# Summary

All reinforced learning testing was done using q-learning. The grid searcher is a very traditional reinforced learning application and followed the q-learning model almost exactly. The Hamming Revision searcher is quite a different problem. The searcher still follows a reward system, but it’s hard to say if it’s still q-learning or not.

# Reinforced Learning (Grid Searcher)

The idea behind the grid searcher is to create an agent that can find a path through a maze to a goal state. Based on the reward system the agent should also be incentivized to fin the most optimal path to the goal, the one that maximizes score.

Teaching an agent requires a system of rewards. The agent will do what is in its best interest given the reward structure. Looping through states endlessly may be a valid option given a suboptimal reward structure.

Each state in the state space is given a score, which is used to create a q-table. The q-table will be used by agents to determine which action to take given its current location on the grid. This reward system incentivizes following ‘check’ states to find the ‘goal’ state. Note, once a ‘check’ state has been explored it can not be reached again. This is an anti-looping solution.

Text

Description automatically generated

The q-table is used by agents to determine which action to take given all adjacent options. The basic idea is this, what state is to the right of me? What is the score? What state is to the left of me? What is the score, etc.

A picture containing text, electronics, black

Description automatically generated

Below is a basic state space that was used in testing and tuning.

Text

Description automatically generated

Q-table scores are updated via the Bellman equation. There are a couple of constants that I tweaked throughout testing. Those constants are **a, y**. **a** corresponds to the learning rate. **y** is a constant that affects the ‘future’ score that is given to a given q-table score. To explain, the maxQ portion of the equation looks at all future state options, finds the max score or best option and returns that q-value. This part of the equation rewards the current q-value based on good future q-values. So, the **y** constant is a way tweak how you want to prioritize current states or future states in the q-table values.

A picture containing text

Description automatically generated

The equation and its constants are very important. The values that you decide on for constants and state scores will directly determine how the agent will behave when traversing the state space.

# Reinforced Learning (Hamming Revision Searcher)

The challenge with the next searcher is to learn an agent that can do some sort of belief revision. I chose to work with Hamming revision. The challenge I find with this comes when deciding what kind of a reward system to model agent behavior towards. To me it makes sense to reward an agent based on a specific state choice at a specific time, given that a belief state is a ranking over states. This becomes challenging when reward values will have to change from revision to revision. This means that whatever reward system I come up with for an example, another example might need a completely different set of rewards. This makes it impossible to teach a generic hamming revision agent given the way that this system is set up.

The following images are some examples of how structures might look when being used by the reinforced learning algorithm.

State scoring:

**Text

Description automatically generated**

Q-table:

A picture containing text, electronics, keyboard

Description automatically generated

The state is no longer in the form of a grid. I modified it because I didn’t want to allow the agent to be able to revisit a state that it has already picked for the resulting belief state. A state is linked with all other states, but once a state has been chosen it cannot be revisited.

**Text

Description automatically generated**

The biggest difference that was made when applying q-learning to this problem was modifying the q-value assignment to something that made sense to this specific problem. The most factor when attempting to place a state in a ranking is making sure that the state is in the correct place in the ranking. The exact placement might not matter. In hamming ranking 1 or more states might be on the same ‘level’ in terms of ranking. The order of states within that ranking is irrelevant.

The q-table has changed to reflect the differences in the scoring. In Hamming ranking we look at differences between characters in states. The maximum hamming distance for a state of 3 variables is 3, and the minimum is 0. Therefore, there are 4 options, or levels, for each state in the q-table. For each state there will be scores associated with possible placement locations given how the agent chooses to place states.

Eg:

Belief state: [00,01,10,11]

Sentence: 11

Result: [11,10,01,00]

11 should be placed at rank 1

10,01 should be placed at rank 2

1. should be placed at rank 3

A picture containing text, electronics, keyboard

Description automatically generated

Given this example and the ranks we know where the states should be placed but it is now up to the q-table, after learning, to find out the true locations. After the learning stage there will be values for each of the possible values in the q-table, but the best value for each state should correspond to the rank that each state **should** be at for the result.

The scoring system for the hamming revision agent does not take future states into account when scoring. It just determines the level the state was placed on, and what level the state should be placed on when determining a score. This would be a modified Bellman equation without the maxQ component.

A picture containing text

Description automatically generated

## Code Notes:

The files of note in the repository are the **environment.py**, **generate\_scoring.py**, and **main2.py** file. The **environment.py** and **generate\_scoring.py** files are used to create structures that are used by the reinforced learning algorithm. The **main2.py** file contains the implementation of the algorithm. The main driver function behind learning is the **basicMethodTwo** function in **main2.py**.

This is the block of code that you would need to modify to play with different examples of hamming revision.

Text

Description automatically generated

To create all the supporting structures hamming revision must be done on the belief state and sentence. The results of revision are used to build the structures necessary to teach the agent how to select a belief state.

There are just a couple of variables that you would need to change to try out new examples.

Text

Description automatically generated

The integer in the **generateAllStates** and **generateModQTable** corresponds to the number of variables in the vocabulary. This number must match. Additionally, states in your ‘sentence’ are in the **sent\_arr** variable. These states must have the same number of variables as the integer passed for the functions. If the same number of variables is consistent there will be no problems.