvements, for the second problem. We achieved convergence.
3. We offered and implemented a method to extend the above solution to the third problem. Our method is general and easy to modify and use with other model-based algorithms.

We tested our solutions on a VIZDOOM navigation problem.

Introduction

Algorithms

Model-based algorithms

Construct an environment model, comprised of:

1. $R(s,a) - reward\ function$
2. $P(s'|s,a) - transition\ probability\ function$

Find a policy by solving the induced MDP, e.g. by value iteration.

R-max

Model based RL algorithm. R-max follows the optimism under uncertainty principal. R-max initializes the reward function to the maximal reward for all states. This initialization promises that each state will be visited at least once to exploit its initially high reward. It is possible to define any number of visits as the threshold to update a state's reward estimate. R-max solves the MDP based on the current model while continuing to explore. When all states are "known" – have a real estimate of the reward, R-max will naturally shift to a more exploitive policy. R-max has the optimal exploration property.

Model-free

Directly find a policy without constructing a model.

Q-Learning

Model free RL algorithm. Approximates the Q function by samples. The samples can be collected by acting according to any policy (off policy) or according to the current Q function estimate (on policy).

Each sample (s,a,r,s') will be updated by:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r + \max_{a'} Q(s',a'))$$

With α as a learning rate which normally decreases over time.

We collected the samples online by acting according to ϵ-greedy. At each time step the action is chosen according to:

$$\pi(s) = \begin{cases} random\ action, & w.p.\ \epsilon \\ argmax_a Q(s,a), & w.p.\ 1-\epsilon \end{cases}$$

With ϵ as an exploration rate which normally decreases over time. Q-Learning normally lacks in exploration and especially in sparse reward environments.

Discretization methods

Uniform

For every axis, a constant discretization granularity is chosen – Δ. An axis with dimensions $[min, max]$ be divided to $\frac{\max - \min}{\Delta}$ buckets of size Δ. Every observation from the continuous environment will be bucketized first, and then passed to the algorithm as a state.

The number of buckets increases as Δ decreases. The buckets serve as the new state space. A too small Δ will produce an over complicated and large MDP, which will be computationally hard to solve. A too big Δ will produce an MDP which does not correspond well enough to the environment's dynamics and may not converge to a good result.

Adaptive

Perform discretization with different granularity at different parts of the state space. The granularity can be decided according to different factors:

1. Number of visits
2. Reward estimate - $r(s, a)$
3. Value estimate – $V(s)$
4. TD Error - $r(s, a) + V(s') - V(s)$

The factor which determines granularity needs to be considered in respect to the algorithm that will operate the agent.

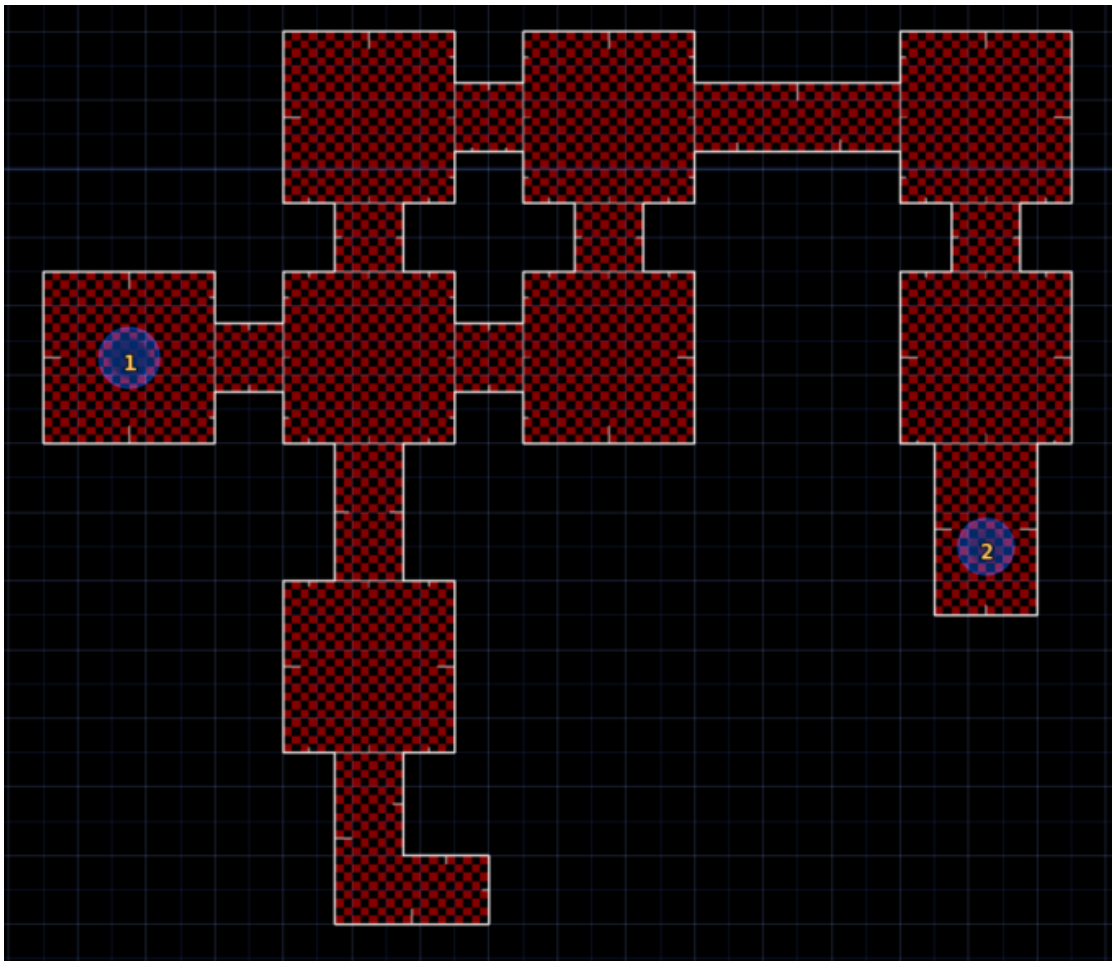
Adaptive discretization can ensure that the MDP will be less precise in constant parts of the state space and more precise in areas with changing behavior (changing over space, not time). Thus, balancing the tradeoff between computation and convergence to a good policy over the underlying environment.

Test Environment

VIZDOOM is a doom-based research platform. This platform has many abilities and act as a python interface to a doom game.

'My way home' scenario

In this scenario the player spawns at point '1' (but in a random angle) and needs to navigate to the green armor artifact, at point "2". The green armor grants a 1.0 reward and ends the episode. For each step, the player suffers a "living reward" of -0.05. The player has 3 available actions: turn and move right, turn and move left and move forward.

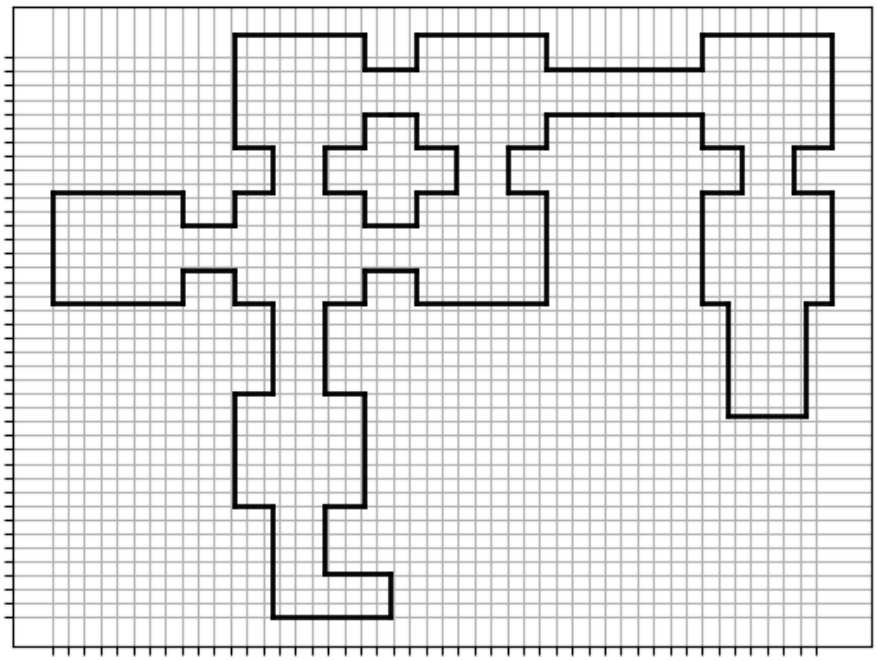


X range: 160 to 1120 Y range: -704 to 128.0

Solution and Implementation

Uniform

Each state is a triplet (x, y, θ). Below is the discretization for $\Delta = 20$, which has 48x44x4 buckets.



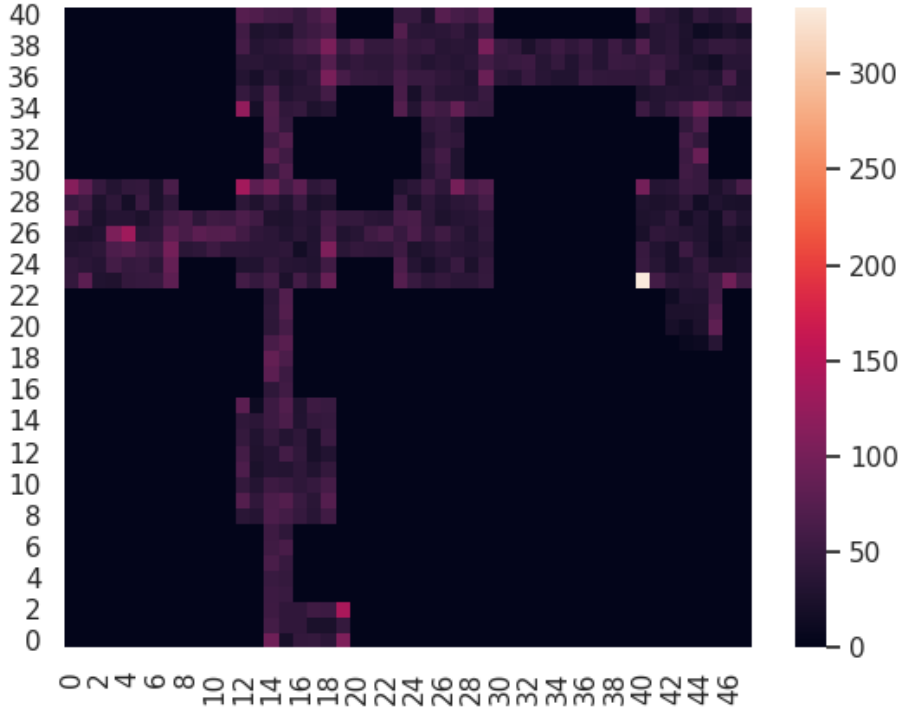
We maintained the following tables for R-max:

- $R: S \times A \to \mathbb{R}, P: S \times A \times S \to \mathbb{R}^+$ - reward and probability.
- $Q: S \times A \to \mathbb{R}, V: S \to \mathbb{R}$ - state-action value and state value.
- $Visits: S \times A \to \mathbb{N}$ - state-action visits counter.
- $Transitions: S \times A \times S \to \mathbb{N}$ - transitions counter.

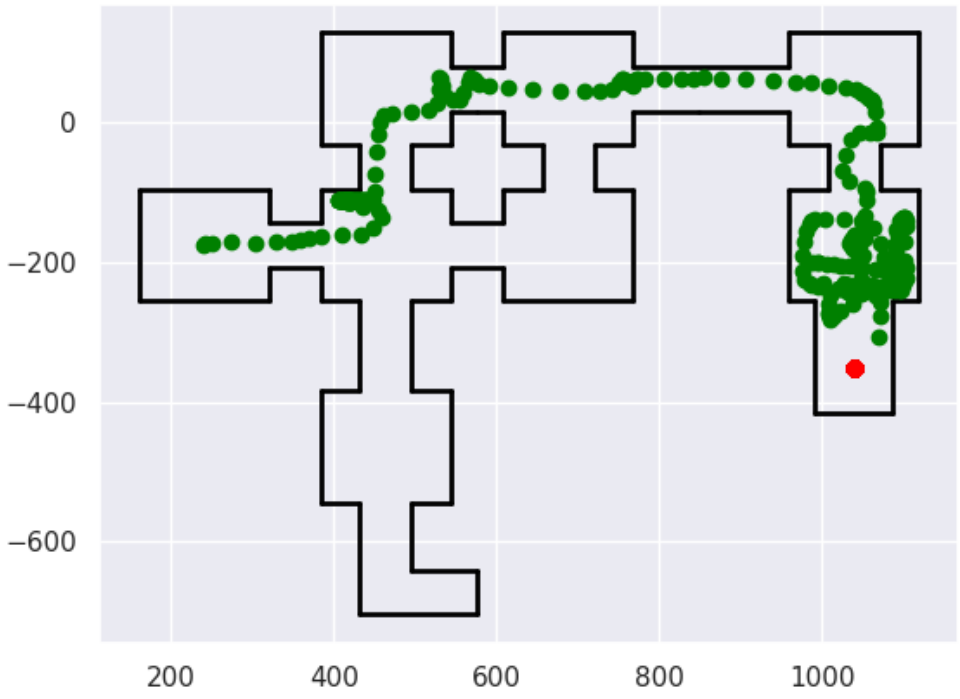
maintained the following table for Q-Learning:

- $Q: S \times A \to \mathbb{R}$ - state-action value.

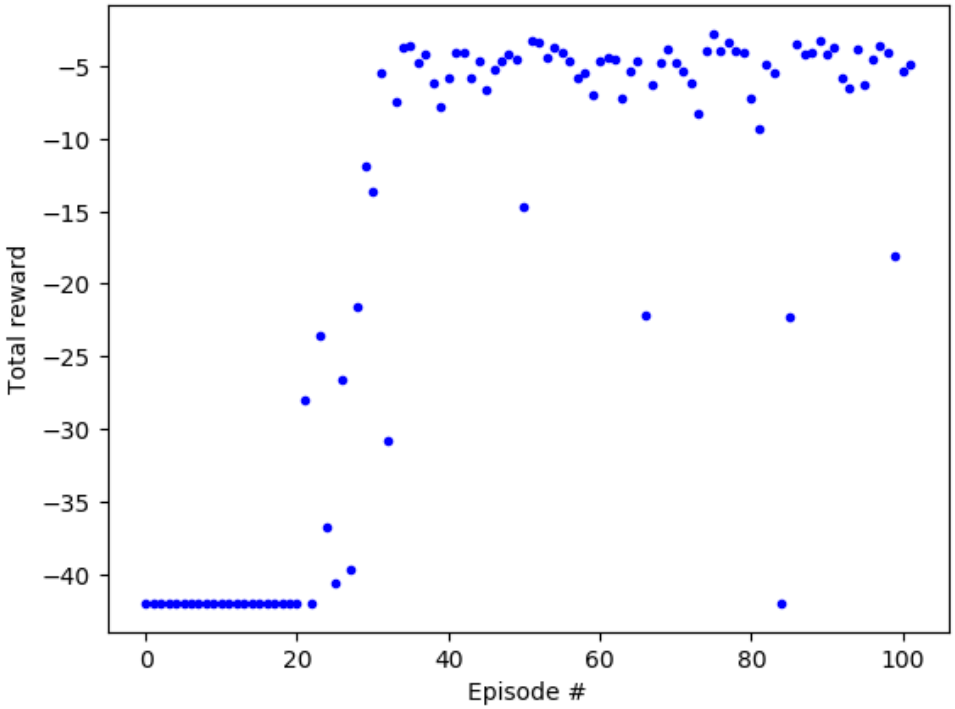
To monitor and debug the agent's behavior we produced a heat map:

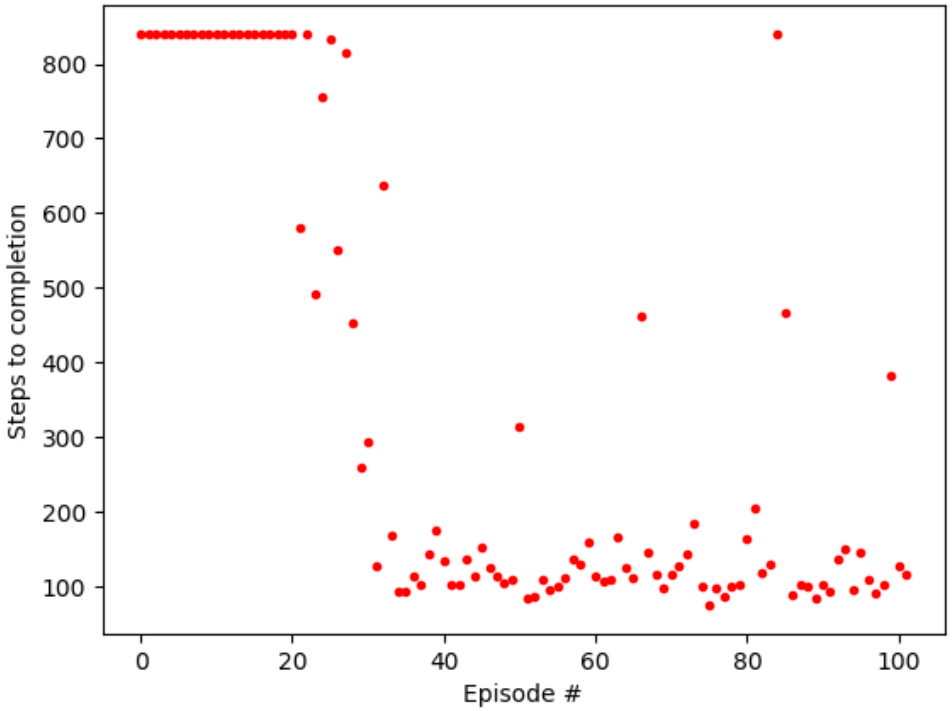


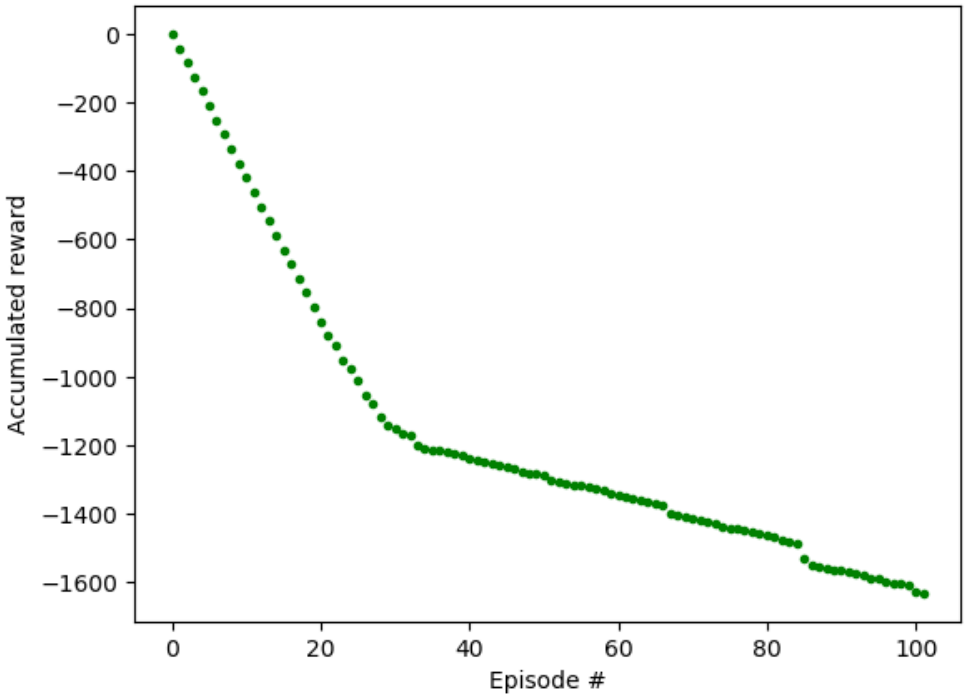
Additionally, an episodic behavior was plotted as a path:



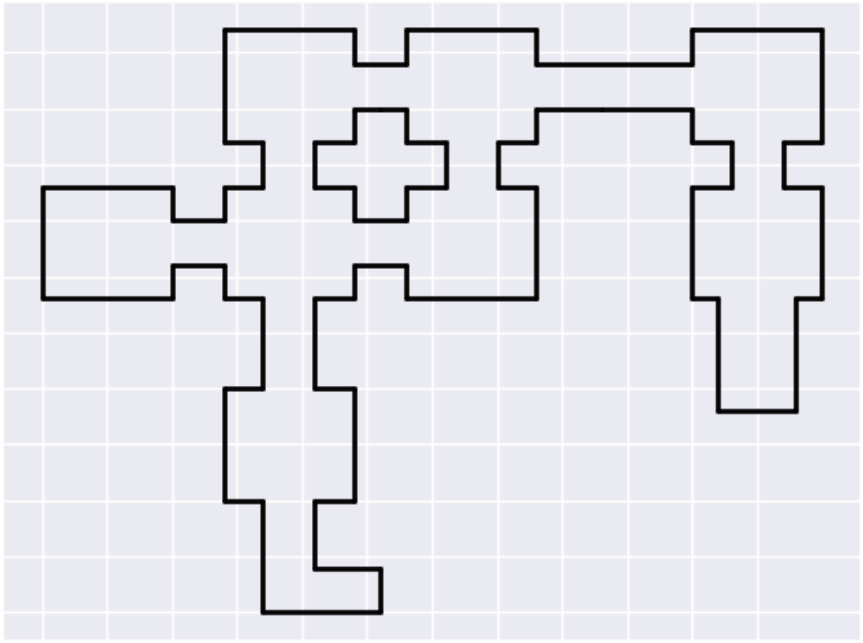
The following graphs show the convergence of this method for $\Delta = 20$:

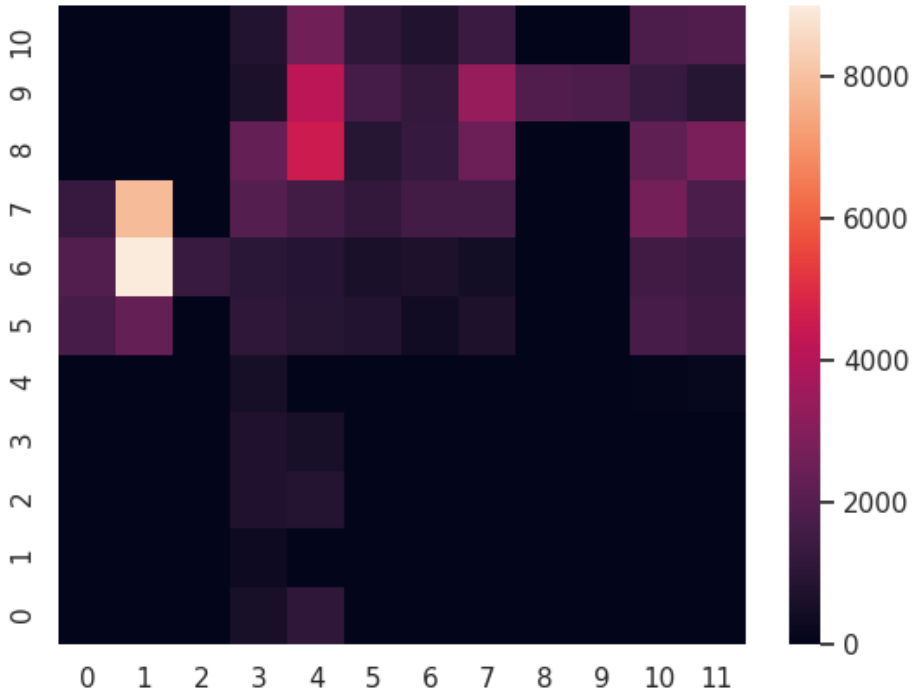


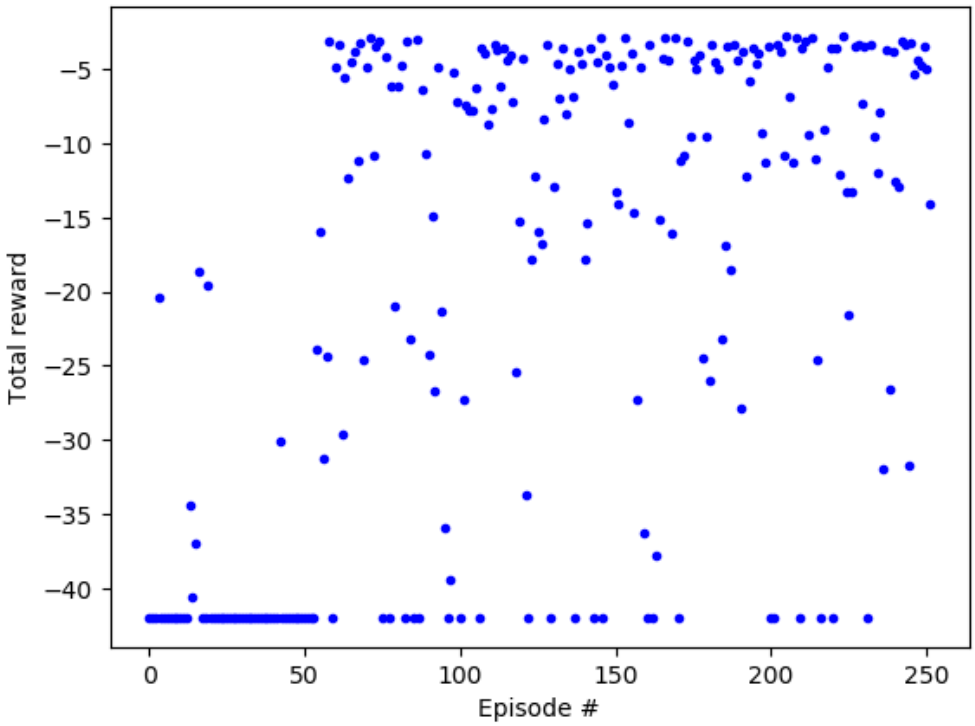


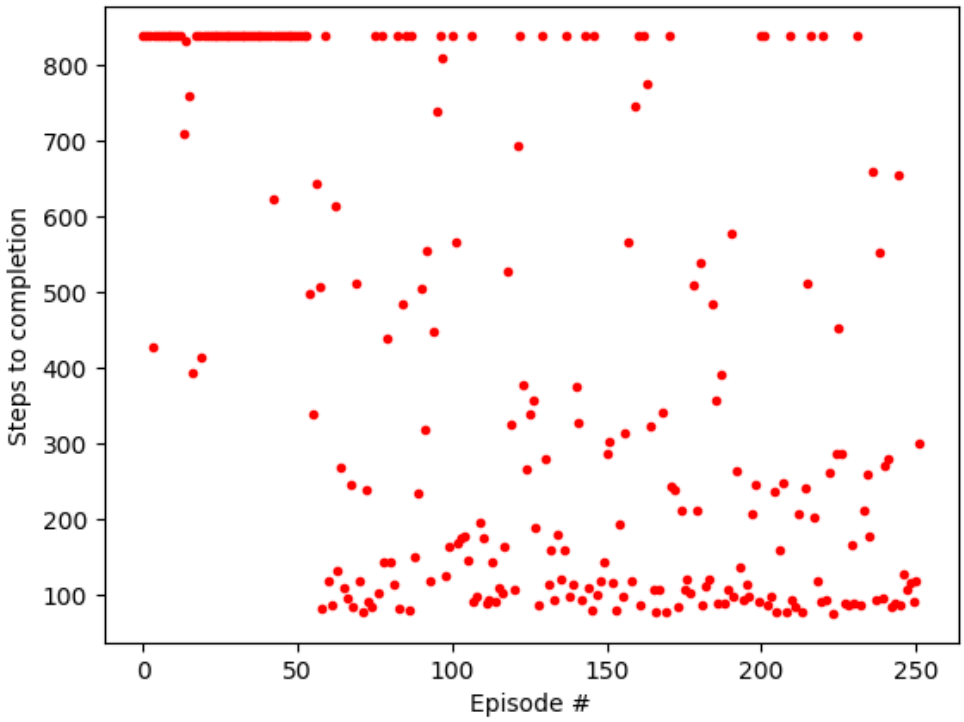


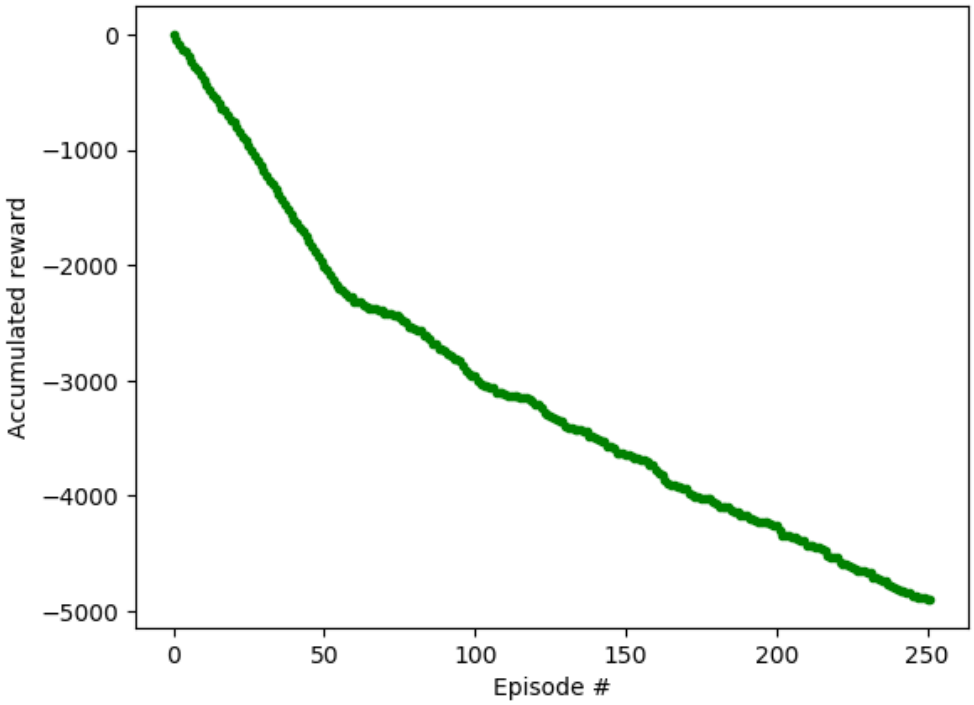
Even bigger Δ can converge, for $\Delta = 80$:




10
9
8
7
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9 10 11
8000
6000
4000
2000
0

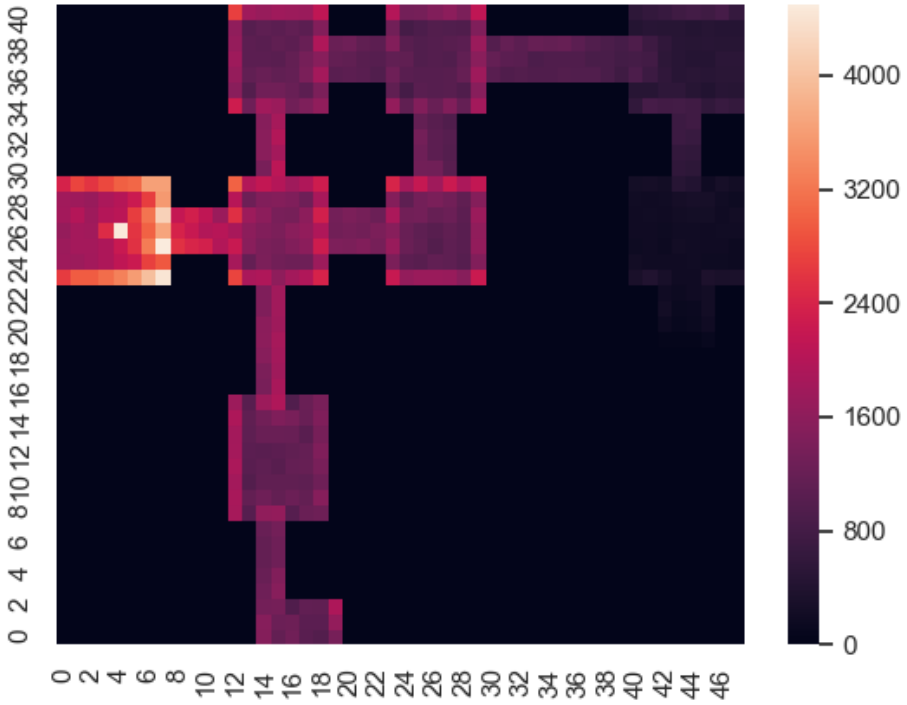

Total reward
−5
−10
−15
−20
−25
−30
−35
−40
0 50 100 150 200 250
Episode #

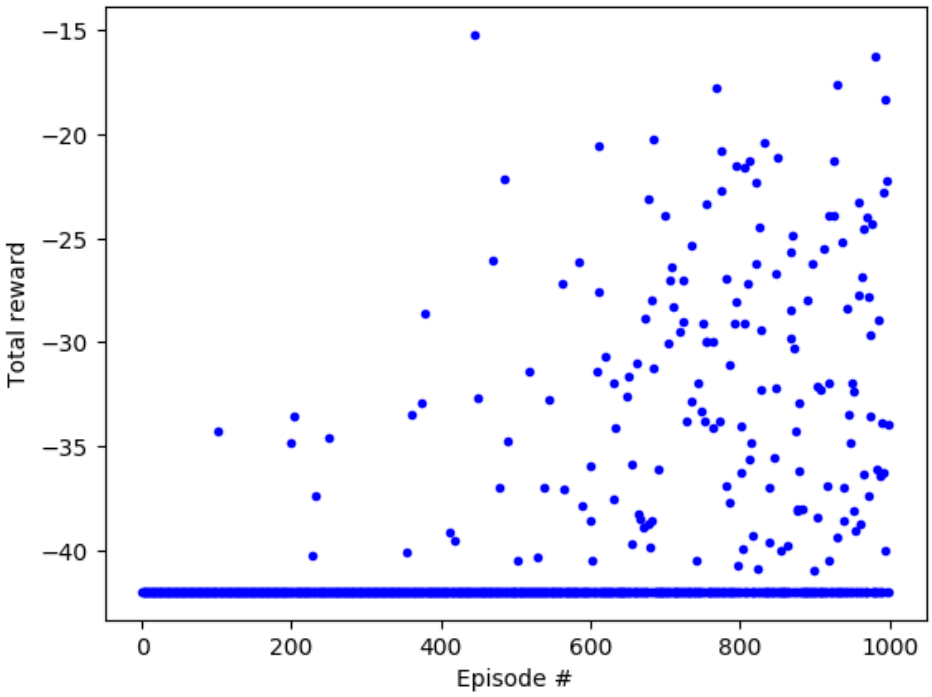


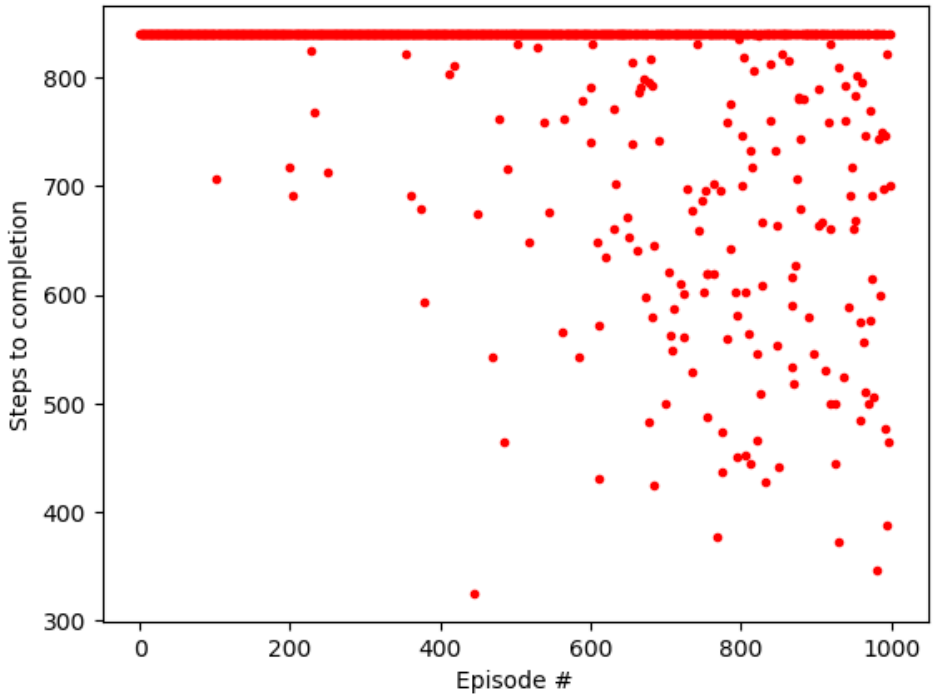


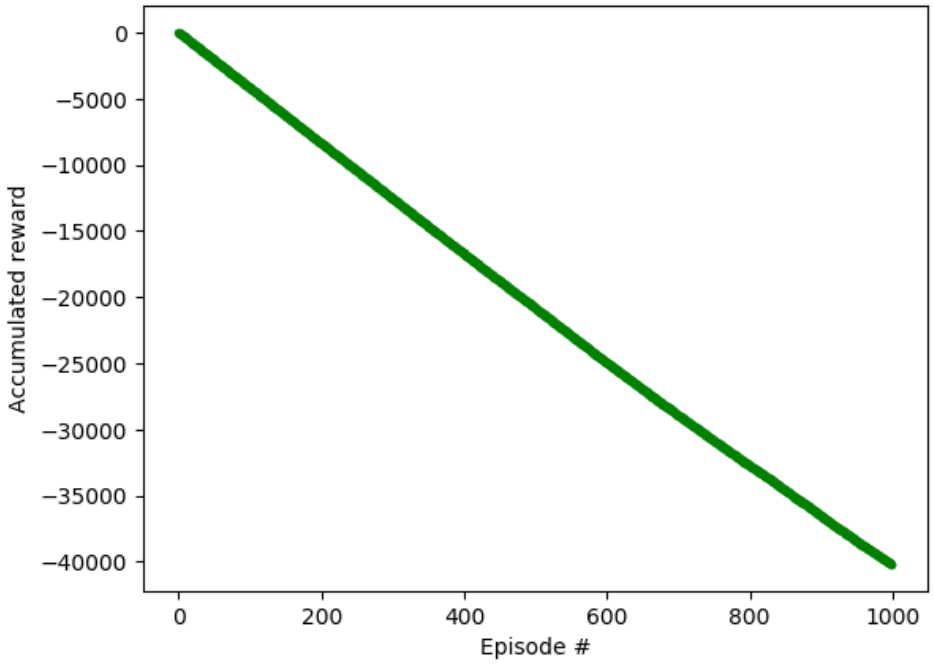
The $\Delta = 80$ trained agent has less consistent success but still converges to a solution.

To demonstrate the importance of exploration in this setting, we tried Q-Learning with $\Delta = 20$:







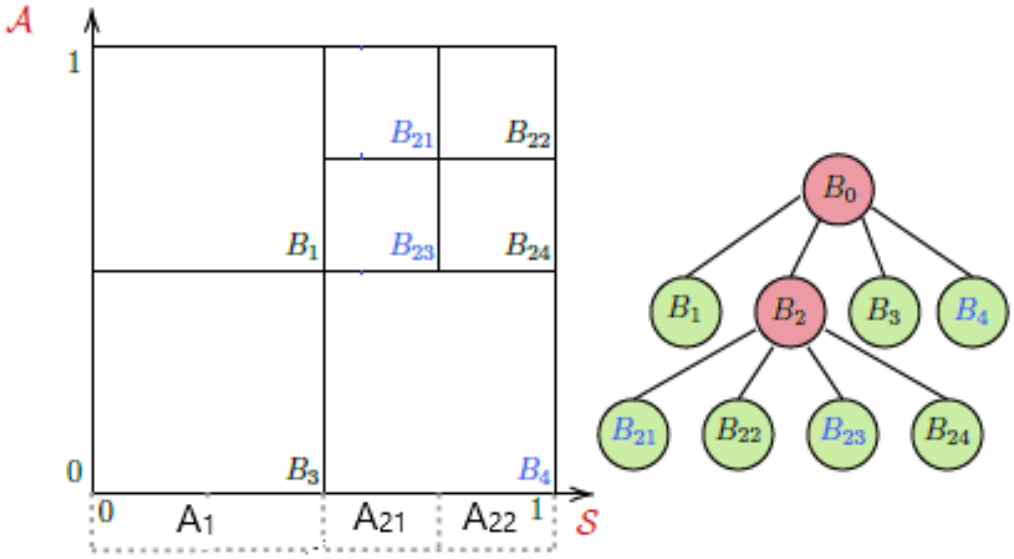


As expected, Q-Learning did not converge. The main reason for this failure is the epsilon-greedy exploration. Problems with sparse reward present a challenge for methods with poor exploration. The goal is far from the starting point and it takes time for agent to find it. Even then, the update may not propagate well enough to all the state-action Q value. So, the agent will have to hit the goal persistently for the value to propagate properly.

Adaptive Finite

The dynamic partition is implemented in a tree structure (as proposed in [1]). Each node of the tree represents a 'ball' – a segment of the $S \times A$ space. In our method a ball is a 4-dimensional box is the $(x, y, \theta) \times A$ space. We normalize this to $[0,1]^4$ space. This normalization is only possible while knowing the limits (finite) of each dimension. Each tree node can be split to smaller sub-boxes to produce a finer discretization – twice as accurate in every axis.

In the following illustrations we will use 2-dimensional boxes for simplicity. Here B_2 was split to B_{21}, B_{22}, B_{23}, B_{24}:



The initial tree structure is a head node covering all the $S \times A$ space and has 3 child nodes – one for each action. Each one of these sub-nodes covers the entire S space and only a third of the A space.

Note that the balls partition of the $S \times A$ space induces a partition of the S space. we will denote a ball by B and a segment of the S space by $\mathcal{A}$. In the above illustration there are 7 active balls (marked green on the tree) and 3 state space segments ($\mathcal{A}$).

A node can be split to smaller balls to get a finer discretization. The main goal is to split where the information needs to be more accurate and not split where the current accuracy is sufficient to solution. This idea is manifested in the splitting threshold.

The splitting always maintains the following tree invariant: A node's children together always cover the same space of the parent and every pair of children do not overlap.

Each tree node holds the following information:

- Estimate of the reward of the ball.
- Estimates of transition probabilities - a vector where each entry contains the probability to transition from the ball to a segment of the S space ($\mathcal{A}$).
- Number of visits to this ball and all its parent balls (deprecated).
- Number of visits to this ball specifically.
- Number of parent balls (number of splits along its tree path).
- Center of the 3-dimensional $\mathcal{A}$ induced on the S space by the ball.
- Radius of the ball.
- Action associated with the ball.
- Children nodes.
- Samples of the form $(x, y, \theta, action)$ the belong to the ball.
- Estimate of the Q-value.
- Q-value used for choosing action (at first it is R-max and not equals the true estimate).

Method's hyper parameters:

- Splitting threshold – determines the condition to split a tree node.
  - Our choice: $\#unique\ visits > 4^{\#splits}$ (visits based)
- Maximum splits – determines the maximum times a node can be split. Equivalently the minimal radius will be $2^{-maximum\ splits}$.
  - Our choice: 4
  - Guidelines: same as uniform discretization Δ tradeoff.
- Visits to known – a node with less visits will be treated as unknown (R-max properties).
  - Our choice: 5.
  - Guidelines: A high value in deterministic setting will waste samples. A low value even in deterministic setting can lead to misjudgment.

E.g. hitting a node that contains the target but not on the target can lead to determine its value to -0.05. The agent may not return to this location to find otherwise.

Actions performed in a split operation:

- In a N dimensional problem, the parent node will be split to 2^N children nodes.
- Each of the parent's samples will pass to the corresponding child.
- If the child is known:
    - Its reward estimate will be calculated according to samples' average.
    - Its transition probabilities will be the same as its parent's.
    - Its Q-value will be the same as its parent's.
- If the child is unknown:
    - Its reward estimate will be R-max.
    - Its transition probabilities will be zero.
    - Its Q-value will be the 2R-max.
- If the new children induce new $\mathcal{A}$s (state space segments):
    - Their V estimate will be 0.
    - All other balls transition will be updated as follows:
        i. The transition probability to the parent will be deleted.
        ii. The parent's probability will be split to the different $\mathcal{A}$s with the same distribution the samples were split to the different 'B's (its children).

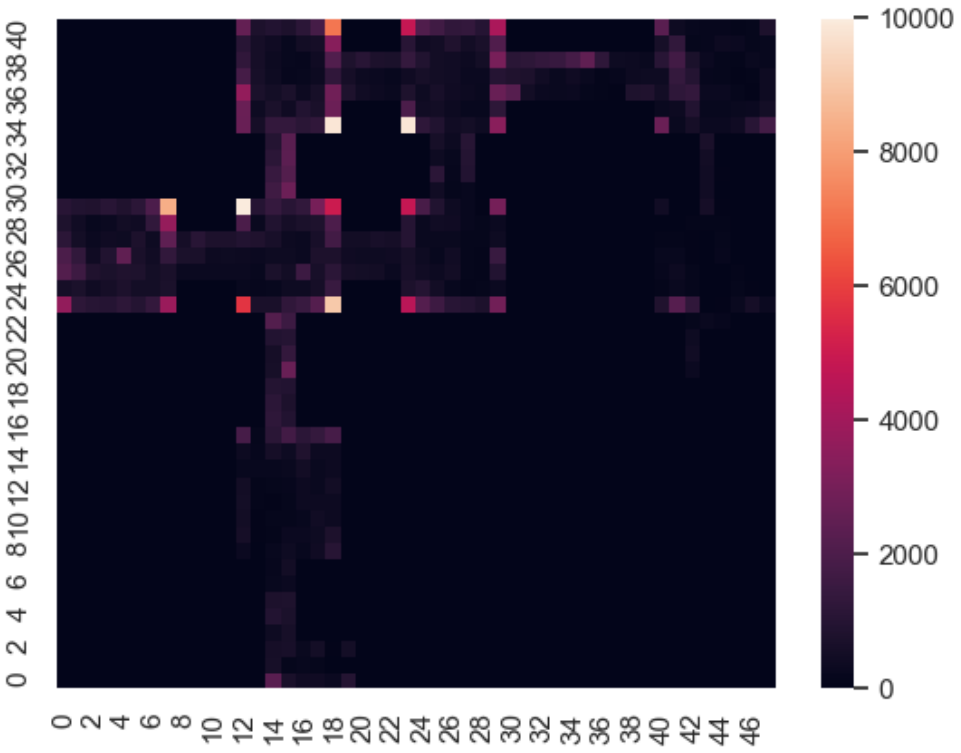
The tree holds the following information (beside the node hierarchy):

- The list of all 'A's
- The corresponding Value estimates.

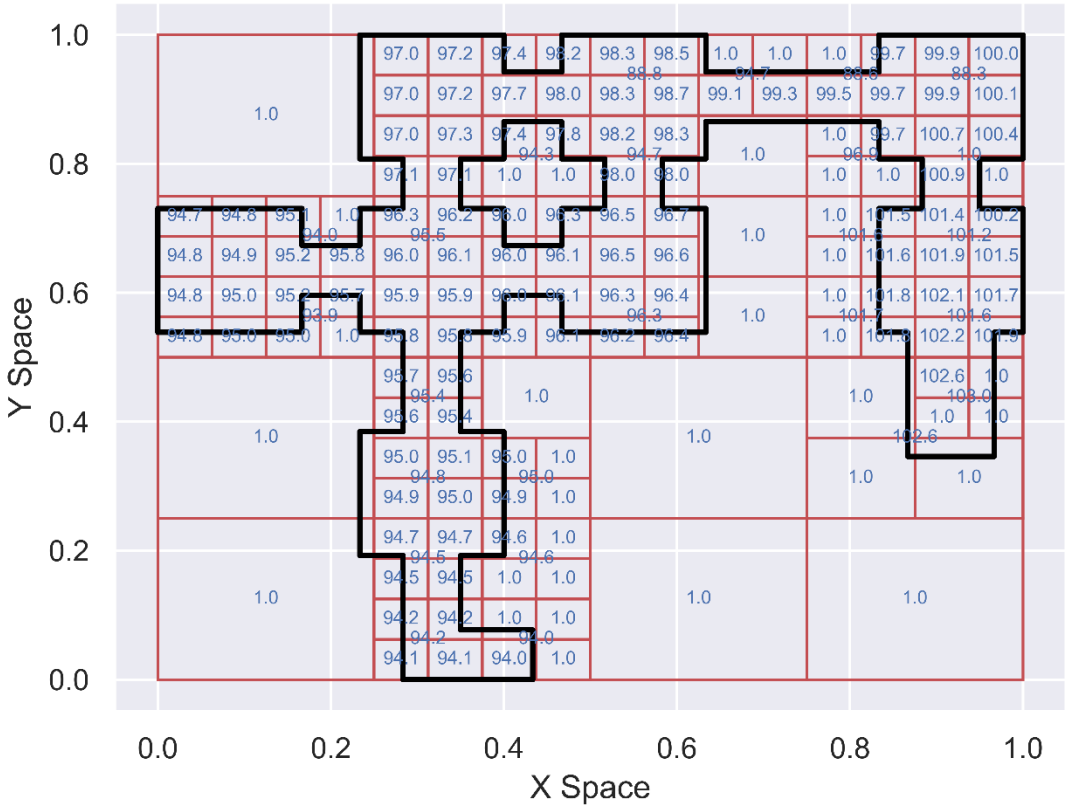
Note: each node's probability vector is with correspondence to the above lists.

To conclude, while the actor is playing and observing the tree grows and produces a finer discretization. The means by which the tree is growing are node splits.

- Results from a run with
    - Split threshold $\#unique\ visits > 4^{\#splits}$
    - Maximum splits = 4
    - Visits to known = 5

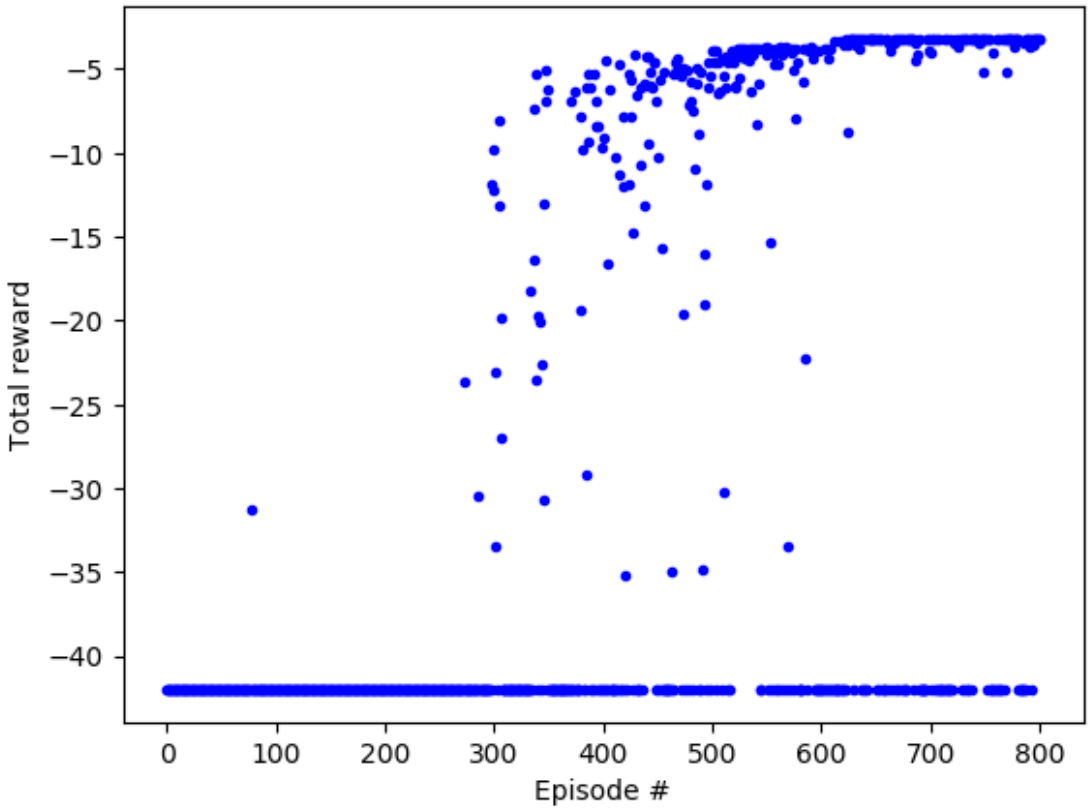


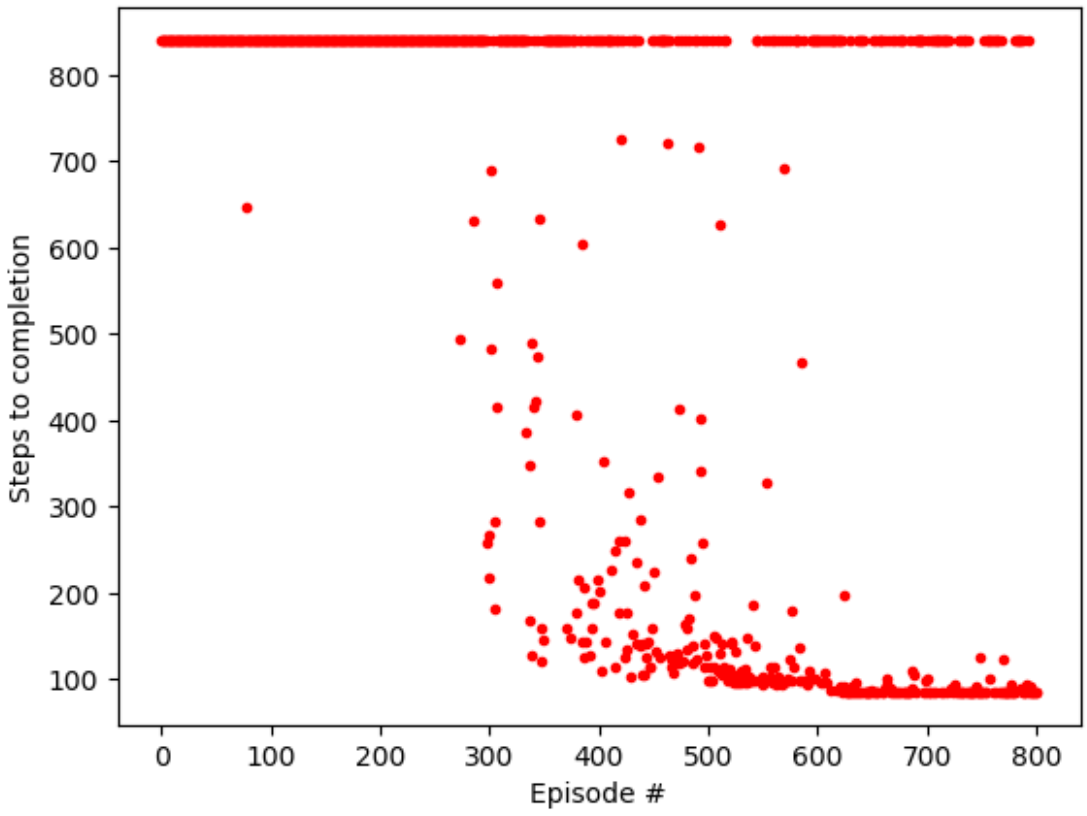
This heat map is much less uniform than the hit map of the uniform discretization run. This is mostly due to the bigger size of balls (buckets) which inserts more indeterminism to the problem.

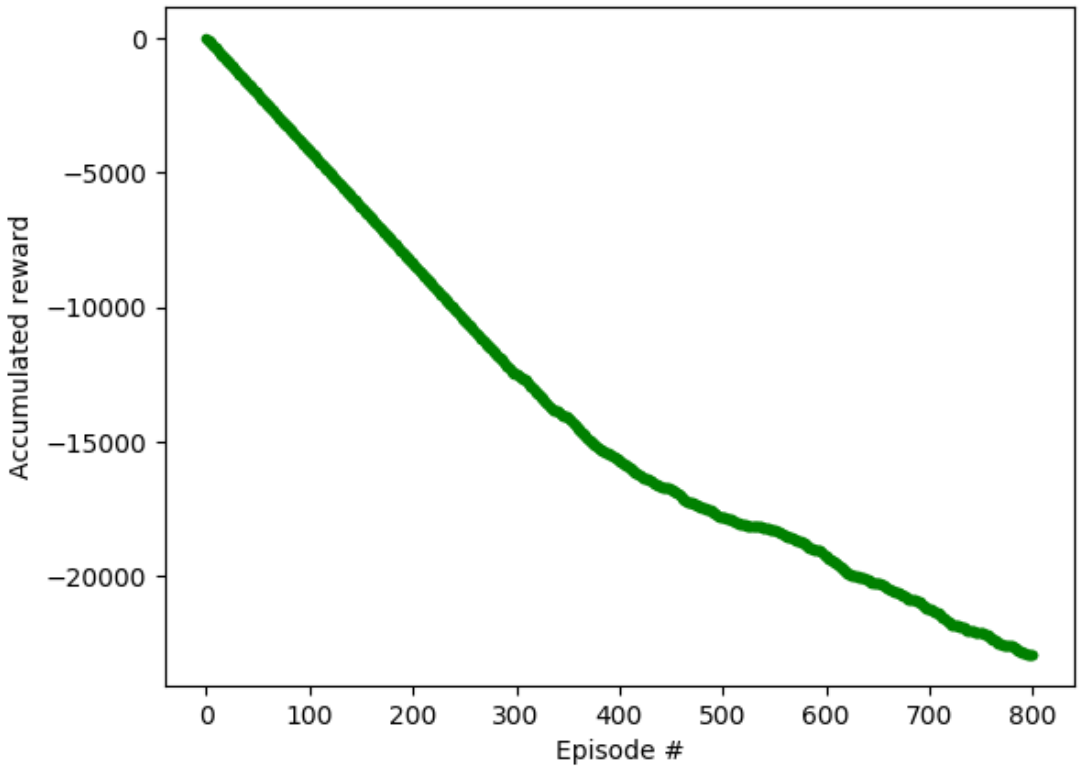


In the above graph we can see the projection of the balls on the 2D space. The numbers indicate the maximal value of the balls which correspond to the same 2D segment. All the relevant balls reached maximal splitting. This is due to the nature of splitting by number of visits – which monotonically increase. Nonetheless the big remaining balls show a benefit of this method against uniform discretization. Here we have 1,786 buckets.

We can notice the high values of the value function. Only the terminal state in our problem returns positive reward so this phenomenon requires an explanation: The target is present in a ball which contains other points as well. Due to the big difference between the rewards (1.0 and -0.05) and the frequent hitting of the target we get a reward estimate close to 1.0 for this ball. Not all points in this ball are terminal and so there is a positive probability to pass from one to another – p. Thus, by each value iteration the value is increased at least by $1.0 \cdot p$. Then all its neighbors' value increases accordingly and so on.

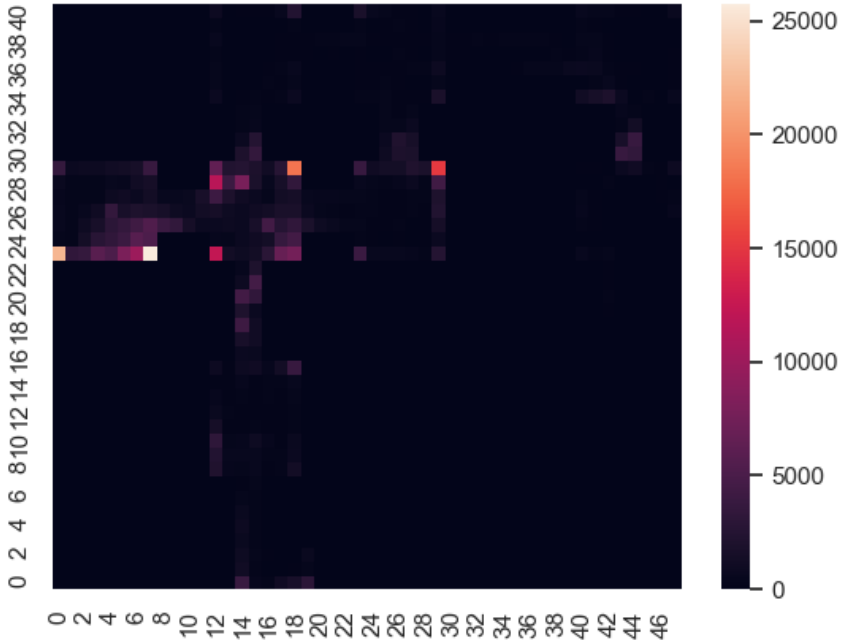

Total reward
−5
−10
−15
−20
−25
−30
−35
−40
0
100
200
300
400
500
600
700
800
Episode #

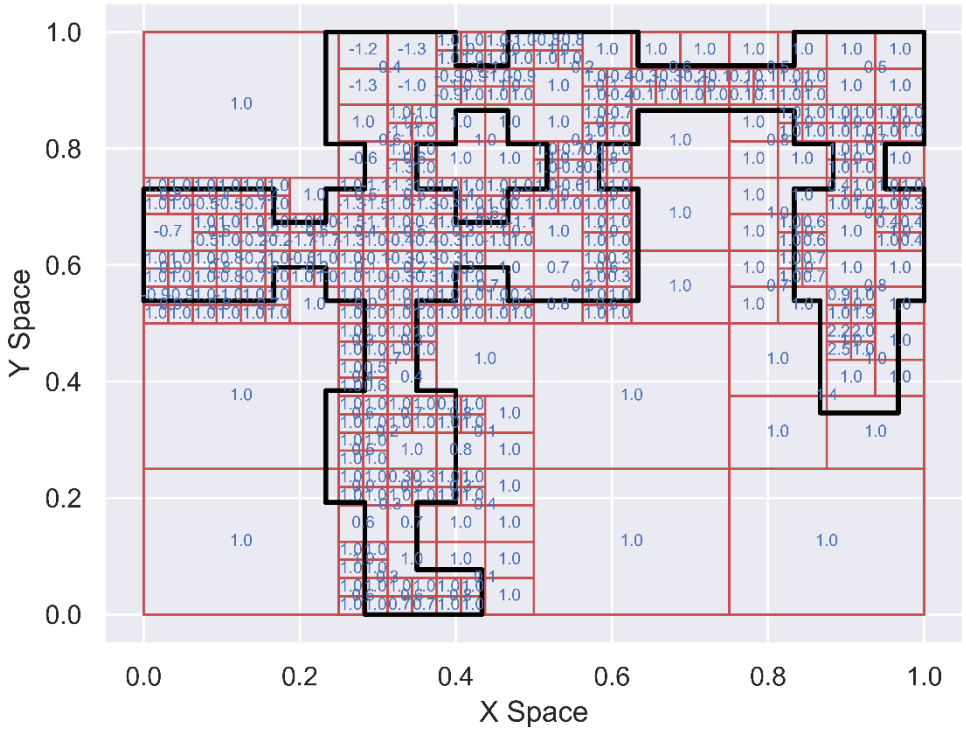

Steps to completion
800
700
600
500
400
300
200
100
0
100
200
300
400
500
600
700
800
Episode #

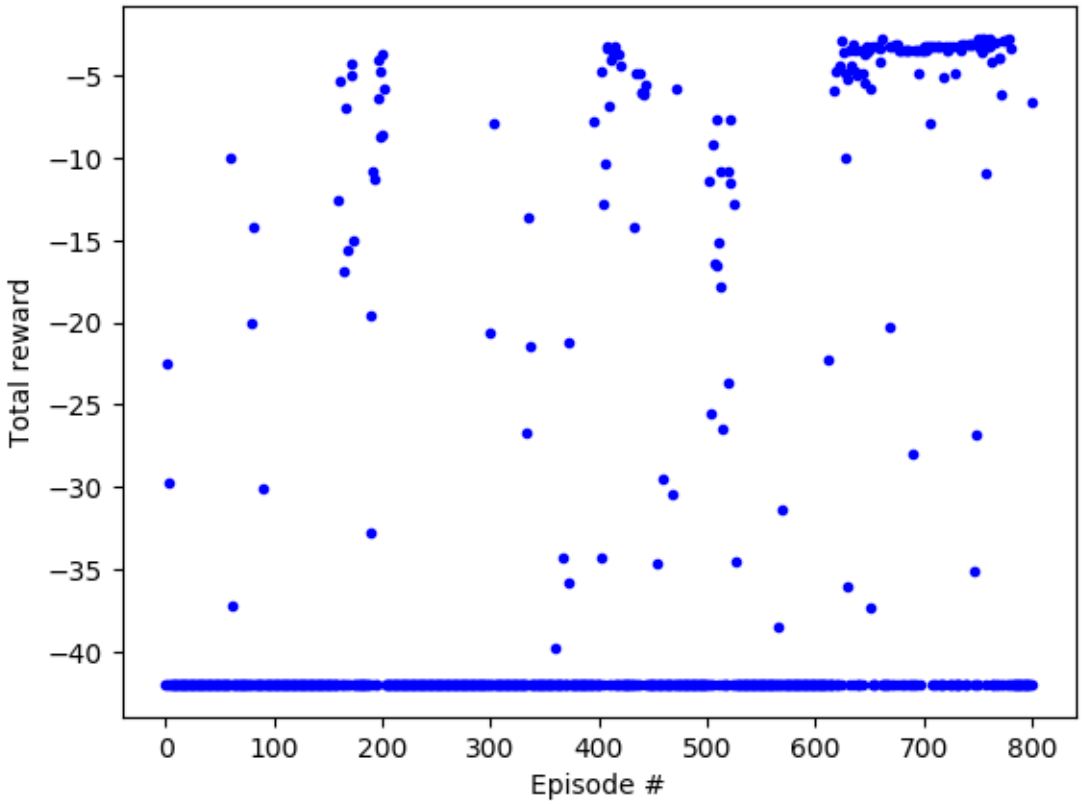


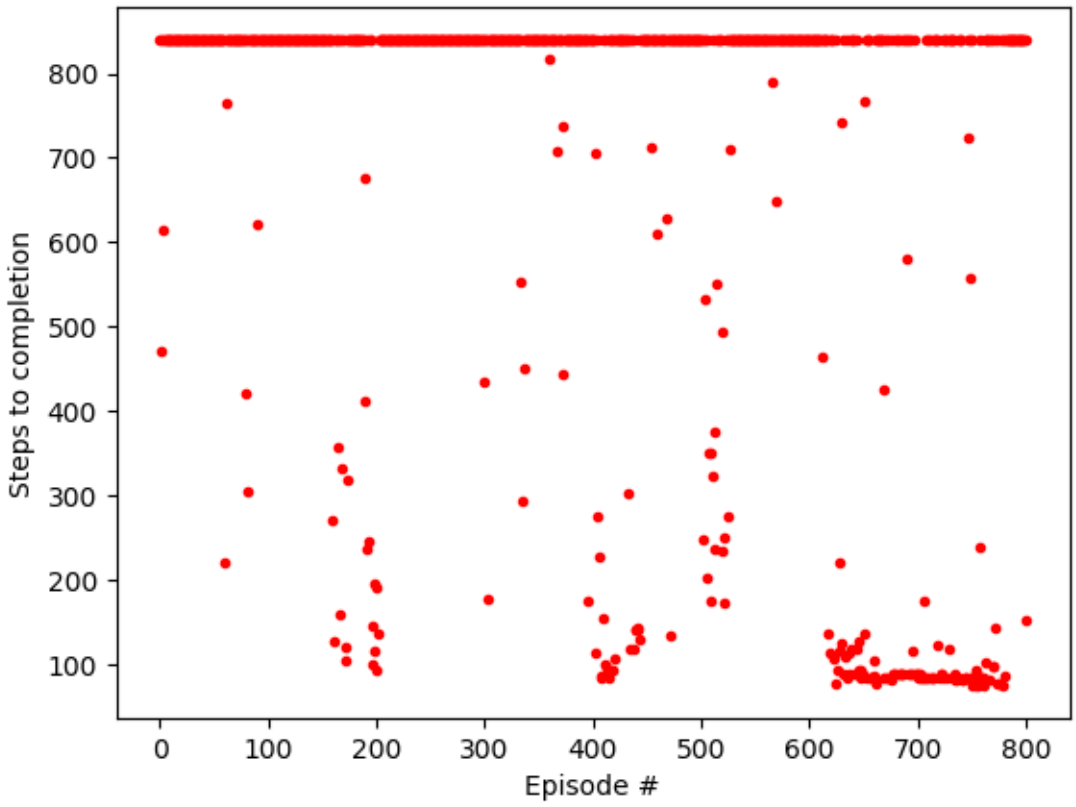
We notice that even after the agent reached what we may call convergence it still has bad episodes. We think that this behavior is also due to the big area of the balls.

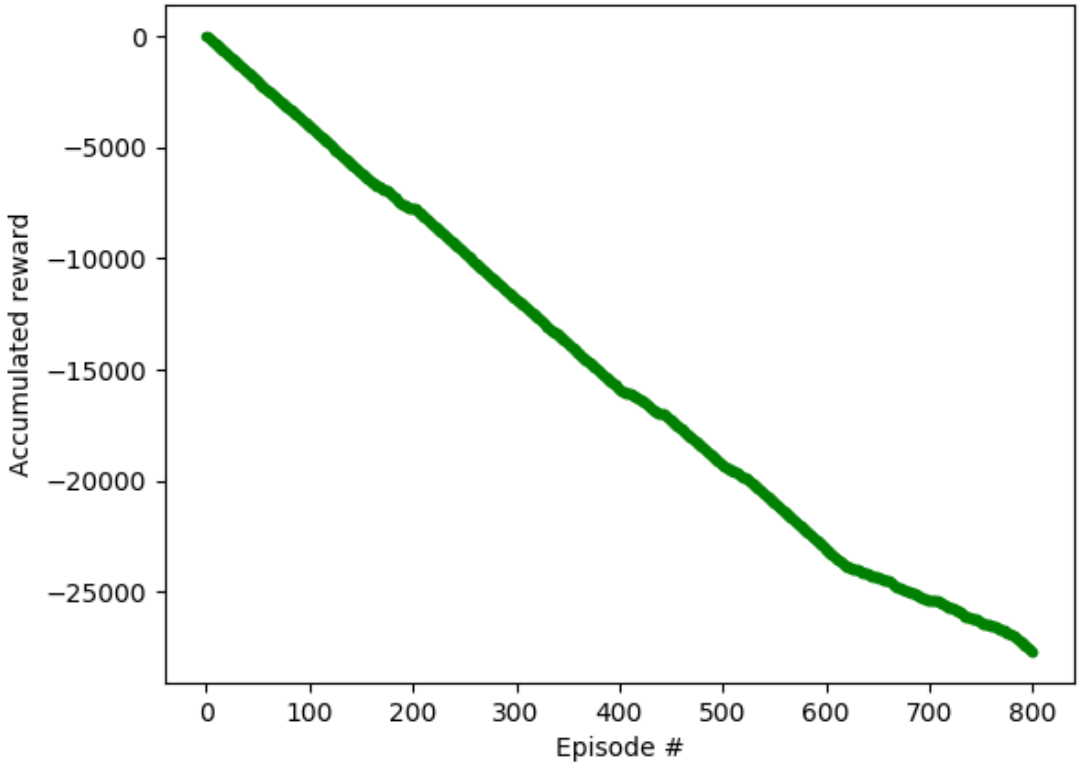
When using **Maximum splits = 5** we get over discretization which damages the convergence:






Total reward
−5
−10
−15
−20
−25
−30
−35
−40
0
100
200
300
400
500
600
700
800
Episode #


Steps to completion
800
700
600
500
400
300
200
100
0
100
200
300
400
500
600
700
800
Episode #



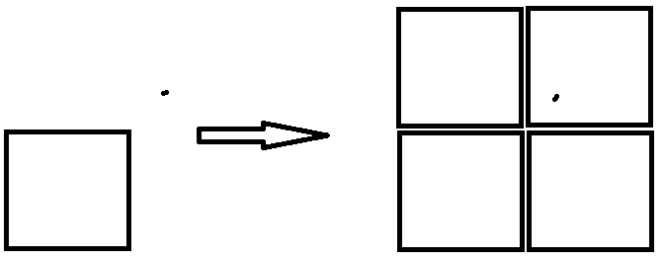
In conclusion, our method produced worse results than the uniform discretization and used a similar number of buckets. We think that this is due to the simplistic splitting threshold based on number of visits. We will suggest an improvement in the further research section.

Adaptive Infinite

We extend the previous method for the infinite setting or unknown limits setting. In this setting we cannot normalize each dimension to [0,1], thus we cannot determine an initial discretization to split.

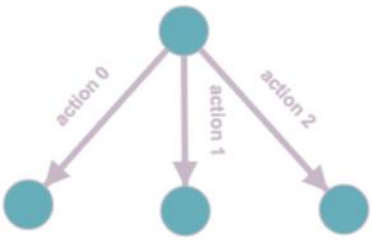
We start playing assuming arbitrary limits. Each time we observe a state outside the current borders we rescale the current tree and make it a quarter of the new space.

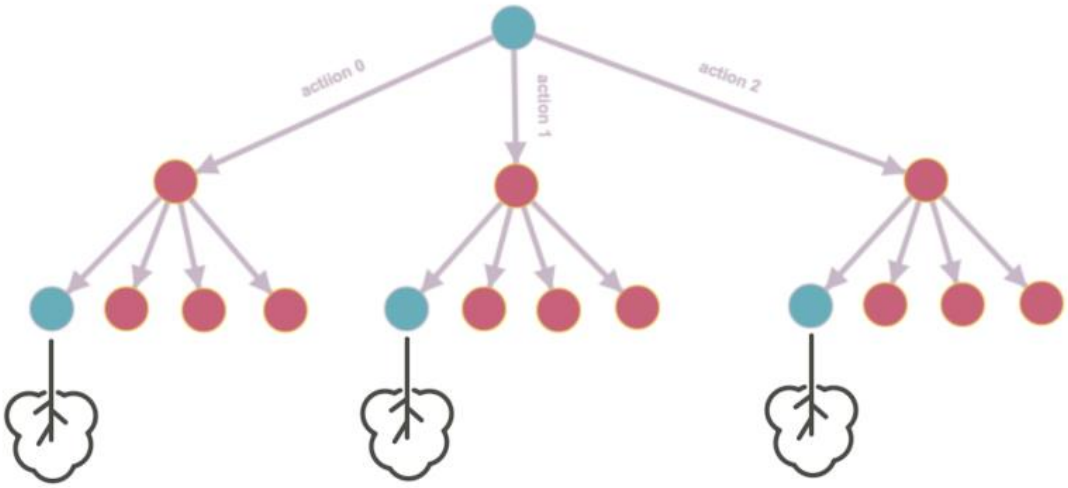
For instance:



To maintain a valid tree, we add nodes for the new quartets of the space.

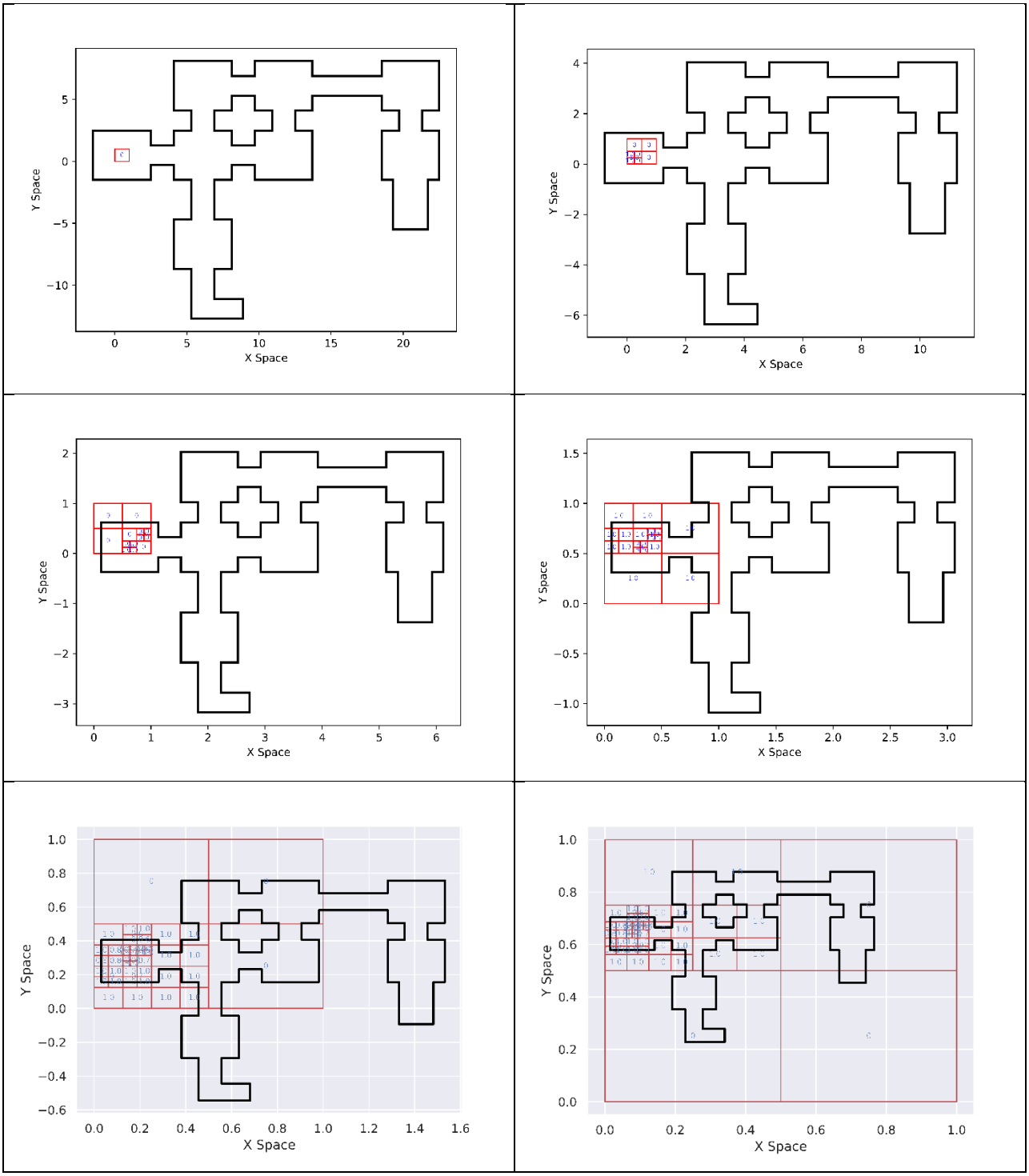
This process is illustrated below:





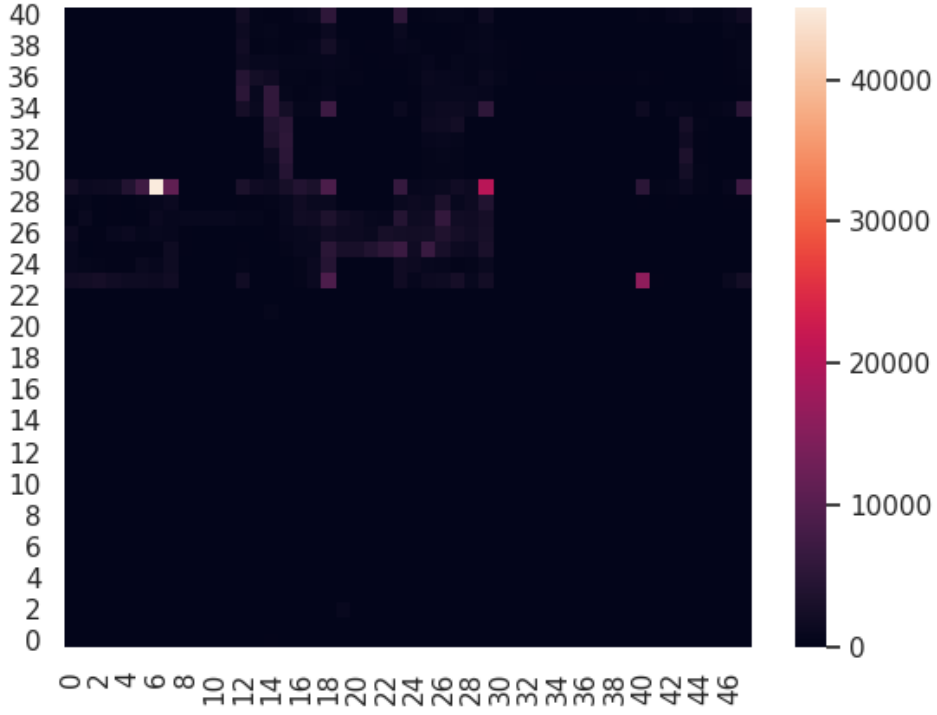
The blue nodes are the same from the original tree and the red nodes are the new "placeholder" nodes.

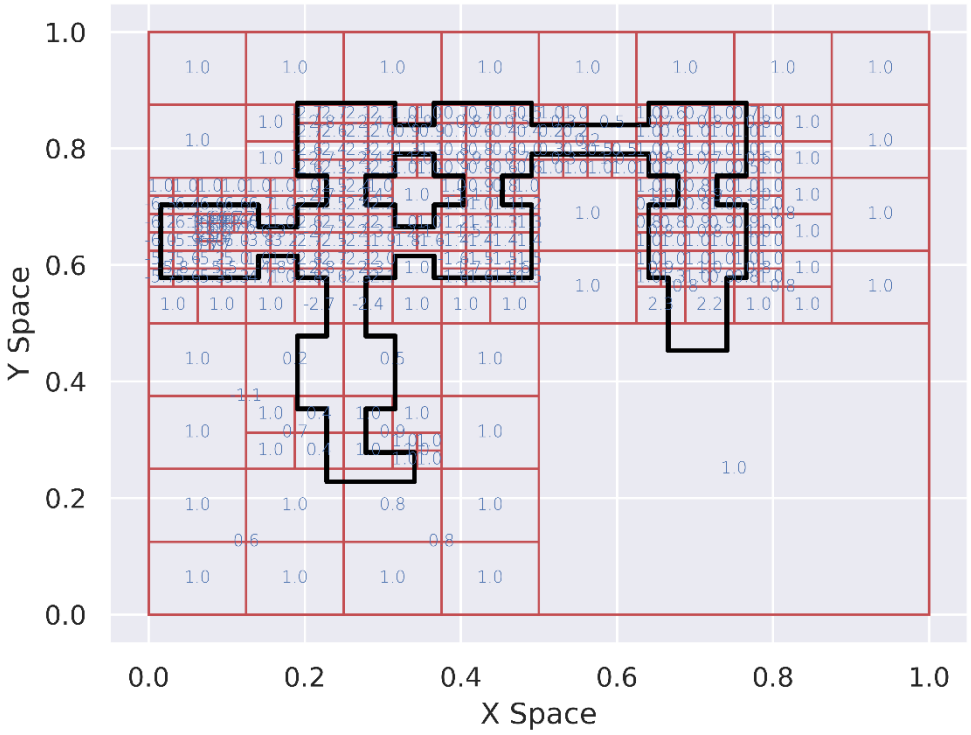
The following graphs show the rescaling process of a run:

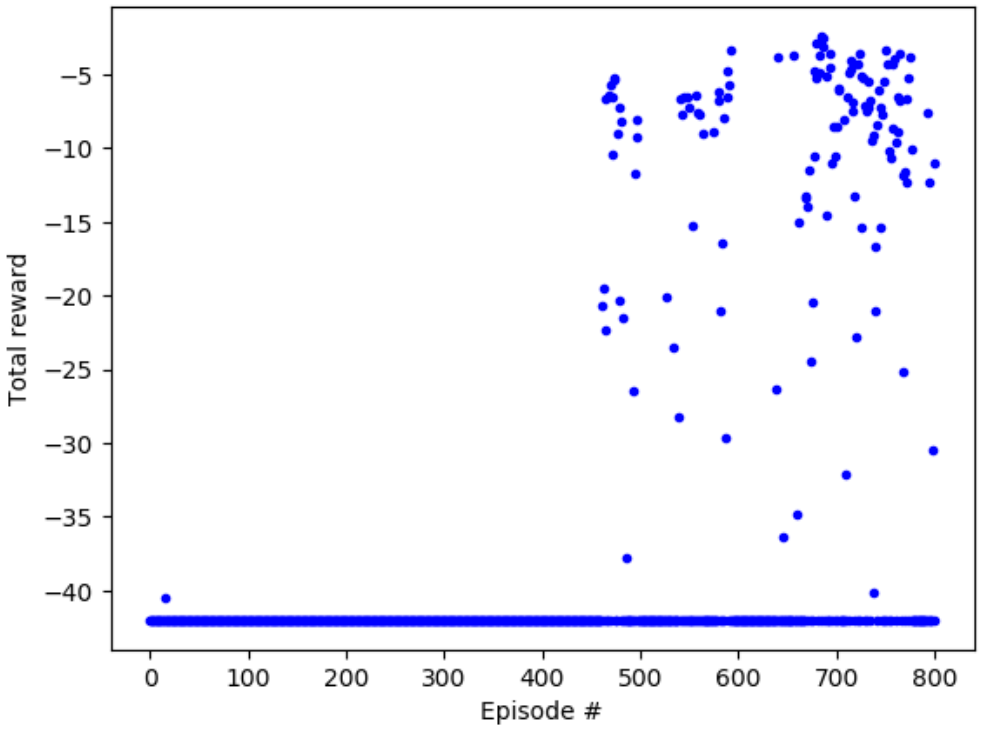


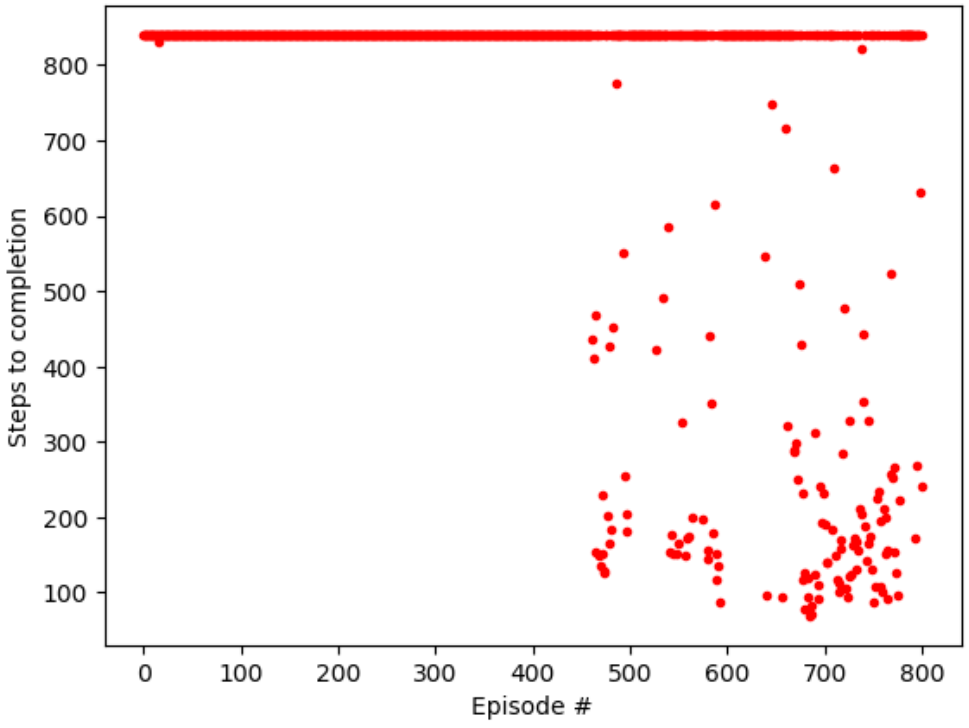
Like in the finite case we restrict the number of splits. Each rescale counts as a split for the old tree nodes, to balance the buckets' sizes.

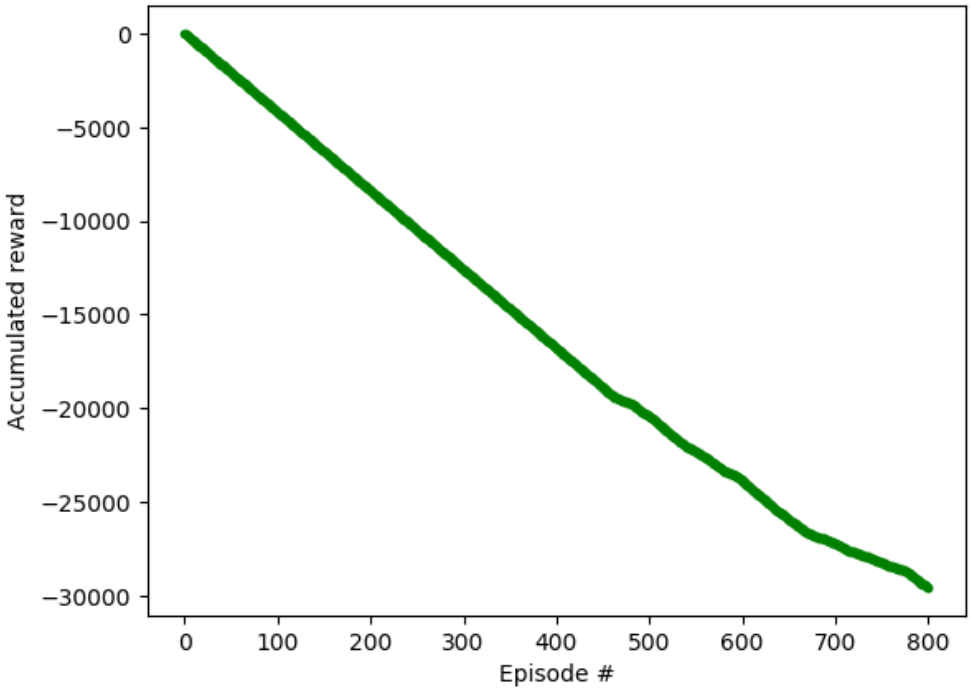
Results for this method:






Total reward
−5
−10
−15
−20
−25
−30
−35
−40
0
100
200
300
400
500
600
700
800
Episode #


Steps to completion
800
700
600
500
400
300
200
100
0
100
200
300
400
500
600
700
800
Episode #



The results are worse than in the finite case and it is hard to consider it a convergence. We suspect that this is yet again due to using the number of splits threshold to the split operation. We will propose another approach in the further research section.

Summary and Conclusions

We offered a general method for adaptive discretization for model-based reinforcement learning. Our method is quite easily adaptable to changes in thresholds and splitting threshold type. Also, changing from R-Max to another model-based algorithm is easy.

The results showed convergence on the one hand, but on the other hand performed worse than uniform discretization in reward results. The number of buckets was lower than in the uniform $\Delta = 20$ but higher than $\Delta = 80$.

We learnt a lot from the project and consider it a great first time RL hands on experience.

Further Research

We think that the results could improve greatly by changing the splitting threshold. Visits based splitting threshold may be more suitable for greedy algorithms that return more to high reward areas (although they perform poorly on our setting none the less). For exploratory algorithms such as R-Max, the number of visits increase almost uniformly on the state space in the learning stage.

A better approach will be splitting according to value discontinuities or TD-Error. This change can be applied to our method easily and to our code.

The discretization tree structure as proposed in [1] is suitable for continuous action space. Giving the agent the ability to rotate at any angle may produce interesting results.

Sources

[1] Sean R. Sinclair, Tianyu Wang, Gauri Jain, Siddhartha Banerjee, Christina Lee Yu. Adaptive Discretization for Model Based Reinforcement Learning. Available: https://arxiv.org/abs/2007.00717.