Applying Reinforcement Learning to the Shortest Path Problem

Dillon Wilson

Abstract

The Shortest Path Problem is extremely well known and has a variety of applications in the real world. Although an efficient and robust algorithm, A*, exists to solve this problem it is still the subject of plenty of published papers attempting to apply a variety of new methods to solve it. In this paper, a handful of various Reinforcement Learning algorithms will be applied to a large, weighted graph in an attempt to evaluate their relative performance. Metrics like cumulative reward and the rate in which each model finds the correct shortest path will be considered, as well as the amount of time needed to train these models.

Introduction

The shortest path problem is one has existed for quite some time, and has had a variety of algorithms applied to it over it's lifetime. Many likely are aware of the A* algorithm, which is one of the most famous algorithms to solve the problem due to its ability to handle negative edge weights, unlike Dijkstra's algorithm, and it also considers the distance already travelled in the path. Although this problem is technically solved, it is still an area of constant research due to its many applications. Examples of the shortest path problem in the real world range anywhere from Navigation Apps like Google Maps, network optimization, transportation and logistic optimization to name a few. So if a faster version of A*, or an alternative algorithm altogether, can solve the problem more efficiently all of those applications can make use of the upgraded algorithm.

Background

A cornerstone of this project is the usage of a very new type of model in the field of Reinforcement Learning known as the Decision Transformer. This model is a modification of the same transformer model that has been made famous due to it's application in the field of Natural Language Processing. It can be adapted with minimal changes for the goal of training a Reinforcement Learning agent. As with language transformers, Decision Transformers treat their input data sequentially meaning the agent will consider previous actions based on an attention mechanism presented in research that first introduced the concept (Vaswani et al. 2023). At each time step, a tuple of reward, action, and state is fed into an embedding layer that creates a representation of that data. Next, a positional encoder marks at which time step that tuple was processed by the model. The embedding is then fed into the transformer model, producing an output that is taken into a decoding layer. At the end of this process, an action is predicted that should have the highest likelihood of the best possible future reward. An illustration of this is provided at the end of this section in Figure 1.

In order to to truly gauge the performance of this advanced model, a plethora of very well known algorithms were implemented along with it. Chen et al. (2021) compare their decision transformer to multiple Q-learning variations in the well known Atari and Gym environments, however they neglected to replicate these experiments for the shortest path problem. In this paper, those experiments will be performed and extended to far larger graphs. Each of the implemented models will be discussed in further detail in their own subsection in the Methodology section.

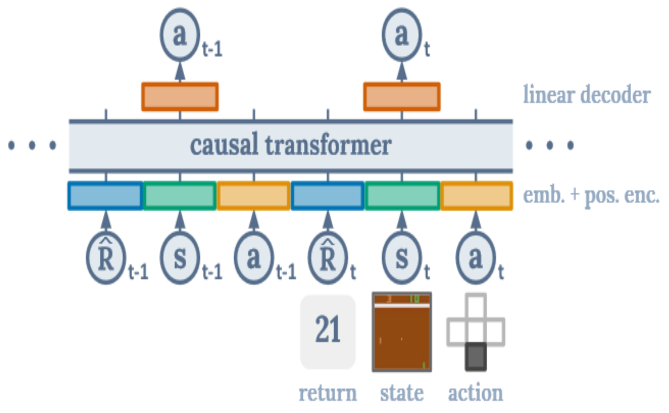

Figure 1: Decision Transformer (Chen et al. 2021)

Related Work

A crucial project that will be referenced constantly while working on this project is the research that introduced the concept of the Decision Transformer by adapting a language model to the problem of Reinforcement Learning (Chen et al. 2021). Luckily enough, there is even an experiment contained in that paper that is directly related to finding the shortest path in a graph. Due to the nature of the problem being essentially a graph where the edges are interstates and the vertices are distribution centers, research from 2023 around the application of Decision Transformers for graphs may prove useful. It is applied to a different type of problem, primarily Atari games, but will likely include useful details that will be applicable to this project (Hu et al. 2023). Another article that may prove very useful when attempting to formulate the problem correctly is known as the Meal Delivery Problem. It has it's differences, but retains the main elements of a source, a carrier that moves an item between locations, and a destination where the item needs to arrive in reasonable times. This article details an approach to solve this problem using deep Q-learning networks (Jahanshahi et al. 2022). Finally, in research done proposing delivery techniques that do not require human contact, an encoder-decoder based architecture is applied to the problem of Reinforcement Learning (Wu et al. 2023). This research may prove useful as a second source of implementation details, albeit using a different model architecture.

Environment

For this project, a plethora of python packages have been found and utilized in order to create a robust and easy-to-use platform for performing experiments. At the core, there are three crucial packages; OpenAI's Gym, NetworkX, and RouteGym.

OpenAI Gym is extremely well known and likely needs no introduction for those familiar with Reinforcement Learning research. It is and end-to-end solution to handle the training process for an Reinforcement Learning agent. It allows the user to create an environment for their problem, simulate actions, and receive the results of those actions like the agent's new state, the reward received, and whether or not the new state is terminal. On top of handling these various functions of the Reinforcement Learning process, gym is one of the most commonly used packages in the field, and as a result a large portion of the existing code and research relies on it.

The second package is NetworkX, this library is used to create custom, weighted graphs. For this project, that means graphs can be implemented that parallel the US Interstate System with custom weights equal to the distance between various nodes. There are also a handful of useful functions for creating visualizations of the graph objects, but they tend to become unmanageable with a large number of nodes.

The final package is RouteGym, this is a third-party library created for Gym that allows the user bring their own NetworkX graph and create a custom environment. This environment is capable of recognizing which nodes are connected to each other, their respective weights, and can even visualize an agent progressing through the graph if desired. The combination of these three packages facilitate the creation of an environment that is a capable testing ground for all of the various models that will be tested for this project. Along with the major three, a handful of packages are used as-needed including PyTorch, Transformers, Pandas, etc. There is one limitation to this environment when used on the largest size graph that made this approach unfeasible. This will be addressed in the Challenges section.

Using these packages, three graphs were created that would be used to test the various Reinforcement Learning algorithms presented in this paper. The standard graph consists of 48 nodes and 78 edges, with edge weights equal to the real world distances between those nodes using the US Interstate system. There is also a smaller subset graph that is used to test prototype algorithm designs. This allows for faster tests due to the smaller size, and a more explainable result due to the far smaller size. And finally, a larger graph created for additional challenge was created. This graph consists of 80 nodes and 289 edges. This is an exceptionally large graph and poses a substantial challenge for many RL algorithms, some of these problems will be addressed in later sections.

Action-space For this project, the action-space for the various agents consists of the set of nodes in the graph itself. This means that in the event that an agent chooses action 0, it is attempting to move from its current state to node 0 in the graph. However an additional consideration must be made when selecting actions, that being that none of the graphs used in this project are fully connected. So an agent cannot be allowed to transition between two nodes unless there is an actual edge in the graph that connects those nodes. For the simpler Q-learning algorithms tested, that is a trivial hurdle, simply disallow the agent from selected an action that is not valid based on the current state. This limits action selection to only valid choices, and results in a Q-table that is populated for the valid actions in each state. But for the more advanced algorithms like Deep Q-Learning and Decision Transformers, a more sophisticated approach must be taken. This problem will be explored in depth in the implementation sections for the Deep Q-Learning and Decision Transformer models.

State-space The state-space, much like the action-space, at its simplest is the set of nodes in the graph, where each element of that set has a corresponding state. The state an agent is in can simply be represented as a single value which is sufficient for the simple models tested in this project. However for the deep learning approaches, the state is represented as a vector where -1 indicates invalid moves, 0 for the current position of the agent, and 1 to indicate valid actions. As an experiment to be conducted in the future, the state could be expanded to include additional elements like state adjacency in order to allow the agents to make more informed decisions as the cost of a larger state-space and likely longer training time required.

Reward For this stage of the project, the reward structure has been set up in a very simple way to meet the desired outcome of the agent. Simply put, the reward an agent receives for taking an action is equivalent to the opposite of that edge weight. In order to create equitable results between the various models that can be easily compared, the rewards were standardized between all models if possible. The reward for a particular action was calculated using the following equation

$$terminal : r(a, s) = -(weight_{s,a}/100) + 100 \quad (1)$$

$$non-terminal : r(a, s) = -(weight_{s,a}/100) \quad (2)$$

where weight refers to the edge weight between the nodes indexed by s and a. As an example, if the agent chooses an action that takes it from node TX to node MS and that edge has a weight of 402, the agent will receive a reward of -4.02 for that action.

This implementation was chosen because the desired behavior of the agent is not to minimize the number of actions required to traverse the shortest path, but rather the real distance. Although this design does work quite well for the Q-learning algorithms, it possibly could be a culprit to the instability exhibited by the Deep Q and Decision Transformer models as will be seen in the Result section. Due to this, it will potentially be necessary to tailor a reward structure to the particular model it is being used for, but that will be explored at a later stage of this project.

Implementation

This section will detail the specific details around how the various agents were implemented and altered to fit the dynamics of this particular problems, it is split into subsections for the various categories.

Q-Learning and Double Q-Learning As one of the most widely known and understood Reinforcement Learning algorithms, Q-Learning serves as a sort of control in this project. It is well known where the algorithms weaknesses lie, however it is very effective and unsurprisingly produced great results during this phase of the project. At its core, the Q-Learning algorithm relies on the following equation in order to inform its policy.

$$Q(s, a) = (1-\alpha) * Q(s, a) + \alpha * (R + \gamma MaxQ(s', a)) \quad (3)$$

where R is the immediate reward the agent receives after taking action a in state s, alpha or learning rate is a parameter between zero and one, and gamma or discount factor is also a parameter between zero and one. Alpha determines the scale in which the existing q value should be modified by the second half of the equation, known as the temporal difference target. Gamma is a parameter that discounts the estimated value of the future state. The only difference between Q-learning and Double Q-Learning is that while Q-Learning uses one estimator in the form of a Q-table, Double Q-Learning uses two. This solves the major issue present with the Q-Learning algorithm, that being the habit of the policy to overestimate Q values. Double Q-Learning uses a slightly altered update equation to accommodate this change.

$$Q_a(s, a) = Q_a(s, a) + \alpha * (R + \gamma MaxQ_b(s', a) - Q_a(s, a)) \quad (4)$$

when an update occurs, the target Q-table is selected randomly. The equation shows the update specifically for the Q(a) table. The only modification to these algorithms is to accommodate the restrictions that must be placed on the agent as described in the Action-space section.

Deep Q-Learning Deep Q-Learning brings a modification to the standard approach of Q-learning with respect to how the Q-values for a particular action-state pair are determined. Where a table of values is used in Q-learning to keep track of these values, Deep Q-Learning uses a neural network to estimate these values. The current state is inputted into the network and a set of values of the same dimension as the action-space is outputted. The maximum of these values would be the action selected by the agent, however as mentioned in the previous section, there is a restriction placed on the action-space. So, the valid action with the highest value as determined by the neural network is selected. The nature in which the network updates its weight is done through a process known as "experience replay". As the agent performs actions during an episode, those transitions are recorded into a buffer and upon reaching a determined size, the agent samples these transitions and updates the weights of its network. However, if only one network was used by the agent it would be attempting to estimate Q-values that are being adjusted every time the network updates. For this reason, two networks are utilized. One network updates after each step assuming the buffer is the appropriate length, and the other network, known as a Target Network, is updated occasionally using the current values in the first network.

$$Q\text{-}Target : R + \gamma max_a Q(S', a) \quad (5)$$

$$Q\text{-}Loss : R + \gamma max_a Q(s', a) - Q(s, a) \quad (6)$$

where Q-Target is used to predict the action the agent should choose and Q-Loss is the value used to perform back-propagation and update the network.

Decision Transformer The implementation of the Decision Transformer model in the project currently is likely the most simple out of all three. As it currently stands, the Decision Transformer operates by storing the transitions of the last twenty time-steps and encoding them into a token. This token can then be input into the Transformer and an action will be outputted. Unlike the other algorithms, the Decision Transformer is trained by using a dataset of expert trajectories that help inform the model what decisions should be made.

The model is based on a pre-trained version of the GPT-2 model and fine-tuned on expert data gathered from the Deep Q-Learning Model. This dataset is approximately 10,000 records and consists of a state vector, action vector and reward. This data is ingested using a trainable version of this model in an identical fashion as described above with the additional steps of calculating loss based on the predicted output and then performing back-propogation.

A2C

An additional algorithm tested in the second half of this project was the Advantage Actor Critic. This algorithm makes use of two separate networks not unlike Deep Q-Learning, but they have different purposes. A2C is a synchronous modification to the A3C algorithm, originally proposed in 2016 (Mnih et al. 2016). Rather than using multiple asynchronous actors working in an environment simultaneously, A2C uses a single actor and critic. While this is not nearly as performant as the asynchronous version, it is substantially easier to implement and troubleshoot due to the absence of multi-threading. In this algorithm evident by the name there are two major components. First is the critic, this network operates in a very similar fashion to a Q-Learning algorithm. When given a state, the model will output a value estimate for that state. This value does not consider which action is being performed in the state unlike Q-Learning. The second component is the actor network. This network uses a policy gradient function, meaning it attempts to optimize it's policy directly rather than through a value function like the critic.

At each step in an episode, an advantage metric is calculated based on the of the following formulas

$$A(s,a) = r + \gamma V(s') - V(s) \tag{7}$$

where A is advantage, r is the immediate reward for taking action a in state s, V(s) and V(s') are values calculated by the critic. Upon finding this advantage, loss is calculated for both networks and back-propogation is performed. However, a method of exploration is needed in order for this algorithm to function properly. Rather that using an epsilon value to determine when the agent will explore, this algorithm uses randomly generated noise to modify the output of the actor network. Normally, the actor model outputs a vector of values that represents a probability distribution for the set of all available actions. This randomly generated noise is then added to the probability distribution, this results in some degree of randomness that will allow the agent to not always perform the most optimal action.

Evaluation

Based on the desired outcome of the agent, the metric for evaluation will be two-fold. Because training efficiency is an important aspect of this project, tracking cumulative reward by episode gives a good idea of how quickly the agent's policy is converging on the optimal solution to a problem. It also provides useful insight into how stable the agent's learning is, high oscillations between episodes likely means the agent's policy is flawed or that the agent is exhibiting too much exploratory behavior. However, if the algorithm is working correctly it is likely that a chart will show erratic behavior early in its training but it should stabilize around a reward that is equivalent to the length of the shortest path.

The second metric to be tracked for each agent is its ability to find the true shortest path in a graph. This can simply be represented as a percentage of successful trials versus total trials. The agent's solution for shortest path can easily be verified by using a built-in function in NetworkX that uses A* to calculate the path between two nodes and its length. It is expected that, when working correctly, all of the algorithms will be capable of finding the shortest path through a graph. Due to this, the true test will be how efficiently the agent learns in the window of 500 episodes, as well as its rate of finding the shortest path especially when its policy is imperfect. The Deep Q-Learning algorithm makes use of two algorithms.

Results

80 Node Graph

The following section contains the results of the three models tested on the largest graph during this project. Results from the models on smaller graphs are omitted as they were intermediate steps to reach the final goal.

Shortest Path Percentage First will be analysis of each models ability to find the shortest path from a random source to a random destination. Each model was given 30,000 training episodes per trial with parameters that were as closely aligned as possible between model architectures. This is a simple percentage of how many trials each agent found the correct shortest path as determined by the A* algorithm. In total, there were ten trials conducted for each model.

Algorithm	Shortest Path %
Double Q-Learning:	100
Deep Q-Learning:	90
A2C:	70

This table is the first in a series of data points that will show that the Double Q-Learning model achieved the best performance out of the three models tested. On top of being the most efficient in terms of training time, the tabular approach, likely because it does not need to estimate values, was able to find the optimal path between source and destination in every trial performed. Deep Q-Learning was able to achieve nearly the same performance, albeit far slower, failing on only one path. A2C performed well enough, but suffers from instability likely due to the different approach to exploration compared to the Q-Learning models. This trend will be easier to visualize in the graphs included below.

Cumulative Reward The following graphs were created by collecting cumulative reward after each episode while conducting the experiments described above. These datasets were then compressed by calculating a rolling average based on the last 25 records and then taking each record that is a multiple of 25. This effectively compresses the original 30,000 records down to 1,200 making the graphs far less noisy while still being representative of the overall trends during training. In total, three trials out of the ten performed were selected for analysis in this paper. Individual results of each model for that trial will be discussed after the chart.

Figure 2 was one of the easier of the ten, with the shortest path consisting of four steps total. All algorithms were able to successfully find the shortest path. On this graph, and the subsequent charts, it can be see that Double Q-Learning produces an extremely smooth curve with very little of the

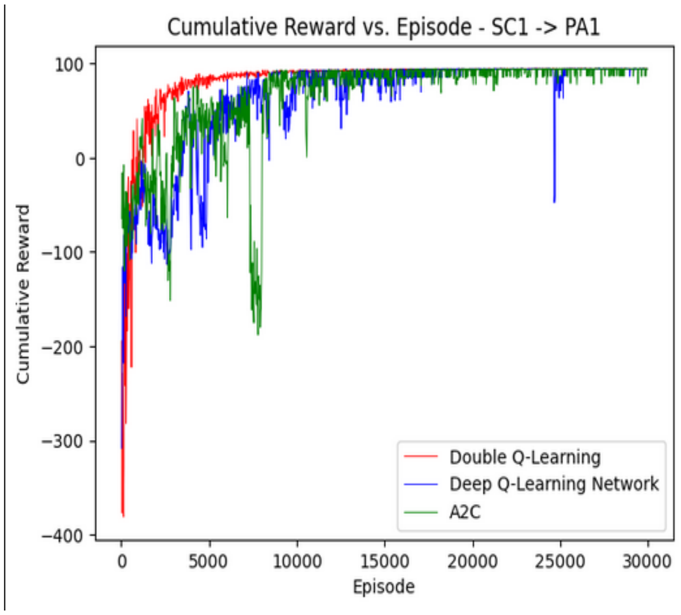

Figure 2: SC-1 to PA-1

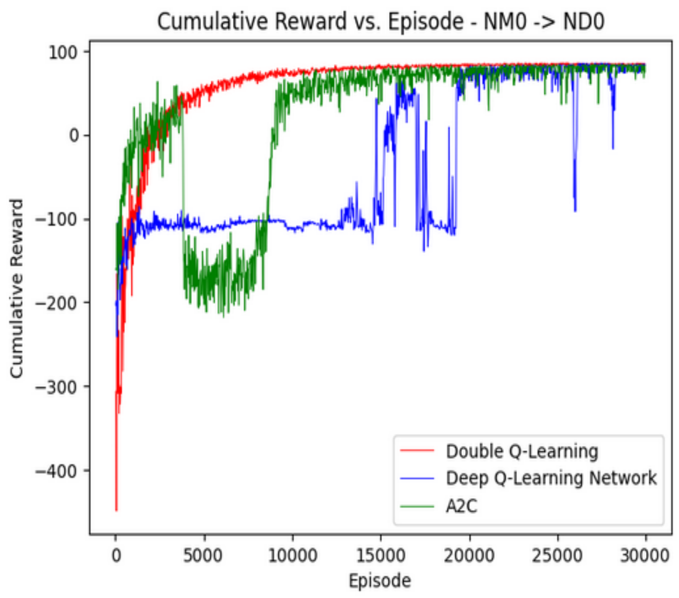

Figure 3: NM-0 to ND-0

fluctuation and instability that can be seen in the A2C and Deep Q-Learning models. Although not a precise analysis, it can be seen that nearly all three lines are overlapping near the end of the trial, implying that they have converged on the same reward value from the shortest path.

Figure 3 was a slight step-up in difficulty, consisting of six steps. A2C did not manage to find the shortest path during this trial, although it did find a path from the source to the destination. This can be seen by the green line falling slightly below and the red and blue lines in the chart. Another trend that can be observed from this chart is the tendency for the Deep Q-Learning model to sit around -100 reward for a substantial portion of the trial before jumping up to the correct reward. This is most likely due to the agent design for the model. When exploring, the agent will only consider valid actions meaning that the reward should not be exactly -100. Once the rate of exploration begins to drop and the agent's policy begins serving primarily, the agent is free to select invalid actions which would result in rewards of -100 or lower. Likely the majority of the time at and around -100 is the agent learning which actions are invalid during the episodes leading up to roughly 20,000.

Figure 4 the most interesting results out of the ten trials performed. Although this is likely the messiest graph of the three, even for Double Q-Learning, all models found the shortest path. It looks like Double Q and A2C performed nearly on par with each other with the exception of the large dips seen from the A2C model. Deep Q-Learning exhibits the same behavior as seen in the previous graph hovering around -100 for a large number of episodes. Interestingly, all three models see a dip in reward right around the very end of training at nearly the same point. It is possible that due to this path's longer length, there was just a larger amount of exploration of invalid actions needed all the way through training.

Overall, the results from this project for both the percentage of shortest path found and cumulative reward were acceptable. Although A2C performed the worst in terms of learning stability and shortest path percentage, 70% meets the threshold of working as expected. Being the last model developed, it is possible that the results for A2C could be improved with additional tweaking and improvements. Double Q-Learning showed strong performance both in terms of training time and results making it clearly stand out among the models. The more complicated models, although interesting to implement and test, do not deliver results commensurate with the amount of work required. That being said, a function approximation or policy gradient approach may make more sense if the graph was substantially larger and a tabular lookup was inefficient.

Challenges

Training Time

A common issue present across training nearly all of the models on the larger graphs is related to the amount of time needed to train the agent. The only exception to this is the Q-Learning and Double Q-Learning models likely due to the tabular approach and lack of deep neural networks. The Deep Q-Learning and A2C models had relatively decent training times, generally in the range of 10-20 minutes for 10,000 training episodes. For the largest graph this was a trade-off between required number of training episodes to fully explore the expansive graph and the amount of time before getting results to evaluate.

Unfortunately the same could not be said for the Decision Transformer. Training this model on a dataset of roughly 10,000 records of expert trajectories required roughly 30 minutes per epoch of training on the smallest graph. In some

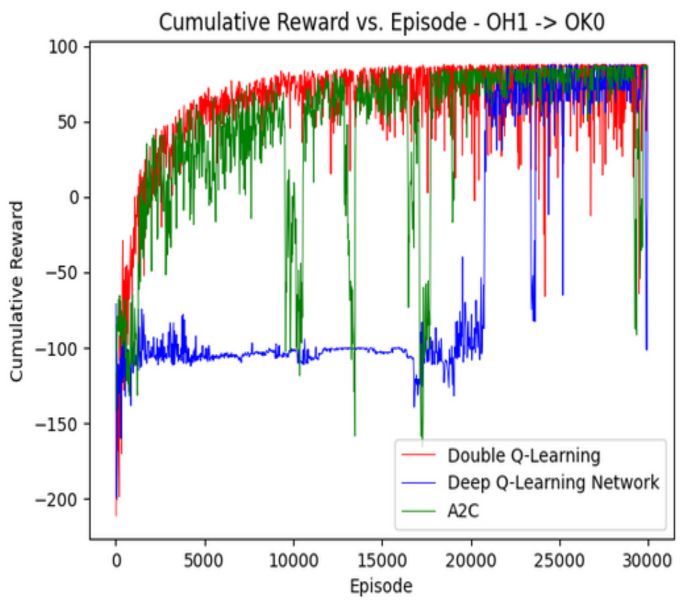

Figure 4: OH-1 to OK-0

ways, this is unsurprising, transformer models are the current state of the art and along with the impressive performance comes high computational costs. With access to more resources in the form of GPUs for training it may have been possible to get this working on the larger graphs, but due to time constraints, the Decision Transformer was excluded from the final tests. On it's own the time needed to train the Decision Transformer could possibly have been allocated, but the major concern was related to turn-around time with the model. Say it takes four hours to train the model on the 48 node graph, that means in order to truly evaluate if the current design on the model, dataset, and current configuration cannot be determined until four hours later. In order to deal with this issue, the Decision Transformer was tested on the subset graph and for this graph acceptable results were achieved, but these results did not transfer to the 48 node graph. With all this in mind, and three other models being designed and tested, the decision to remove this model was made.

Environment Complexity

As mentioned above in the Environment section, for the first two graphs tested during this project, a combination of NetworkX, RouteGym, and Gym were able to be used in conjunction to create a gym environment capable of serving as the agent's testing ground. This was extremely convenient in order to get the project up and running, but even with the 48 node graph, there were issues with how long the environment took to be created and reset. In order to work around this issue, five environments were created ahead of time and pickled so that they could be loaded from a file when needed during tests. In total, the creation of these environments took roughly 10-15 minutes each so this was a minor road bump. However when beginning tests for the XX node graph, the code to create the environment ran for hours without ever concluding. This is likely because the environment must calculate the shortest path from each node to every other node in the graph while the environment is being created. Even with an efficient algorithm like A*, this becomes intractable when considering that multiple trials must be conducted to obtain results for the various models tested.

Due to this issue, a version of each model that did not rely on a gym environment was created. Luckily, this involved minor code changes at the cost of some of the convenience that comes along with using a gym environment for Reinforcement Learning.

Conclusion

This paper concludes a research project performed around the idea of creating multiple Reinforcement Learning agents with the purpose of solving the shortest path problem on a graph that represents the United States Interstate System. In total, three algorithms were implemented; Double Q-Learning, Deep Q-Learning, and A2C. Ultimately, all models were able to achieve decent results on a graph consisting of 80 nodes and 289 edges, all with acceptable training times. From the results, the Double Q-Learning model achieved the best results and the lowest training times, making it the strongest performing model tested. Although a simple approach, the tabular method has shown that in the discrete action and state-space presented in this paper, it shows excellent results.

References

Chen, L.; Lu, K.; Rajeswaran, A.; Lee, K.; Grover, A.; Laskin, M.; Abbeel, P.; Srinivas, A.; and Mordatch, I. 2021. Decision Transformer: Reinforcement Learning via Sequence Modeling. arXiv:2106.01345.

Hu, S.; Shen, L.; Zhang, Y.; and Tao, D. 2023. Graph Decision Transformer. arXiv:2303.03747.

Jahanshahi, H.; Bozanta, A.; Cevik, M.; Kavuk, E. M.; Tosun, A.; Sonuc, S. B.; Kosucu, B.; and Başar, A. 2022. A deep reinforcement learning approach for the meal delivery problem. *Knowledge-Based Systems*, 243: 108489.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous Methods for Deep Reinforcement Learning. In Balcan, M. F.; and Weinberger, K. Q., eds., *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, 1928–1937. New York, New York, USA: PMLR.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2023. Attention Is All You Need. arXiv:1706.03762.

Wu, G.; Fan, M.; Shi, J.; and Feng, Y. 2023. Reinforcement Learning Based Truck-and-Drone Coordinated Delivery. *IEEE Transactions on Artificial Intelligence*, 4(4): 754–763.