Heuristic for the Tree Augmentation Problem using Reinforcement Learning

Cody Klingler

May 4, 2022

1 Introduction

The Tree Augmentation Problem (TAP) is an NP-hard optimization problem with applications in networking. There exist polynomial-time approximation algorithms which guarantee a solution within a constant factor of the optimal solution. These algorithms don't have public implementations and most are quite large or complex [GKZ18].

The goal of my project is to create a heuristic to solve the TAP using reinforcement learning to serve as an alternative to the approximation algorithms.

2 Tree Augmentation Problem Definitions

A graph is considered to be *2-edge connected* if there is no edge whose removal leaves the graph disconnected. See Figures 1 and 2.

The *Tree Augmentation Problem* (TAP) is defined as follows: Given a graph $G = (V, E)$ and a tree $T = (V, F)$ where $E \cap F = \emptyset$, find a minimal $E' \subseteq E$ such that $(V, E' \cup F)$ is 2-edge connected. See Figures 3 and 4.

An edge $(u, v) \in E$ is said to cover $(u', v') \in F$ if the tree-path between u and v, $PATH(u, v) \subseteq F$, contains (u', v'). Note that adding (u, v) to F creates a cycle along $PATH(u, v)$. Therefore removing an edge $e \in PATH(u, v) \cup (u, v)$ from $(V, F \cup (u, v))$ does not disconnect the graph. The graph $(V, E' \cup F)$ is 2-edge connected if and only if each edge in F is covered by an edge in E', therefore the TAP can be restated as "find the minimal E' that covers all edges in F". See Figures 5 and 6.

The concept of covering can be used to reduce a TAP problem. If it is known that an edge $(u, v) \in E$ is in the solution E', then all edges along $PATH(u, v)$ may be contracted without loss of generality. See Figures 7 and 8.

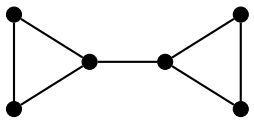


Figure 1: Not 2-edge connected

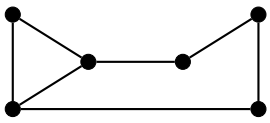


Figure 2: 2-edge connected

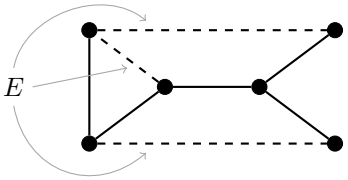


Figure 3: TAP Problem

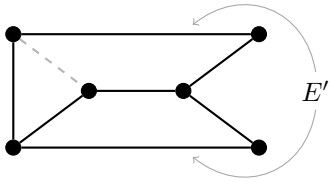


Figure 4: Solution to Figure 3

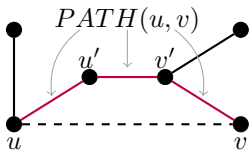


Figure 5: (u, v) covers (u', v')

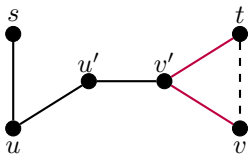
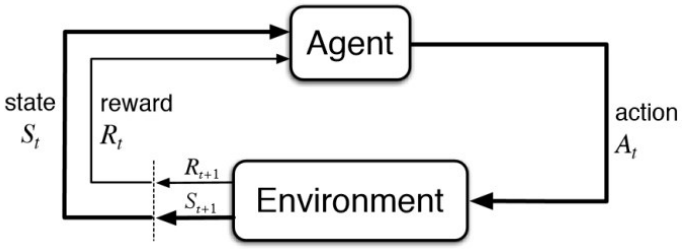


Figure 6: (u, v) does not cover (u', v')

3 Approaching TAP with Reinforcement Learning

3.1 Reinforcement Learning Principles



The Reinforcement Learning Model

Reinforcement learning (RL) consists of two primary components, the *agent* and the *environment*. The agent is essentially synonymous with the model, which is often some neural network based decision making algorithm. The environment is an abstract space that the agent interacts with by making decisions based on information given to it. The decisions made by the agent are called *actions*, and the domain of all potential actions is called the *action space*. Once an agent takes an action, information is given to it by the environment that consists of a *reward* and an *observation* or *state*. The reward is a quantity that serves as feedback about how "good" the action was. The objective of the agent is to maximize the total reward that it receives from the environment. The other piece of information given to the agent is the observation, also called state, which serves simply as tool to help the agent make decisions. The *observation space* describes the form and content that observations take. When an agent acts on an environment to a state of completion, it is said that it has completed an *episode*.

The following sections will describe how attributes of RL model fit the TAP in the heuristic.

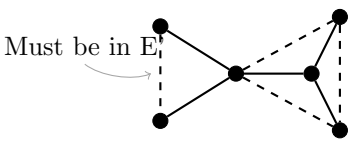


Figure 7: Contractable TAP instance

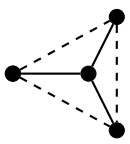


Figure 8: Figure 7 after contraction

3.2 Observation Space

The observation space for the agent should describe all of the relevant information needed to solve the problem. In this case, the tree edges and the graph edges. This could be represented by a dictionary or a list, but I have selected the adjacency matrix for its ease of use and consistent size.

3.3 Actions and State Changes

Actions correspond to edges in the graph or E, which may be represented by tuples such as (u, v). For each state of the problem, one edge will be selected and the TAP will be reduced using tree path contraction. The reduced problem will be the next state. This ensures that the agent must arrive at a valid solution at the end of an episode.

3.4 Actions Spaces

Actions are represented as tuples, but the way tuples are chosen may represented in more than one way that are distinct when it comes to implementing the model.

The following examples show the *discrete* and *multi-discrete* action spaces for a graph with 4 vertices.

Discrete action space
Select a single action $\alpha \in S$, where
$S = \{(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),...\}$

Multi-discrete action space:
Select an action (α, β) such that $\alpha, \beta, \in S$, where $S = \{1, 2, 3, 4\}$

The multi-discrete action space is much more natural to the problem, and as you will see, performs better in practice. Note that neither of these action spaces restrict an agent from selecting an action that corresponds to an invalid edge, which in this case would be an edge not in E'. This is a problem as it not only slow down the execution of an episode, but also hinder the agent from learning. This is addressed by the recent notion *action masking* which outright prevents the agent from making invalid actions.

3.5 Masked Action Spaces

Action masking algorithms are a new development in RL, but can significantly boost the performance of agents [TLCY20]. Once an RL agent is done training with masked actions, the masking can actually be lifted, which is not always necessary. Because masking is new, implementations are limited and unstable. The masked algorithm used in this heuristic uses the discrete action space because of limitations in multi-discrete masking.

The *masked discrete* action space of a graph for the TAP is simply the remaining edges in E. The masked discrete action space for a graph with 4 vertices could look as follows,
Select a single action $\alpha \in S$, where $S = \{(1,3),(1,4),(3,2),(4,3)\}$

Notice that redundant, yet valid actions may also be removed.

3.6 Rewards and Invalid Actions

For the unmasked action spaces, the agent will learn to avoid invalid actions on its own if it leads to a negative reward, which is also known as a punishment.

I believe the optimal reward function is a constant negative reward for selecting *any* edge because the objective function for the agent and TAP would be the same, which is literally to minimize the number of edges selected. This reward function will also make the agent avoid invalid actions given that they do not reduce the environment.

3.7 Training

The agent will be repeatedly trained on random initializations of the graph and tree of a fixed size and density. A better way of doing this is described in 5.5.

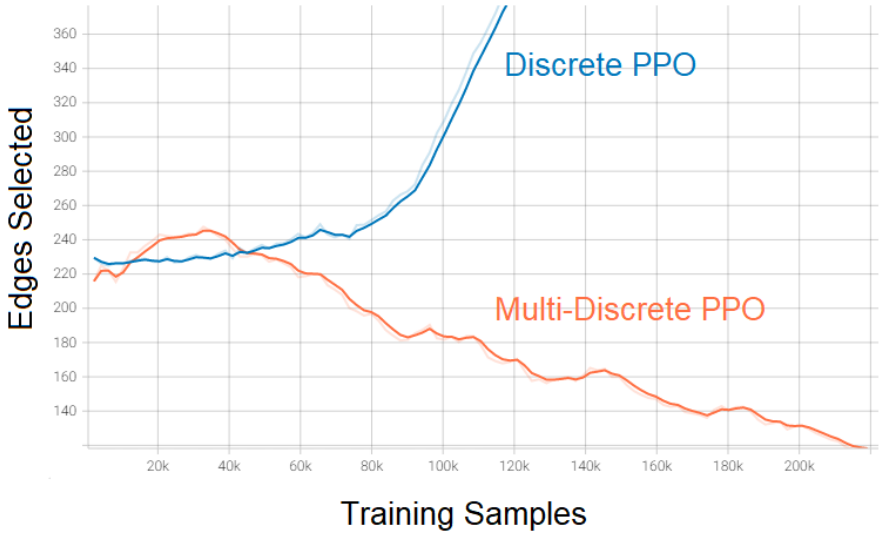
3.8 Implementation

The model selected was Proximal Policy Optimization (PPO), which uses a neural network (NN) to predict total reward for a given action. The NN is repeatedly re-trained after a series of episodes. PPO will randomly select what it thinks to be sub-optimal actions in a monte-carlo fashion in order to find actions that it previously underestimated. PPO is *the* state-of-the-art in RL and is generally the go-to algorithm for solving most problems.

The implementation of PPO is used is provided by Stable Baselines 3, which is a fork of OpenAI's Baselines. Both of these libraries are built on the abstractions of OpenAI's Gym. The masked-PPO algorithm is from Stable Baselines 3's experimental fork named "contrib".

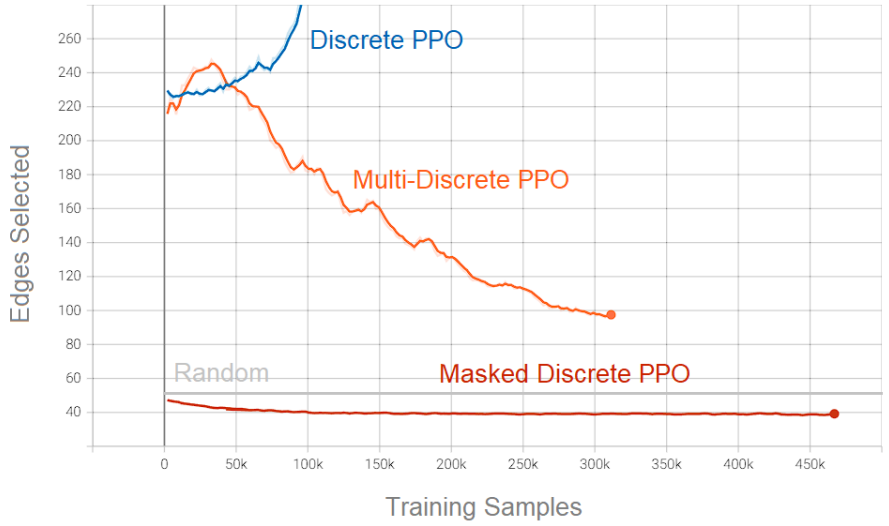
4 Results

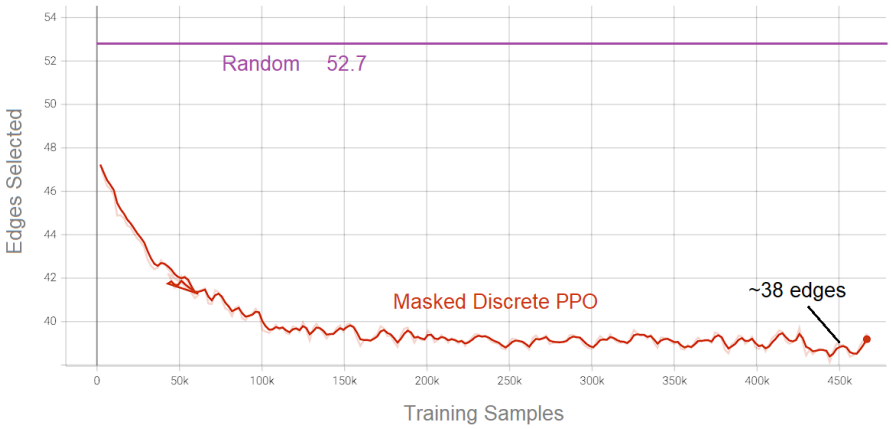
All of the following results are from randomized edgesets and trees with 100 vertices and a density of .5. Note that randomly selecting *valid* edges yields an average solution size of 52.7*edges*, this is also known as the *random algorithm*.



Training without action masks

The performance here is so poor because most of the edges selected by the agent are invalid. The agent takes a significant amount of to learn to select valid edges, which is why masked algorithms are beneficial. As the agents are trained longer, they will perform better. It took 40 minutes of training to generate this data on a system with no GPU. Training periods of around 20 hours yields much better solutions, even for discrete PPO. Note that the discrete PPO environment is not incorrect. It's actually the same environment used for action masked discrete PPO.





It is shown that masked discrete PPO, vastly out-performs the unmasked algorithms because it has far less to learn about solving the TAP.

Ideally the agent's performance would be compared to exact solutions, but I do not have an exact solver as mentioned in section 5.1.

A data-set using random graphs of the same size and density lists their random algorithms performance as 38, and exact as 31. The problem is that their data-set is on different graph types like caterpillars and star-like graphs, which have smaller solutions. This is important because it shows that solutions for graphs of this size are fairly close to the solutions found by the heuristic.

5 Future Work

The heuristic looks promising in its current state, but there are many tests and improvements that can be done. The following are ways that the heuristic could likely be improved or made more well-rounded.

5.1 Exact Solutions

An integer program should be formulated to find exact solutions to the TAP so that the heuristic can be more accurately evaluated. Currently, it can only be concluded that the heuristic performs better than the random algorithm.

5.2 Environment and Observation Space

The observation space is not efficient, and could be larger. The use of adjacency matrices is unsustainable for very large instances of the project. Dictionaries are a natural and efficient way to implement graphs in Python. Furthermore, OpenAI's Baselines supports dictionaries as an observation space.

The observation space could have information beyond which edges are connected together. For example, a list of leaf vertices or the degree of each vertex. There are many possibilities for expanding the observation space

5.3 Pre-Processing

Most of the approximation algorithms for the TAP include pre-processing stages that can determine some of the edges that are in the solution in polynomial time. There are a few pre-processing procedures including leaf-closed maximal matching and identifying solo-leaf edges. The TAP could pre-processed between each action and the number of edges selected added to the reward function. This is very likely to boost the performance of the RL heuristic.

5.4 Action Space

The masked-PPO algorithm was the most performant model evaluated, but it used the discrete action space which is not natural to the problem. In the unmasked version of PPO, the multi-discrete action space performed far better than the discrete one. It is natural to conclude that the same applies for masked-PPO. If the multi-discrete PPO were modified to eliminate specific combinations of it's actions rather than entire actions themselves, the heuristic may be improved. It also may be the case that there is a better way to represent the discrete action space, specifically including the redundant actions.

5.5 Training Structures

The agent was trained on random graphs that were generated in only one way and on one density. In practice the input to the TAP may have all kinds of shapes and densities that aren't well captured by the random generation used. The agent should be trained on many kinds of graphs so that the model is more well-rounded. This could include star graphs, caterpillar graphs, random forests, etc. Also, the density could be set to vary randomly within a certain range, rather than be a fixed value.

5.6 More Results

The graphs shown in the section 4 are for a single graph density and size and was only trained for a couple of hours. More data would allow for higher confidence in the results of the heuristic. There should be longer training times and charts showing performance on cases that weren't directly trained for. The behaviour of the heuristic during unexpected cases is important.

6 Conclusion

The heuristic was successful in defeating the randomized algorithm and certainly learns how to solve the problem completely unsupervised. It is hard to say how effective the heuristic without a comparison to the approximation algorithms or exact solutions. There are several improvements that can be made to the project in its current state as discussed in section 5. I really enjoyed working on this project. I learned a lot about Python and reinforcement learning. Prior to this project, I didn't have much interest in RL but now I'm hooked and will definitely use my new abilities in future projects.

References

[GKZ18] Fabrizio Grandoni, Christos Kalaitzis, and Rico Zenklusen. Improved Approximation for Tree Augmentation: Saving by Rewiring. 2018.

[TLCY20] Cheng-Yen Tang, Chien-Hung Liu, Woei-Kae Chen, and Shingchern D. You. Implementing action mask in proximal policy optimization (PPO) algorithm. *ICT Express*, 6(3):200–203, September 2020.