Progress Report

Annabelle Cloutier

April 10, 2023

Contents

1 Paper

1.1 Network Structure

All of the below 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.

1.1.1 Node Embedding

The MPNN (Message Passing Neural Network) structure used converts each vertex into a set of m observations and encodes them into a n-dimensional embedding using learned weights. The paper states that they use 64 dimensional embeddings and they do in code, but this could be any size.

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 make an informed decision on whether to add or remove it, or in this agent's case, possibly reversing an action.

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. Because this observation gets larger as time goes on and a vertex is unchanged, the agent will be further incentivised to choose other vertices if it gets stuck in a loop choosing the same few vertices over and over. They also state it allows the agent to exploit reversible actions just as with observation 2. 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 just as it will discourage loops.

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.

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, specifically in src/envs/spinsystem.py, in the SpinSystemBase class' step function.

   Example for clarity:

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

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

   Difference: 1.

   Counting the number of different vertices they do in code by subtracting the two and counting the number of non-zero values. This can also be done using a bitwise XOR and doing a sum on the resulting bitmask.

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.

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.

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 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, catenated with the number of neighbors, and passed through another learned layer, resulting in an n-dimensional embedding representing the edges for a vertex.

The end result of these two steps are an n-dimensional embedding representing a vertex and another representing it's neighbors, all created using the same learned weights for every vertex. Meaning there will be $2|V|$, n-dimensional embeddings.

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

   It's nice that it's 64 but finding the reason why will be important to make informed decisions on modifying the network.

1.1.3 Message Passing

Here there is a message pass layer and an update layer. The message pass per vertex is summing the products of the connected vertices and the weight, divided by the number of neighbors, catenated with the edge embedding for that vertex and then passed through 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 and passed through another set of learned weights, into an n-dimensional vector, representing the "new" 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.

1.1.4 Readout

The readout layer goes through each vertex, summing the 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 the resulting vector and passed through another set of learned weights (without applying ReLU), giving in a single output value. This value represents the Q-value for that vertex, which is used by the algorithm to determine which vertex will be added/removed from the solution set on that step. The vertex associated to the maximum Q-value is added to the solution set if it doesn't yet belong to it, or removed from the solution set if it belongs to it.

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 environments. 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 [4]. The agent during training only chooses the actions associated with its current policy with probability $1 - \epsilon$ 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, the agent chooses a random vertex based on the existing Q function with a probability ϵ and a vertex dictated by it's 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 time steps, the network is updated by stochastic gradient descent on a minibatch sampled from the experience replay memory.

1.2.2 Reward Shaping

The reward for 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's instead set to 0. 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. If this were to happen, it would discourage the agent from exploring different cut values and cause it to stay in a very small space near a locally maximal cut value. Because previously seen optimal states are stored in memory, it's 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 locally optimal states of $\frac{1}{|V|}$. This is once again to encourage exploration. Because the previously seen best cut is stored, it's beneficial for the agent to explore other states, even if other local optimums may be worse than the previously found best cut. 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. 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 optimum cuts could end up rewarding the network too much for imply finding local optimums instead of attempting to find a global optimum, while using the exploration of local optimums as a means to that end. 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 therefore 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's 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 local optimum and explores that space by revisiting previously seen states.

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 or Bin Packing Problem could possibly include states that are not valid solutions by either not being a complete path, not covering every vertex, having two sets of different sizes or having items that are not in a bin, respectively for each problem.

1.3 Discussion on Generalization

Most of the internal structure is generic for any graph, however because of the large number of changes that would likely need to be made to observations 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.

Everything outside of observations (input) and Q-values (output) only propagates information throughout the graph. This means the internal structure likely could remain mostly the same for any problem, so long as good decisions are made for the observations and that the output or it's interpretation are modified to fit new problems.

It is possible for more complex problems that some of the internal structure for the message and update sections of the network may have to change in order to accommodate for extra information. It is also possible that global observations could be embedded somewhere within the internal structure of the network instead of as input observations for every vertex.

The main issue comes with the output and it's 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 would not correctly work as the current implementation of simply adding or removing a vertex from the solution could not work.

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.

The Minimum Bisection Problem can be represented as a set of chosen vertices, however the extra constraint that the number of chosen vertices has to be the same as the number of unchosen vertices makes the current output interpretation incomplete for solving this problem.

1.4 Benchmarks

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

They use GSet graphs G1-10 and G22-32 for calculating the approximation ratio, as well as the Physics dataset.

1.5 Graph Generation

They train and test on Erdos-Renyi [5] and Barabasi-Albert [6] graphs.

2 Code

The code holds the capability of running all of the above tests, but they're not explicitly written and must be generated via self-written code.

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. However, there is no code for reproducing their specific tests. These all need to be hand-coded. The only results I've reproduced so far are training and testing.

2.2 Graph Generation

Unsure of the exact implementation, it seems they use NetworkX's ability to generate random graphs in order to do this. It's completely unnecessary though seeing as they have many test, validation and benchmark graphs pre-built within their code that can be used. If absolutely necessary, creating random graphs and storing them within a pickle file as they do would likely lower the workload on inputs as the code for parsing through these types of files is already prebuilt.

The code for generating new random graphs exists in src/envs/utils. This includes the code for generating Erdos-Renyi and Barabasi-Albert graphs, as well as classes for numerous other types of random graphs.

2.3 Input Conversion

The code for converting graphs into observations seems to exist within the file src/envs/spinsystem.py but further code inspection is needed to see exactly how this information is represented and decoded by the agent.

2.4 Benchmarks

They generously provide testing, validation and benchmark graphs in Pickle files, which is a special type of object file for Python, as well as their solutions, which are also stored in Pickle files, but are simply a boolean (technically floats, just 1.0 and 0.0) list for determining whether a vertex is in the solution set or not.

3 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. The vertex cover of smallest magnitude 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:

1. What observations will be used for each vertex
2. What will the reward be for finding a valid solution
3. What will be the punishment (if any) for finding sets that are invalid solutions
4. Should an intermediate reward be used for finding valid solutions that are locally optimal
5. Should we allow invalid candidate solutions

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.

The last question we'll tackle in two parts. First, we'll attempt this problem by giving the network valid solutions and not allowing it to select invalid ones by severely penalizing those actions and randomizing a new solution to give it at test time to avoid the network getting stuck in local optimums in case it continuously tries to suggest invalid candidates.

3.1 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 observed.
- 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 removed from the set will reduce the number of vertices in the cover.
- Observation 7 "Steps remaining in the episode". This, once again, can be directly copied.

The observation that will change is the immediate cut change. Due to the nature of the Minimum Vertex Cover, the immediate change in the cover set size is implied by observation one. Therefore, we will change this slightly to instead represent an observation for in the case where we'll allow invalid candidates:

1. Difference of current edge cover from the best observed. The "edge cover" in this case will be the number of edges covered by the state, which can be less than the total number of edges in the graph.

We also want to embed information about the validity of the current state in case we allow invalid candidates, therefore we also add the following observations:

1. Immediate change of edge cover on vertex change, which will compute the difference of the number of edges that are covered by the current candidate compared to the best observed.
2. Whether the current cover is valid; that it covers all edges.

In the end, we'll have 9 observations, some of them only used for when the network will be permitted to explore candidates that are invalid solutions. Observations 1-6 will be used regardless of whether we allow invalid candidates, observations 7-9 will only be used if we allow invalid candidates.

1. Vertex state i.e. whether the vertex is in the current solution set.
2. Steps since the vertex state was last changed.
3. Difference of current cover set size from the best observed.
4. Distance of current solution set from the best observed.
5. Number of available actions that immediately decrease the number of vertices in the solution.
6. Steps remaining in the episode.
7. Immediate change in current edges covered.
8. Difference of current edges covered from best observed.
9. Whether the current cover is a valid cover.

In this case as with ECO-DQN [1], we have local (1-3, 7) and global (4-6, 8-9) observations.

To start, we'll do a basic algorithm to select edges at random, picking the nodes incident on the edge and removing all edges incident on the chosen nodes until all edges are covered. We will then use this generated solution and give it to the agent as a starting point. In this scenario, we won't allow for invalid candidates, and will give a large negative reward for choosing invalid candidates. If an invalid choice is suggested during test time, we'll generate another random valid solution and get the network to continue from the newly generated solution.

Where we'll allow for invalid candidates, we'll generate a random candidate as the starting point, irrespective of validity.

3.2 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
$R(S) = max(0, C(S) - C(S*))$
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 locally optimal cut is found, which is a cut 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.

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. This previous best will be defined as follows:

1. Any new candidate solution S such that the number of edges covered by the vertices in S is greater than S^* will be favoured.

2. If candidate solution S and S^* cover the same number of edges, we choose the candidate solution with the lower number of vertices. Therefore $S^* \leftarrow S$ if $|S| < |S^*|$.

These are greedy and cover all cases and will ensure that any valid solution will always take precedence over an invalid one by point 1, as well as ensuring that any new valid solution with a lower number of vertices will be chosen by point 2. Any invalid candidate will only replace a previous invalid candidate

if it covers more of the graph. If they cover the same number of edges, the candidate with less vertices is chosen.

These two cases also cover the initial attempt for only allowing valid candidates. Valid candidates are always chosen if they have a smaller set size, and therefore in the case where invalid candidates are dissallowed, we can use these same principles.

This is a greedy approach for tracking invalid candidates, but once a valid candidate is chosen, it makes more sense to track the previously best seen valid candidate as the previous best rather than use an invalid candidate. This is not just for computing rewards, but also to ensure that once the agent terminates it's search, that the best candidate seen should always be a valid solution to the problem, regardless of whether it is the correct solution.

As for the reward, we can define this in a similar way as S2V-DQN, with a modification: $R(S) = max(0, |S^*| - |S|) + max(0, cover(S) - cover(S^*))$. This idea is similar to S2V-DQN but instead uses the idea of comparing the current candidate to the previous best, and without punishing the agent for choosing covers of larger magnitude to allow for exploration as explained in ECO-DQN. We also add a term for comparing covers, which we define as the number of edges covered by the vertices in S. This is to reward the network for choosing candidates that cover more edges up until it finds a valid candidate. Each of these will be normalized by the number of vertices for the case of set size and by the number of edges for the cover, to avoid disproportionate reward structures on different graph topologies.

Some things to note:

- If S^* is a valid candidate, the only candidates that give rewards greater than 0 are new valid candidates that are better, or invalid candidates with a smaller number of vertices. This is due to the second term of the reward being 0 in any case once a valid candidate is found.
- If S^* is an invalid candidate, any valid candidate always gives a reward greater than 0 because any valid candidate S would necessarily have $cover(S) > cover(S^*)$.
- If S^* is an invalid candidate, any invalid candidate always gives a reward greater than zero so long as it has a smaller set size, a larger cover, or both.
- Neither point 2 or 3 are necessarily favoured over the other. That is to say, valid candidates do not always give greater rewards than invalid ones, meaning the reward isn't purely greedy.

These three facts are important to note as it means the agent is intrinsically rewarded for having a lower number of vertices in a candidate solution even when S^* is a valid candidate, therefore giving the agent incentive to explore invalid candidates in search for better valid ones.

As in ECO-DQN, rewards may become incredibly scarce as better and better valid candidates are found. Therefore, we will also give an intermediate reward

of $\frac{1}{|V|}$ any time a new locally optimal state is found. In this case, a locally optimal state is one where S is a valid solution and removing any vertex from S results in a situation where $cover(S) < |E|$. As in ECO-DQN, this is to encourage exploration. Paired with the lack of negative reward (punishment) for leaving an optimal state, this should allow the agent to traverse out of local optimums into new ones.

For the simplified version where we won't allow for invalid candidates, we'll instead add a negative reward equal exactly one if the chosen vertex to change results in an invalid candidate. As such the reward will be $R(S) = max(0, |S^*| - |S|)$ in the case of a valid cover, and simply $R(S) = -1$ in the case where it suggests an invalid candidate. The reason for not normalizing this value is it doesn't matter the degree of which the solution is invalid in this case, and with everything else being normalized such that the values are all between zero and one, this will force the value down without being too disproportionate comparatively.

3.3 Training

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

3.4 Implementation and Results

Notes for when I start implementation:

- src/envs/utils.py defines an enum Observations for maximum cut oberservations. Create a new one for Min Vertex Cover. This does not include calculations
- src/envs/spinsystem.py defines the operations on a graph required for the Maximum Cut in SpinSystemBase, an abstract class.
- methods for SpinSystemBase could likely be overwritten in a new class for MVC. The step() method contains information for computing observations AND reward. This is where attention should be focused to create the new class that mimics behavior to minimize the amount of code I need to copy (like the entire MPNN which I don't yet need to modify)
- Consider making test graphs with some known answers for observations and rewards that can be confirmed to ensure code is bug free before moving forward. (i.e. unit tests)
- Still need to find Integer Program for comparison. Do this last. It's useless to have one if I can't even get the above working.
- Start with basics: do basic solution of MVC then use the network to refine (i.e. don't allow invalids)

3.5 Generalization

Just some footnotes/brainstorming for now

To generalize these ideas to any problem, some framework needs to be solidified for choosing observations and rewards. Something related to the constraints of the problem (if any) would need to be specified. For Minimum Vertex Cover there's a simple way of doing this as described in this section, but extending this to other problems is much more difficult. Further, there's no real way that I know of that could actually confirm whether other frameworks for learning would actually yield good results other than manually verifying it by creating the observations and reward mechanism then training a network (obviously bad and not very useful).

If the framework I thought of for MVC works as well as the different framework for MaxCut did, I'd say there's at least some merit to the general structure and idea of ECO-DQN and the idea of encoding some reward and even extra observations to encode validity constraints. It's a matter of formulating observations and rewards so they can encode not only the actual problem constraint (i.e. cut size for max cut or minimum bisection, vertex cover size for minimum vertex cover, path length for traveling salesman), but also some sort of validity constraint (reward to determine whether a candidate is a valid solution) that can generalize to different problems (i.e. size of partition for minimum bisection, whether a cover actually reaches every edge for minimum vertex cover, whether a path is a full tour for traveling salesman).

Many of the constraints would likely need to be formulated such that they change depending on whether the problem is maximization or minimization. For example, choosing $max(0, cover(S) - cover(S^*))$ for MVC only makes sense because you're trying to maximize the number of edges covered. When S^* is a valid candidate, this term will always be 0. If you're instead trying to minimize something, the terms would have to be swapped. For example, in the minimum bisection, you'd want to minimize the absolute difference betweeen the size of the partitions of the graph. Therefore, the idea for that problem would be more like $max(0,$ partdiff$(S^*)-$partdiff$(S))$ for a new candidate. Note that taking the absolute difference doesn't work because we actually want negative values to exist to reach 0.

Definitely problems arise if the decision isn't binary, adding or removing a vertex from a single solution set, like with Minimum K-Cut.

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.