Progress Report

Annabelle Cloutier

May 31, 2023

Contents

1 ECO-DQN, Max Cut Problem

1.1 Network Structure

All of the below information is based on the work in [1]. The network structure is incredibly generic most of the way through, but notes are written here to avoid having to parse through the equations that describe the network structure in the paper. Note that at any step where learned weights are used, the ReLU function is used on the resulting vector. The only exception is for the last set of learned weights in the readout layer, which instead is a linear output, giving the expected reward for choosing that graph vertex.

1.1.1 Node Embedding

The Message Passing Neural Network (MPNN) structure used converts each vertex into a set of m observations and encodes them into a n-dimensional embedding using learned weights. These m observations contain information about the state of the solution with respect to that vertex that they call local observations, as well as global information for the entire candidate solution being considered. The paper states that they use 64 dimensional embeddings and they do in code, but this could be any size. Their code provides parameters for changing the size of these layers through parameters to the network on initialization.

The observations in the paper as well as their justifications for them are as follows:

1. Whether the current vertex belongs to the solution set S or not;

   This is a local observation. The reason they state this observation is important is that it provides useful information for the agent to make a decision on whether the vertex should be added or removed from the solution set.

2. The immediate cut change if the current vertex state is changed;

   This is also a local observation. Just as with the above observation, they state this provides useful information for decision making. They also state is that it allows the agent to exploit having reversible actions. This makes intuitive sense, as knowing the difference between having a vertex in the solution or not can allow the agent to learn to make an informed decision on whether to add or remove it, or in this agent's case, possibly reversing an action, based on the change of the cut value if they were to change that vertex' state.

   Not mentioned in the paper is that this result is normalized by the largest non-zero change in the cut value from the empty solution set. Intuitively, this is the largest non-zero weight of a vertex, being the sum of the weights connected to the vertex. This does not guarantee that all values will be normalized between -1 and 1, but it does give a frame of reference to

know how much better a greedy action would be in the current candidate solution compared to an empty solution set.

3. The number of steps since the current vertex state was changed;

   Also a local observation. For this observation, they mention that it provides a simple history to the agent to prevent looping decisions. They also state it allows the agent to exploit reversible actions just as with observation 2. Therefore, this specific observation could be important as the value increases as the vertex remains unchanged, which can entice the agent to choose vertices that have not been chosen in a long time to encourage exploration, as well as discouraging it from getting stuck in a local minimum, changing the same vertices repeatedly over the decision making stage.

   This result is normalized by the total number of steps in an episode (which I'll define later), to range each value between 0 and 1.

4. The difference of the current cut value from the best observed;

   This and all future observations are global. This observation they state ensures the rewards are Markovian. Just as with observations 2-4, this observation they also mention allows the agent to exploit having reversible actions, which makes intuitive sense. The agent knowing the difference between the current cut value and the best one observed, along with the other observations, can allow the network to make informed decisions on whether the current solution is a promising set to explore.

   The way this is calculated is by comparing the current cut value to the best observed. Not mentioned in the paper is that they use the absolute value, but because the best observed is always chosen prior to a decision being made, this value is necessarily always either 0 (the current candidate is the best observed) or larger, as the best observed is necessarily a larger value cut for the maximum cut. This detail is important This value is also normalized by the largest non-zero weight of a vertex.

5. The distance of the current solution set from the best observed;

   This observation as with observation 5 they state ensures the rewards are Markovian. Again, this observation allows the agent to exploit reversible actions.

   The difference between this and observation 5 is that observation 5 compares the cut value only. This observation compares the solution sets and counts the number of vertices that differ between the best observed solution set of vertices and the current one. This is not explained in the paper, but can be seen in the code.

   Example for clarity:

   Best observed bitmask of vertices: $[0, 0, 1, 0, 1]$

   Current bitmask of vertices: $[0, 1, 1, 0, 1]$.

Distance = 1.

6. The number of available actions that immediately increase the cut value;

   Once again, this observation ensures Markovian rewards, they also mention that this allows exploitation of reversible actions, which is more easily understandable. Although this information can technically be inferred through observation 1, that information may not be transmitted throughout the entire graph in a finite number of steps in the message passing stage of the neural network.

7. The number of steps remaining in the episode

   The only reasoning they have for this observation is that it accounts for the finite time used for solution exploration. This information can allow the network to make different decisions based on how deep in it is exploration it is. The exact behaviour isn't necessarily predicted, but their findings do show that the further in an episode the network reaches, the more likely it is to revisit a vertex.

1.1.2 Edge Embedding

The edges for each vertex are also encoded into a separate n-dimensional embedding, same size as for each vertex, once again learned. The input for this step is the set of m observations of the neighboring vertices catenated with the weight on the connecting edge, creating an $m + 1$ dimensional vector for each neighboring vertex. All of these vectors are then summed item-wise and passed through a learned layer, creating an $n - 1$ dimensional vector. At this stage, the resulting $n - 1$ dimensional vector is divided by the number of neighbors and catenated with the number of neighbors, resulting in an n dimensional vector, and passed through another learned layer, resulting in an n-dimensional embedding representing the information about the neighboring vertices and the edges connecting them.

The end result of these two steps are an n-dimensional embedding representing a vertex and another representing it is neighbors, all created using the same learned weights for every vertex. In the paper, n is chosen as 64, but this value could be anything in theory, which would just enlarge the size of the network.

1. Why 64 dimensions on the vertex and edge embeddings?

   Finding the reason why will be important to make informed decisions on modifying the network. This seems to primarily be motivated by the fact that Python libraries tend to optimize better with data sizes that are powers of two. In theory, larger embeddings should allow more information to be encoded for the network to use which may become important to change later if different, more complex, problems are tackled, where more information may be necessary to embed.

1.1.3 Message Passing

Here there is a message pass layer and an update layer. The message pass for a vertex multiplies every neighbouring vertex's node embedding by the weight on the connecting edges and takes a sum over all of these neighbouring vertices. It then normalizes by the number of neighbours and catenates the edge embedding for the current vertex. This resulting vector is of size $2n$, an embedded vector representing the neighours for that vertex. This vector is passed through a set of learned weights, resulting in a n-dimensional vector.

The next is the update layer, which is the embedding for that vertex, catenated with the message, resulting in a vector of size $2n$. This is once again passed through another set of learned weights, into an n-dimensional vector, representing the new node embedding for that vertex.

The message then update is performed K times. Mentioned in the paper and corroborated in the code, this is done 3 times, but can be done however many times necessary.

1. Why have new network layers for these steps?

   As with the decision on the hidden layers sizes, understanding the justification for why certain steps are passed through learned functions before more calculations are performed will help make informed decisions on network changes. This decision seems to be based on the idea that a new learned layer should be added whenever new information is introduced to the network. This makes sense, as it makes sure that at every step of message passing or updating, that the network is given the opportunity to detect any relevant changes, whether existant or non-existant, that would influence the decision.

1.1.4 Readout

The readout layer goes through each vertex, summing the node embeddings for the neighbors of that vertex, dividing the result by the number of vertices in the entire graph and then passing it through learned weights, resulting in an n-dimensional vector.

The embedding for the vertex itself is then catenated to it, resulting in a vector of size $2n$ which is passed through another set of learned weights (without applying ReLU), giving in a single output value. Because this message passing process and readout is performed on very vertex, this gives you $|V|$ Q-values. The maximum Q value, which is associated to a particular vertex, is used to determine which vertex will have an action performed on it. Because there is only one Q value per vertex, this is associated to one single action that can be done on any vertex, which is to either add or remove it from the solution. For the maximum cut, which they use as an example in the paper, this is analogous to moving a vertex between the left and right sides of the cut depending on it is current state.

1.2 Reward Shaping and Training

1.2.1 Q Function, Q values and Training

The Q function is an idea derived from Q-learning [2] which proposes a way for an agent to learn how to behave in an environment. It is defined as the expected value of the discounted sum of future rewards for any state-action pair of an environment. When an optimal Q function is derived, the agent chooses which action to perform in a certain state by selecting the state-action pair which corresponds to the largest Q-value, which is expected to give them to largest reward. The equation following equation represents this idea:

$Q^{\pi}(s,a) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)|s_0 = s, a_0 = a, \pi]$

where π is a policy, mapping a state to a probability distribution over the actions, $R(s)$ is the reward for a given state and γ^t is a discount given to change whether the agent prefers immediate or future rewards.

This Q function is learned by a Markov decision process. The agent traverses the environment and rewards are given at each step depending on the result of the action the agent performs.

Trying to find this Q function was known to be unstable or even diverge when nonlinear function approximators, like neural networks, were used to try and represent it [3]. In *Mnih et al.*, they propose an approach to Q learning with two main ideas; namely experience replay and an iterative update, adjusting the Q values towards target values only periodically, while progressively constructing a training set throughout training, ensuring that the agent not only explores the environment, but also that the neural network can stabilize by forcing it to review previous state-action pairs [4]. The agent during training only chooses the actions associated with its current policy with probability ϵ and otherwise chooses a random action. They demonstrate experimentally that with these ideas, they were able to train a model that had significant performance improvements compared to other existing models on 49 different Atari games.

These ideas are replicated in the training of ECO-DQN. For every episode in the training phase, a random graph is sampled with a random solution set. Then, for each time step in that episode, which they set to $2|V|$, the agent chooses a random vertex based on the existing Q function with a probability ϵ and a vertex dictated by it is Q function otherwise. Then, the starting state, chosen vertex, reward and resulting state are added to the experience replay memory. Finally, after some fixed set of episodes, the network is updated by stochastic gradient descent on a minibatch sampled from the experience replay memory.

They set $\epsilon = 1$ and decrease it linearly to $\epsilon = 0.05$ over the first 10% of training.

1.2.2 Reward Shaping

The reward in a certain state is given by the difference between the cut value in that state and the highest cut value seen so far in the episode, divided by the

number of vertices. If the difference is negative, it is instead set to 0, meaning the current state is worse than the best one seen. The justification behind this choice is that a negative reward will discourage the agent from exploring states that give an immediately worse cut value, even if other cuts including that change later on may give better cut values overall. If this were to happen, it could discourage the agent from exploring different cuts and possibly cause it to stay stuck in a very small space near a local optimum. Because previously seen optimal states are stored in memory and returned as the result, it is therefore beneficial to later let the agent explore more states in an attempt to find more locally optimal states, even if not always better than the previously seen local optimum.

They also define a reward for reaching previously unseen locally optimal states of $\frac{1}{|V|}$. This is once again to encourage exploration. They state that local optima in combinatorial problems are typically close to each other and therefore by giving rewards for reaching new local optimums, the agent learns to hop between them during training as it is rewarded for this behaviour. In being rewarded for finding more unseen local optimums while being near a global optimum, it increases a behaviour pattern that has a propensity to finding this global optimum. Without this, they state that because there are far too many states to be visited in a finite time, which is typical for combinatorial optimization problems, it is therefore useful to focus on these subsets of states near local optimums.

There is no exact reason stated for the choice of value, however it can be hypothesized that if a constant value is chosen, this could cause disproportionate reward shaping on different sized graphs. For example, graphs that have significantly more locally optimal cuts could end up rewarding the network too much for simply finding local optimums instead of attempting to find a global optimum, while using the exploration of local optimums as a means to that end. Because larger graphs are likely to have more locally optimal states, it therefore makes sense to choose a value that decays as the size of the graph increases. Therefore a reward proportional to the inverse of the number of vertices in the graph makes the most sense.

The choice to not randomly change states when a new local optimum is found appears to be mostly a choice that the authors made intentionally as they want the network to find some space that has numerous locally optimum states near each other to explore, and the best way to do this is to encourage finding nearby local optimums instead of sending the agent in different random directions every time a new local optimum is found. They do state that because there are far more states than can be visited within a finite time period, it is therefore more useful to find some local optimums that are near each other and explore that specific space of possible solutions.

They demonstrate this behaviour by observing the probability during any given timestep that the agent will either revisit a state, find a locally optimal state or find the maximum cut on the validation set. They show that as the number of timesteps goes up, the probability that the agent revisits a state goes

up, as does the probability of finding the maximum cut, while the probability that the agent finds a local optimum goes up very quickly and stabilizes for the rest of the time steps, showing that the agent picks a certain set of states and explores the space around them to find new local optimums by revisiting previously seen states. This also demonstrates that the agent does not tend to get stuck in a specific local optimum and refuse to explore.

One thing this paper does not tackle is the issue of invalid solution states and how this would affect reward shaping, especially in a situation where the agent would be allowed to revert previously made actions. This isn't necessary for the Maximum Cut Problem, as any partition of the graph into two containing all vertices is a valid cut, but problems like the Traveling Salesman, Minimum Vertex Cover, Minimum Bisection and non-graph problems like the Knapsack Problem could possibly include states that are not valid solutions by either not being a complete path, not covering every edge in the graph, having two sets of different sizes or having an overfull bag, respectively for each problem. This becomes an issue for reward shaping especially, as it would therefore become impossible to compare invalid states to valid ones. A possible solution is to disallow invalid solutions, but with a problem like the Minimum Bisection, any change made to a valid state immediately makes it an invalid candidate solution. Because of this, some framework to allow an agent to explore invalid candidate solution states should be devised.

1.3 Discussion on Generalization

The internal structure of the MPNN should be generic for any graph or problem as it merely propagates information about observations on vertices throughout the graph. However, because of the large number of changes that would likely need to be made to observations (inputs) as well as the interpretation of the output for many other types of problems, it is likely that some changes to the internal structure would have to be made for the agent to make more educated decisions on new problems.

The main issue comes with the output and it is interpretation. Each vertex is represented by a single value as the output, and that value is interpreted as adding or removing it from the solution set, in the context where the solution is a subset of the vertices in the graph. More specifically for the Max Cut problem, the vertex associated with the maximum Q-value (output value) is taken and then either added or removed from the solution set, depending on whether it already belongs to it or not. This approach works fine for a problem like Maximum Cut or Minimum Vertex Cover where the solution can be represented as a set representing chosen vertices for the solution, however any extra constraints forces the output interpretation to be completely redesigned.

For example, the Traveling Salesman Problem where the solution is an ordered list of vertices could pose issues as the current implementation of simply adding or removing a vertex from the solution may not be adequate to allow the network to explore different states, and may have to be redesigned, like not allowing the network to remove states that aren't at the tail of the current path

or outside of the current path. Any other problem where the solution is an ordered list of the input would have this same issue.

The Minimum K-Cut Problem which can have an arbitrary number partitions of the graph would also not work with the current model as it would require extra decision making on deciding which set to move a vertex to. Because the number of sets can also change, this means the size of the output would have to change as the solution changes which would mean a new network would have to be trained for every graph size or k-value for the cut, or some significant rework of the output interpretation would have to be made. Any other problem where the solution is to split the input into k sets would have this same issue.

1.4 Benchmarks

The paper displays the performance of the network in reference to S2V-DQN, a similar paper where the algorithm does not allow for reversing actions. They also compare its performance against modifications of itself, namely where some observations are restricted, intermediate rewards are not given for reaching locally optimal solutions, as well as stopping it from reversing its actions.

They also a MaxCutApprox algorithm, which is a greedy algorithm choosing the vertex that provides the greatest cut improvment for one full episode of $2|V|$ steps. They implement two variations of this, one where it can reverse actions and one where it cannot reverse it's actions.

They also use an approximation ratio $C(S^*)/C(S_{opt})$. However, because S_{opt} cannot be calculated exactly, they use multiple optimizations. Specifically, they use CPLEX, an integer programming solver, as well as a pair of simulated annealing heuristics by Tiunov *et al.* (2019) and Leleu *et al.* (2019) in order to calculate S_{opt}.

The implementation of these different algorithms is not included in the codebase provided. However, the optimal solutions and cut values for the ER graphs with 40 vertices are provided for the testing graphs. The validation graphs of all sizes have their maximum cut values included.

They use GSet graphs G1-10 and G22-32 as well as the Physics/Ising dataset as benchmark graphs. These, as well as their optimal cuts and solutions are provided in the codebase.

1.5 Graph Generation

They train and test on Erdos-Renyi [5] and Barabasi-Albert [6] graphs. For training, they generate random graphs and perform a full episode of solving them, adding each state, action and reward to the experience replay memory. For testing, they have a set of 50 graphs, being either ER or BA graphs with different numbers of vertices which are used to compare the trained network to other algorithms.

2 Code

The code includes most, but not all, of the tests ran in the paper. Some of the algorithms are not included as well.

2.1 Running Code

They very generously provide a README file that specifies the exact commands to run in order to train, test and validate networks. These simply run specific files, namely the test_eco.py and train_eco.py files, for the respective graph sizes, so they can also be run by doing the typical process for running a python file. However, there is no code for reproducing the specific tables and plots. The data saved that is capable of generating the tables and plots in the paper are saved and therefore could be used to generate them. The code for generating that data is also provided and can therefore be analyzed and used for different problems, as different algorithms would have to be used for comparison.

2.2 Graph Generation

The random graphs for training are generated using the NetworkX library which includes the implementation for random ER and BA graphs. For training, they use the classes defined in src/envs/utils to generate new graphs at each step, adding every action done on the new graphs to the experience replay memory which is used to train the network. This ensures every training session is random and includes different data. The testing graphs are all the same, but change depending on the size of the training networks. For example, a network trained on graphs with 40 vertices will also have it is performance tested on graphs with 40 vertices or greater, but not less.

3 Minimum Cut

The Minimum Cut is not an NP-Hard problem, but it is still a combinatorial optimization problem that could be solved by ECO-DQN. To confirm that the idea is expandable to other problems, we will train an identical structure, using the negative of the cut value for determining the solution's reward with respect to the best observed, forcing the network to minimize the cut instead of maximize it as it will now receive greater rewards for smaller cuts.

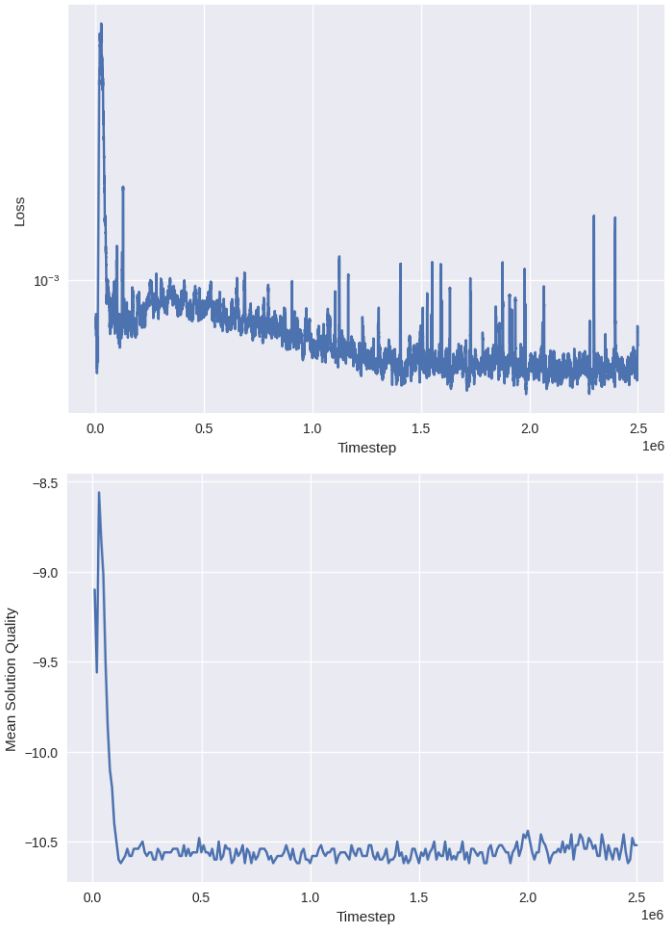
$R(S) = max(0, -C(S) - -C(S^*))$.

Just as with the original idea for ECO-DQN, at each step the best found solution is simply the solution with the minimal cut found so far during an episode. This best seen is then returned at the end of the episode after $2|V|$ steps.

3.1 Results

The result of training found in Figure 1 show this network is very similar to the maximum cut found in the ECO-DQN paper. It starts giving random solutions and eventually reaches a point where the cut value found stabilizes on the test graphs. This demonstrates that the network structure can be utilized to solve other problems. This was not tested against other algorithms, so it is possible that the cuts found are not minimal, but this provides some indication that at least the network can learn something about other problems than the Maximum Cut Problem.

Figure 1: Loss and Average Cut Found on Validation Graphs During Training, $|V| = 20$



4 Minimum Vertex Cover

A vertex cover is a set of vertices such that every edge in a graph has at least one endpoint in the that set of vertices. More formally, given a graph $G = (V, E)$, provide a vertex cover of G, being $V' \subset V$, such that for each edge $(u, v) \in E$, at least one of (u, v) belongs to V'. We want to find the smallest cardinality of any V'. This vertex cover for a graph is known as the Minimum Vertex Cover. Finding this cover is known to be NP-hard. To tackle this issue with the approach in ECO-DQN, a few problems need to be solved.

First, because this is not a cut problem, different observations that do not rely on calculating a cut need to be devised that can adequately report on the information about the vertices in the solution as well as the current candidate solution as a whole.

Further, some of the candidate solutions for this problem can be invalid candidate solutions. Due to this fact, the current reward system needs to be modified to take this into account, as we can no longer simply calculate the difference between the return values (i.e. $|V'|$) as some $|V'|$ that the network can generate may not be valid vertex covers. So, some punishment needs to be made for this invalid candidates, and a reward needs to be devised for finding smaller vertex covers. For this problem specifically, you can in theory disallow invalid candidates from being searched to avoid this issue. However, for the sake of allowing generality to other problems where this may not be the case, we will still allow invalid candidates to be searched.

Thankfully, some of these are somewhat trivial to solve, like the observations and reward, by either using ideas from ECO-DQN [1] and the similar S2V-DQN [7] which does not allow for exploration/undoing actions but does tackle the Minimum Vertex Cover.

4.1 Observations

In ECO-DQN, they have their set of observations from which we can draw to design the observations for the Minimum Vertex Cover. Some of these are even somewhat trivial to convert, namely:

- Observation 1, "Vertex sate, i.e. if v is currently in the solution set, S" is very easily translatable. We can directly copy this information, irrespective of if the current state is a valid solution.
- Observation 3 "Steps since the vertex state was last changed". This one also can be directly copied.
- Observation 4 "Difference of current cut-value from best observed". We will translate this to the difference between the current cover set size and the best cover set size. we will ignore the validity of the solution, so as to adquately represent all of the information about the current candidate without hiding some of the information due to a candidate solution being invalid.

- Observation 5 "Distance of current solution set from the best observed". In the paper they do not define this, but it is explained through their implementation in Section 1.1.1, Node Embeddings. We can also use this information for the Minimum Vertex Cover in the same way.

- Observation 6 "Number of available actions that immediately increase the cut value" can also be translated. Namely, counting the number of actions (or vertices) that when flipped will reduce the number of vertices in the cover. Again, in this case we will ignore the validity of the solution created by changing that vertex state.

- Observation 7 "Steps remaining in the episode". This, once again, can be directly copied.

What's not mentioned in the paper is how these values are normalized. Some of the normalizations described in the paper can be identical, however there is one outlier, namely the fact that Observation 4 is normalized by the largest non-zero weight of a vertex. Instead of using this value, we will create an analogous value, which is described in greater detail later, which is the largest possible score difference between two states. This happens to be the score for a perfect valid solution (in theory, $|V'| = 0$) and the maximum degree of the graph. The reason for this is that we will evaluate the "score" of an invalid solution by counting the number of uncovered edges and the score of a valid solution with be $|V/V'|$. Therefore the maximum difference, or local change, in the score of a solution is if you add the vertex of highest degree to the solution, resulting in a perfect solution. This is of course not the true value, but it is analogous to the largest change in cut value from the empty candidate solution. This could also be changed to be simply the number of vertices in the graph, because that is what we are comparing, but for the sake of consistency this is what will be used for the network, as they name this factor the maximum local reward.

Observation 2 in the ECO-DQN paper was the immediate cut change if the vertex state is changed. We would interpret this as the immediate change in the size of the vertex cover's set ignoring validity, but this is implied by Observation 1 for the Minimum Vertex Cover and would therefore provide no added information to the network.

With all of this done, we want to add information about the change in validity of the candidate solution both for local vertex changes, as well as compared to the best seen solution. Therefore, we can add the following observations:

1. Immediate change in the number of edges covered on vertex flip. This counts the number of edges that were previously uncovered but are now covered as a negative number and positive if it increases the number of edges that are uncovered. This value is normalized by the total number of edges in the graph to account for graphs with different numbers of edges and vertex degrees.

2. The solution's validity on vertex flip. This represents whether the solution

produced by changing whether the vertex is in V' or not is valid (1) or invalid (0).

3. Number of actions that immediately increases the number of edges covered by the solution. This simply counts the number of actions that make the solution approach validity. This value is normalized by the number of vertices in the graph.

4. Difference in number of edges covered from current solution and best observed. This counts the number of edges covered by the best observed solution and the current candidate and compares the values. This value is positive if the current candidate covers less edges than the best observed. This value is normalized by the number of edges in the graph.

5. Validity of current solution.

This means we have the following observations for the Minimum Vertex Cover:

1. Observation 1: Vertex state
2. Observation 2: Steps since the vertex was changed
3. Observation 3: Immediate change in the number of edges covered on vertex flip
4. Observation 4: Immediate change in the solution's validity on vertex flip
5. Observation 5: Difference of current set size from best observed
6. Observation 6: Difference of number of edges covered by current solution from best observed
7. Observation 7: Distance of current set from best observed
8. Observation 8: Number of actions that immediately reduce the set size
9. Observation 9: Number of actions that immediately increase the number of edges covered by the solution
10. Observation 10: Validity of current solution
11. Observation 11: Steps remaining in episode

This leaves us with local observations 1-4 and global observations 5-11 which all grant information about the state of the solution at that time, both for local changes and global differences from the best observed solution.

4.2 Reward Shaping

Due to the difference between how proposed solutions are going to be in the Maximum Cut and Minimum Vertex Cover, some different rewards are going to be required to adequately represent the problem. In ECO-DQN [1], the reward for a certain state is framed as

$R(S) = max(0, C(S) - C(S^*))/|V|$

where $C(S)$ is the cut value for state S and S^* will be the previously best seen cut value. They also grant intermediate rewards $\frac{1}{|V|}$ any time a new locally optimal cut is found, which is a cut that has not yet been seen where any change to a vertex reduces the value of the cut.

In S2V-DQN [7], the reward for a Minimum Vertex Cover they define as $R(S, v) = -|S| - -|S'|$ being the change in their cost function when going from state S and adding vertex v to it, resulting in state S'. In these cases, the state is the set of vertices chosen for a candidate solution. We can reformulate their idea slightly to coincide with the idea in ECO-DQN to allow for exploration by instead looking only at defining the reward on a specific state in comparison to the best seen instead of the previous state.

In our case, the best seen is not so simply defined. Because constructed candidates may not be valid, some way to compare invalid candidates to valid ones has to be devised such that we can choose a previous "best" candidate to use as a comparison for future candidates. What we want is a reward mechanism that allows valid candidates to always be chosen as the best over invalid ones, as well as for valid candidates to be compared to each other in such a way that a candidate that improves the solution gives a better reward than ones that don't. Similarly for invalid candidates, we would like for them to be compared to each other in such a way that an invalid candidate that is closer in some measure to being a valid solution gives a better reward than one that is further away. One way to do this is to ensure that valid solutions gives positive rewards increasing in magnitude depending on it is quality, and for invalid solutions to give negative rewards increasing in magnitude depending on it is relative lack of closeness to being a valid candidate.

For the Minimum Vertex Cover, the score of a valid candidate can be defined in a similar way as in S2V-DQN by calculating the size of the set not including the current candidate solution, $|V/V'|$. This grants higher scores to smaller candidate set sizes. Therefore when comparing valid candidates using the same reward function as ECO-DQN, we will get positive rewards for smaller set sizes. We can of course normalize this value by the number of vertices to accommodate different graph sizes.

For invalid candidates, their scores can be the negative of the number of edges not covered by the candidate. Therefore, invalid candidates being compared will give greater positive rewards for getting closer to a valid solution. Of course, this also will result in positive rewards for moving from an invalid candidate to a valid one. To normalize this, we can use the number of edges in the graph, to account for graphs with differing numbers of edges. A property of using this for getting the score of an invalid candidate is that valid candidates always have

exactly zero uncovered edges.

Including these two ideas, we can define the score of a candidate for the Minimum Vertex Cover as being

$score(S) = validity(S) * |V/S| - edges\_uncovered(S)$.

Where S is the current candidate solution state being evaluated. Normalizing this score would be similar,

$norm\_score(S) = validity(S) * |V/S|/|V| - edges\_uncovered(S)/|E|$.

To elaborate on the previous point mentioned in the Observations section of the Minimum Vertex Cover, this means that when starting from the empty candidate solution, the largest possible change in score would be difference between the score for $|V/S| = |V|$ and the score for an invalid solution, maximizing the number of edges uncovered such that when a vertex is added, the solution is now valid. This would be like adding the vertex of highest degree to the current candidate, resulting in the empty solution, while also being a valid solution.

When calculating the reward, we can use the same idea as ECO-DQN, $R(S) = max(0, norm\_score(S) - norm\_score(S^*))$, where S^* is the previously best seen candidate. Of course, by comparing scores we can select the best score as the best previous candidate, as the scores increase in value for valid candidates of smaller set size and decrease as the set size becomes larger or as the candidate is not valid. The following shows the results of comparing the scores of different solutions S and S^* for determining a new best seen candidate:

1. Invalid S compared to invalid S^* only give positive rewards when S is closer to being valid than S^*. Therefore S is only a better candidate if it covers more edges.

2. Invalid S compared to valid S^* will never give a positive reward. Therefore an invalid S cannot become the new best seen if S^* is a valid candidate. This ensures we always return a valid candidate (if one is found) at the end of a search.

3. Valid S compared to invalid S^* will always give a positive reward. Therefore is S^* is invalid and S is valid, S becomes the new best seen candidate.

4. Valid S compared to valid S^* will only give positive rewards if S is a better solution than S^*. Therefore S only become

As with ECO-DQN, we will also give intermediate rewards for finding new local optimums in order to avoid situations where rewards become incredibly scarce and to encourage it to find new locally optimal states to explore. This reward will be the same amount as well, of $frac1|V|$ to account for larger graphs having more locally optimal states.

4.3 Training

Training this network will be algorithmically identical to ECO-DQN [1] using the ideas from Deep Q-Learning described in *Mnih et al.* [4].

Because the original test graphs were generated using discrete edges weights of either 1 or -1, we first recreate the test graphs with the same edges but setting all the weights to 1. This functions as an undirected, unweighted graph for evaluating a Minimum Vertex Cover. All of the training graphs will also be generated using uniform weights, on ER graphs with edge probability of 0.15. Training and testing graphs will have the same number of vertices.

Within these parameters for training and testing graphs with $|V| = 20$, the network seems to rapidly improve it is performance on the test graphs and reaching a plateau where it remains for the remainder of the training time. This same trend is observed for larger training graphs. Figures 2 and 3 show the training loss and average solution found on the testing graphs over the training period.

Figure 2: Training Loss and Average Solution $|V| = 20$

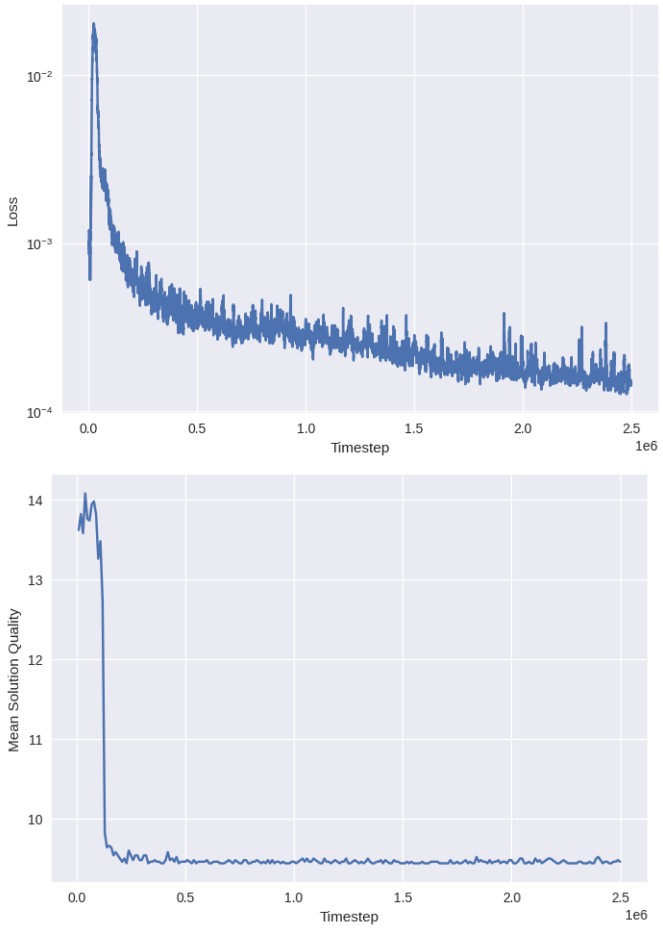
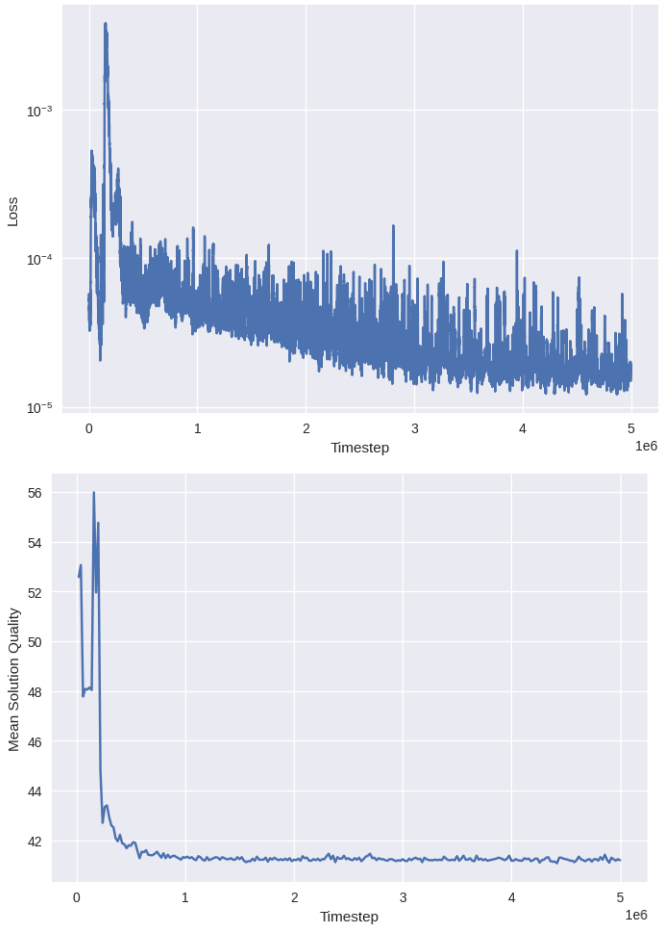


Figure 3: Training Loss and Average Solution $|V| = 60$



4.4 Testing

To test the network, it was compared to other algorithms on a set of validation graphs of different sizes. First, the CPLEX linear program solver was set to solve the Minimum Vertex Cover. I also used a greedy algorithm which takes an initial state $S \subseteq V$ and makes greedy actions for $2|V|$ steps based on the score function defined for the neural network. I also implemented a matching algorithm, which selects a random uncovered edge and puts the nodes incident on the edges into S. Once there are no more uncovered edges, the algorithm terminates. I also evaluated the minimum weighted cover implemented by Python's NetworkX library, which is based on the work of *Bar-Yehuda and Even* [8], that has a worst case run-time of $O(m * log(n))$ and approximation ratio $2 - \frac{1}{k}$, where k is the smallest integer that satisfies $(2k-1)^k \geq n$.

Because some of these algorithms are randomized either through implementation or input, any of these were instead run 50 times on each validation graph and the mean solution was selected as the found solution for that graph. The following results show the average cover found by CPLEX, matching algorithm, greedy algorithm with a randomized initalization and empty initialization, local-ratio algorithm and three initializations for the neural network; one where the initial states were random in which case the mean solution found was used over 50 attempts from random states, one test from an empty starting state, and one test from the state where every node is in the cover. Figure 4 includes the results found for training the network on graphs with $|V| = 20$.

These results show that the Minimum Vertex Cover is likely a relatively easy problem to solve on small graphs, as the network gets very close to the optimal solution on average, but only marginally improves on the solutions found by the minimum weighted cover solution from [8], as well as the greedy solution, which picks greedy for edges and culls vertices randomly until a local optimum is found or the algorithm terminates. More tests on graphs significantly larger than the current ones evaluated should be run to determine whether this truly scales up to large graphs or the performance degrades.

5 Generalization, Single Subset

In order for this network to solve different problems, the observations, rewards and actions needs to be modified in such a way that they can be generalized to other problems on graphs with an input $G = (V, E)$. Currently, the most intuitive of these to generalize is the actions. In the network's current state, the action is defined in a way that exclusively works for problems where the solution is a subset V, minimizing or maximizing some function. For example, in both maximum and minimum cut problems, the solution is a subset of the vertices of a graph. For the Minimum Vertex Cover, this principle also applies, as the solution will be some $S \subseteq V$. This specific type of output representation exists for numerous graph problems. For example, the Minimum Quotient Cut also requires the solution to be a subset of V. Therefore, if some way of measuring a reward and observations for this problem exists, it should also be solvable with this network structure. In general, due to the way the actions are defined through the Q function, any problem where the solution is some $S \subseteq V$ should be a target of study for this network to be able to solve.

5.1 Reward

5.2 Observations

References

[1] T. D. Barrett, W. R. Clements, J. N. Foerster, and A. I. Lvovsky, "Exploratory combinatorial optimization with reinforcement learning," in *Proceedings of the Thirty-fourth AAAI conference on Artificial Intelligence*, arXiv, 2019.

[2] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[3] J. Tsitsiklis and B. Van Roy, "Analysis of temporal-diffference learning with function approximation," in *Advances in Neural Information Processing Systems* (M. Mozer, M. Jordan, and T. Petsche, eds.), vol. 9, MIT Press, 1996.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[5] P. Erdős, A. Rényi, *et al.*, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[6] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of Modern Physics*, vol. 74, pp. 47–97, jan 2002.

[7] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[8] R. Bar-Yehuda and S. Even, "A local-ratio theorem for approximating the weighted vertex cover problem," in *Analysis and Design of Algorithms for Combinatorial Problems* (G. Ausiello and M. Lucertini, eds.), vol. 109 of *North-Holland Mathematics Studies*, pp. 27–45, North-Holland, 1985.

Figure 4: Average Cover Found on Validation Graphs