Deep Q-Learning For The Traveling Salesman Problem

Behzad Karimi, Deanta Kelly, Ian Kessler, Fatima Ododo

May 7, 2024

1 Introduction

This project focuses on the implementation and analysis of the algorithm described in the paper *Learning Combinatorial Optimization Algorithms over Graphs*, which we will reference as the source paper. Although this paper applied their algorithm to various Combinatorial Optimization (CO) problems (Minimum Vertex Cover, Maximum Cut, and Traveling Salesman), we focused on the algorithm applied to only the Traveling Salesman Problem (TSP).

According to the source paper, the machine learning algorithm for the TSP is an attempt to address the issue of classical algorithms "seldom exploit a common trait of real-world optimization problems: instances of the same type of problem are solved again and again on a regular basis, maintaining the same combinatorial structure, but differing mainly in their data." Thus, the issue, in the context of our problem, is if we take two graphs G_1 and G_2 from a distribution of graphs $\mathbb{D}$–graphs that have the same combinatorial structure but differ due to the variance of $\mathbb{D}$–then classical algorithms would solve the TSP for both G_1 and G_2 as if they were independent problems, essentially solving the same type of problem twice.

If we are given a known distribution of graphs $\mathbb{D}$, we would like to learn heuristics on efficiently solving the TSP from graphs in $\mathbb{D}$ that generalize to unseen instances from $\mathbb{D}$. The way we can learn these heuristics is by Deep Q-Learning.

2 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) takes as input of a weighted complete graph, where each pair of distinct vertices (i, j) has one and only one edge between them that has a weight w(i, j). (See CompleteGraph for the implementation of a complete graph.) Furthermore, the type of complete graphs that the source paper and we are focused on are Euclidean graphs. For a Euclidean graph with $V = [0..m-1]$, the vertices are identified as a set of points $[(x_i, y_i) : i \in V]$, where each $(x_i, y_i) \in \mathbb{R}^2$. Then, for $i \neq j$, the weight from i to j is simply $||(x_j, y_j) - (x_i, y_i)||_2$. (See EuclideanGraph for the implementation of an Euclidean graph.)

Let G be a Euclidean graph. Then, let Ψ_G be the set of permutations of G, where each $S \in \Psi_G$ is represented as an ordering of the vertices V such that i is mapped to $S[i]$ for all

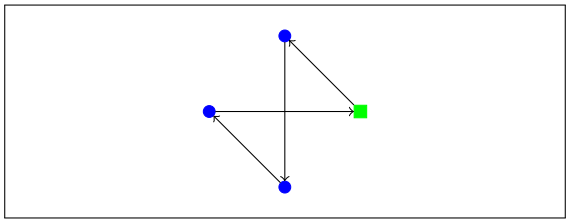
$i \in V$. Now, let $S \in \Psi_G$. Then, the tour distance of S for G is defined as

$$\text{tourDistance}_G(S) = \sum_{i=0}^{|S|-2} w(S[i], S[i+1]) + w(S[|S|-1], S[0]).$$

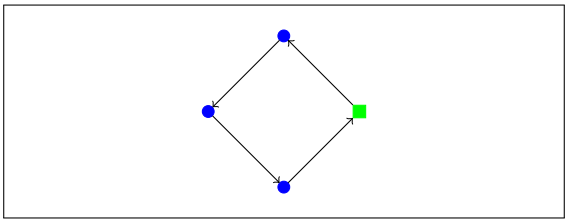
Then, the Traveling Salesman Problem (TSP) can be formulated as finding

$$\text{argmin}_{S \in \Psi_G}(\text{tourDistance}_G(S)).$$

For example, let us consider the Euclidean graph G with points $[(1,0),(0,1),(-1,0),(0,-1)]$. The figure below shows two subfigures, each subfigure showing the same graph but different tours given by different permutations, where the tours start at their respective green nodes.



Permutation is $S = [0,1,3,2]$, yielding $\text{tourDistance}_G(S) = \sqrt{2} + 2 + \sqrt{2} + 2 = 4 + 2\sqrt{2}$.



Permutation is $S = [0,1,2,3]$, yielding $\text{tourDistance}_G(S) = \sqrt{2}+\sqrt{2}+\sqrt{2}+\sqrt{2} = 4\sqrt{2}$, which is the minimal tour distance.

3 Q-Learning

Q-learning is a reinforcement learning technique, for learning the evaluation function Q in the context of optimization problems. The evaluation function is utilized as a model for the state-value function in reinforcement learning. The formulation involves defining states, actions, rewards, and policies within the reinforcement learning framework. Specifically, states represent sequences of actions on a graph, actions correspond to nodes on the graph, rewards quantify changes in a cost function resulting from actions, and policies dictate action selection based on Q. The Q-learning algorithm updates the function approximator's parameters iteratively, utilizing experience replay to enhance sample efficiency. The Q-learning algorithm, as described in page 6 of the source paper (with small changes), is described below :

for episode $e = 1$ to L do

Draw graph G from distribution $\mathbb{D}$

Initialize the state to empty $S_0 = []$

for step t=0 to $m-1$ do

$$v_t = \begin{cases} \text{random node } v \in \overline{S_t}, & \text{w.p. } \epsilon \\ \text{argmax}_{v' \in \overline{S_t}} \hat{Q}_\Theta(S_t, v), & \text{Otherwise} \end{cases}$$

Add v_t to partial solution: $S_{t+1} := S_t + [v_t]$

if $t \geq n$ then

Add tuple $(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t)$ to M

Sample random batch B of size $\min(\beta, |M|)$ from M

Update Θ by SGD from B

end if

end for

end for

return Θ

(See Q-Learning Code to see its implementation.)

The SGD in the algorithm is an abbreviation for Stochastic Gradient Descent. For each sample $(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t)$ in our batch that we perform SGD on, we update Θ by going in the negative direction of the gradient $\nabla_\Theta J(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t; \Theta)$, where

$$J(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t; \Theta) = \frac{1}{2}(R_{t-n,t} + \gamma \max_{v' \in \overline{S_t}} \hat{Q}_\Theta(S_t, v') - \hat{Q}_\Theta(S_{t-n}, v_{t-n}))^2.$$

(See Gradient Calculation to see the math used to calculate the gradients.) For Q-Learning, for each episode, the n is the number of steps we wait before we do Q Learning. Furthermore, $\gamma \in (0, 1]$ is the discount factor, which tells us how much influence future Q values have on current Q values. We use $\hat{Q}_\Theta$ rather than Q in the J formula to note that the hat notation tells us that the Q function used is an approximation to the ideal optimal Q function, Q^*, and to note that this approximation is determined by Θ, as described in the next section.

We define

- The Q-Learning Cost Function: $c_G(S) = -\text{tourDistance}_G(S)$
- The Reward of Action v at State S: $r(S, v) = c_G(S + [v]) - c_G(S)$

- Rewards Gained From State $(t-n)$ to State t: $R_{t-n,t} = \sum_{i=0}^{n-1} r(S_{t-n+i}, v_{t-n+i})$

Thus, we want, for our final solution S_m, to maximize c_G. Furthermore, we see that

$$R_{t-n,t} = \sum_{i=0}^{n-1} r(S_{t-n+i}, v_{t-n+i}) = \sum_{i=0}^{n-1} c_G(S_{t-n+1+i}) - c_G(S_{t-n+i}) = c_G(S_t) - c_G(S_{t-n})$$

By performing gradient descent with gradient $\nabla_\Theta J(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t; \Theta)$, we are wanting $(R_{t-n,t} + \gamma \max_{v' \in \overline{S_t}} \hat{Q}_\Theta(S_t, v') - \hat{Q}_\Theta(S_{t-n}, v_{t-n}))$ to move closer to zero, meaning we want $\hat{Q}_\Theta(S_{t-n}, v_{t-n})$ to move closer to $(R_{t-n,t} + \gamma \max_{v' \in \overline{S_t}} \hat{Q}_\Theta(S_t, v'))$. Thus, our aim is for the Q-learning to make $\hat{Q}_\Theta(S_t, v_t) \approx R_{t,m} = c_G(S_m) - c_G(S_t)$ for each time step $t \in [0..(m-1)]$. In other words, at each time step, for our current state S_t, we want $\hat{Q}_\Theta(S_t, v') + c_G(S_t)$ to reflect the best final reward, $c_G(S_m)$ we could get after choosing action v' at state S_t. Therefore, because of our Q Learning, at each time step t, we pick action $v^* = \text{argmax}_{v' \in \overline{S_t}} \hat{Q}_\Theta(S_t, v')$ in order to hopefully get S_m at the end that maximizes c_G or, equivalently, minimizes tourDistance_G.

4 Evaluating Q With Structure2Vec Neural Network

Deep Q-Learning is a type of Q-Learning that uses a neural network to approximate the Q function. The neural network architecture used for our Deep Q-Learning algorithm is known as *structure2vec*. This architecture uses an embedding that embeds each node $v \in V$ of our graph into a p-dimensional vector $(\mu)_v \in \mathbb{R}^{p \times 1}$, where p is a hyperparameter to tune.

To be more precise, this embedding gives each node a vector representation in $\mathbb{R}^{p \times 1}$ in each layer of our neural network. Furthermore, since this neural network is used to approximate our Q function, it should depend on our current partial solution S.

Thus, for a given partial solution S, our p-dimensional embedding for each node $v \in V$ and each layer i is labeled by $(\mu_S^{(i)})_v \in \mathbb{R}^{p \times 1}$. Having $m = |V|$, the i^{th} layer of our neural network is

$$\mu_S^{(i)} = [(\mu_S^{(i)})_0, (\mu_S^{(i)})_1, ..., (\mu_S^{(i)})_{m-1}] \in \mathbb{R}^{p \times m}.$$

In computing each embedding of node v in each hidden layer of our neural network for the overall purpose of finding the next best node to travel to from our current partial solution S, we want our embeddings to depend on:

- Current Partial Solution S
- The Graph Structure From v
  - The neighbors of v, labeled as $\mathscr{N}(v)$
  - The weight of edge (v, u), labeled as $w(v, u)$, for each $u \in \mathscr{N}(v)$.

To have our embedding depend on S, we used the binary vector

$$x_S := [1\{v \in S\} : v \in V] \in \{0,1\}^{1\times m}.$$

Our input layer of our neural network is a matrix of all zeros: $\mu_S^{(0)} := 0_{p\times m}$. Then, going from one layer to next, the embedding of node v is found by

$$(\mu_S^{(i+1)})_v \leftarrow \text{relu}(\theta_1 x_S[v] + \theta_2 \sum_{u\in\mathscr{N}(v)} (\mu_S^{(i)})_u + \theta_3 \sum_{u\in\mathscr{N}(v)} \text{relu}(\theta_4 w(v,u))).$$

Thus, for a hidden layer $\mu_S^{(i)}$, we compute the next hidden layer $\mu_S^{(i+1)}$ until $i = T$, where T is the number of hidden layers we use, which is a hyperparameter to tune. Thus, from the input layer $\mu_S^{(0)}$, we compute a sequence of T hidden layers $\mu_S^{(1)}, \mu_S^{(2)}, \ldots, \mu_S^{(T)}$, where $\mu_S^{(T)}$ is our final hidden layer.

With our given partial solution S, after we compute our final hidden layer, $\mu_S^{(T)}$, we can calculate the value of Q at each node $v \in V$:

$$\hat{Q}_\Theta(S,v) = \theta_{5a}^\intercal \text{relu}(\theta_6 \sum_{u\in V} (\mu_S^{(T)})_u) + \theta_{5b}^\intercal \text{relu}(\theta_7 (\mu_S^{(T)})_v).$$

Letting $\overline{S} := V\backslash S$, we update $S \leftarrow S + [v^*]$, where

$$v^* = \text{argmax}_{v\in\overline{S}} \hat{Q}(S,v).$$

(See Structure2Vec Architecture code for more details.)

We see that $\hat{Q}(S,v)$ depends on $\Theta = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_{5a}, \theta_{5b}, \theta_6, \theta_7]$. These parameters Θ are learned through Q-Learning, as described in the previous section. (See Theta for the implementation of the Θ object.)

5 Data Analysis

4-Fold Cross-Validation

To calculate the error of a Deep Q-Learning model, we performed 4-Fold Cross-Validation from eight random graphs from a distribution $\mathbb{D}$, such that Euclidean graphs G from $\mathbb{D}$ had a vertex set V with properties

- $|V| = 9$
- $V \subseteq \mathbb{Z}^2 \cap [-5,5]^2 \equiv [-5..5]^2$.

(See Euclidean Graph Distribution code to see implementation of $\mathbb{D}$. See Walked Graphs Dataframe for Cross-Validation to see the graphs used for cross-validation.) To calculate the error of a model, we calculated the error of each fold and recorded the average fold error. Calculating the error of each fold required two phases: the training phase and the test phase. For the training phase, we performed Q-Learning from the 6 graphs in our training set to learn Θ. For the test phase, we used our calculated Θ for our approximated Q function, which was used to predict the solutions for the 2 graphs in the test set. Then, we calculated the error of each prediction and recorded the average error for the fold.

Approximation Ratio as Error

For a full solution S for Euclidean graph G, let

$$\text{cost}_G(S) = \text{tourDistance}_G(S) \geq 0,$$

as described in the "The Traveling Salesman Problem" section. For graph G, let $\hat{S}_\Theta$ be a full solution for G found using our model with weights Θ and S^* be a full solution that minimizes cost_G. Then the error for our prediction $\hat{S}_\Theta$ is the approximation ratio:

$$\rho = \frac{\text{cost}_G(\hat{S}_\Theta)}{\text{cost}_G(S^*)} \geq 1.$$

This is the error we calculated for each graph in the test set for each fold. Optimal solutions S^* were found using the Held-Karp algorithm, described in the paper *Boosting Dynamic Programming with Neural Networks for Solving NP-hard Problems*. (See Held-Karp Algorithm to see the implementation of the algorithm used to find optimal solutions.)

The graphs used had a stored optimal solution, both the optimal order of nodes to walk and the total distance to walk the cycle generated by that ordering. (See Walked Graphs for the implementation to create graphs with stored solution.)

Tuning Hyperparameters

Each Deep Q-Learning model is uniquely defined by seven hyperparameters. Among the models used in cross-validation, two values were used for each hyperparameter. Thus, error was calculated for $2^7 = 128$ models, each created by choosing between 2 values for each of the 7 hyperparameters. The following are the hyperparameters used, along with the values used for each hyperparameter among models used in cross-validation:

- Dimension of Each Node Embedding: $p \in \{3, 4\}$
- Number of Hidden Layers: $T \in \{1, 2\}$
- Probability of Choosing Random $v \in \overline{S}$ to Append to Partial Solution S: $\epsilon \in \{0.01, 0.05\}$
- Number of Steps Between States for n-Step Q-Learning: $n \in \{2, 3\}$
- Learning Rate for Gradient Descent: $\alpha \in \{0.01, 0.1\}$
- Maximum Size of Batches for Mini-Batch Gradient Descent: $\beta \in \{5, 10\}$
- Discount Factor for Q-Learning: $\gamma \in \{0.9, 1\}$

A error dataframe was created to store each model along with its assignment of hyperparameters and its average error in cross-validation. (See Analysis 1 to see the implementation of creating this error dataframe.)

Average Error Grouped By Hyperparameter To Find "Best" Model

For each hyperparameter, we grouped the error data by the hyperparameter's values and averaged the approximation ratio. We created bar graphs to display the results.

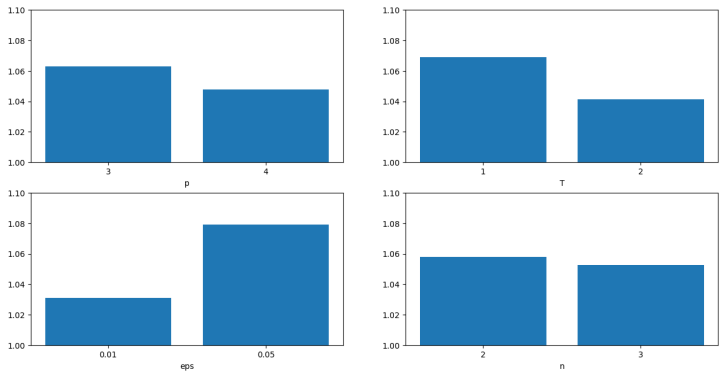


Figure 1: Average Approximation Ratio For p, T, ϵ, and n

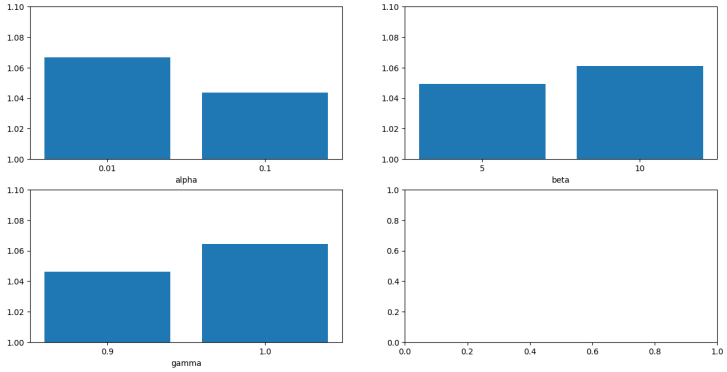


Figure 2: Average Approximation Ratio For α, β, and γ

.

For each hyperparameter, we see which value yielded the lower approximation ratio in its respective graph. According to the graphs, it seems that the best values for the hyperparameters are $p = 4$, $T = 2$, $\epsilon = 0.01$, $n = 3$, $\alpha = 0.1$, $\beta = 5$, and $\gamma = 0.9$. Furthermore, this assignment of values yielded an approxmation ratio of $\rho = 1$. Although this assignment was not unique in its perfect approximation ratio, we will consider this model to be our "best" model for the remaining analysis. Of the 128 models used in cross-validation, 39 of them had an approximation ratio of $\rho = 1$. We stored these models in a dataframe. (See Analysis 2 to see implementation to create both the bar charts and the dataframe of "$\rho = 1$" models. See Models With $\rho = 1$ dataframe to see these 39 models.)

Q-Learning With "Best" Model Over 3 Episodes

Using our "best" model found previously, we performed Q-Learning over 3 episodes with other distributions and created plots to visualize the graphs, their optimal paths, and their approximated paths. Let $\mathbb{D}_m$ be the distribution of graphs with vertex set V such that $V \subseteq [-5..5]^2$ as before, but $|V| = m$. We performed Q-Learning over 3 episodes with distributions $\mathbb{D}_9$, $\mathbb{D}_{14}$, and $\mathbb{D}_{17}$. (See Analysis 3 to see implementation that created the following plots.)

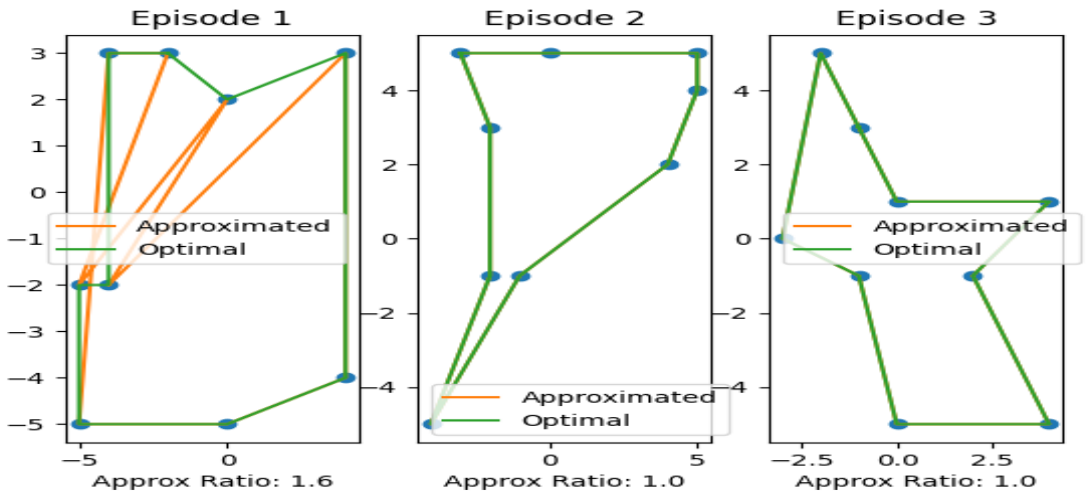


Figure 3: Q-Learning Over 3 Episodes (With Graphs From $\mathbb{D}_9$)

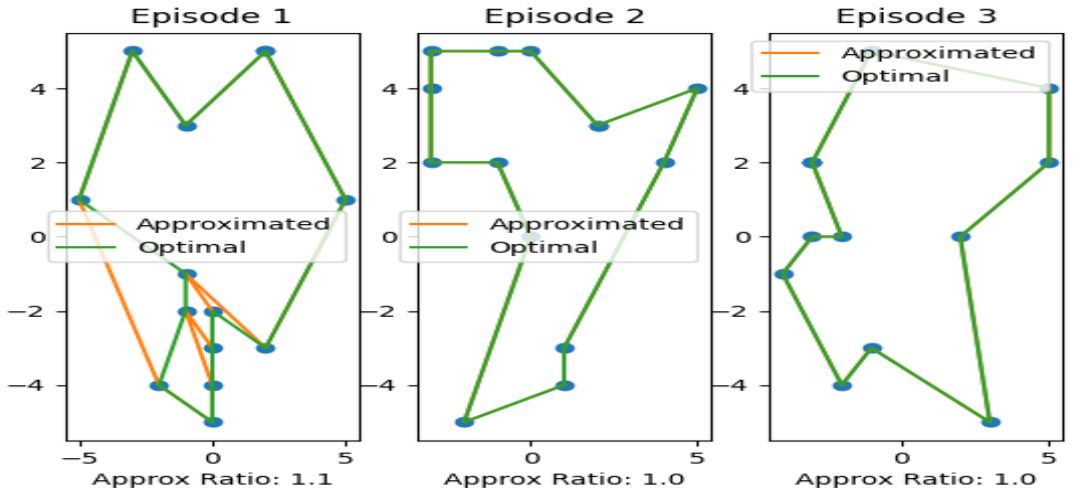


Figure 4: Q-Learning Over 3 Episodes (With Graphs From $\mathbb{D}_{14}$)

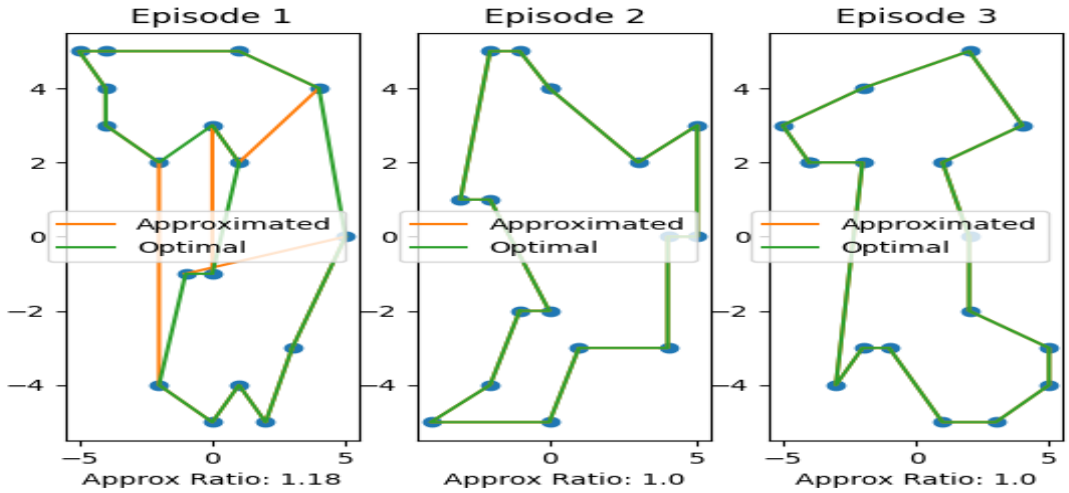


Figure 5: Q-Learning Over 3 Episodes (With Graphs From $\mathbb{D}_{17}$)

6 Conclusion

In cross-validation, the model with the hyperparameter assignment $p = 4$, $T = 2$, $\epsilon = 0.01$, $n = 3$, $\alpha = 0.1$, $\beta = 5$, and $\gamma = 0.9$ yielded an approximation ratio of $\rho = 1$, along with 38 other models. We consider this model our "best" model.

Using this model, we looked at its performance of Q-Learning over 3 episodes with graphs from distributions $\mathbb{D}_9$, $\mathbb{D}_{14}$, and $\mathbb{D}_{17}$. For each sample, we see, as expected, the Q function does not generate the optimal path in the first episode. However, for each sample, the Q function generated the optimal path in the second and third episode, suggesting that the Q function learned how to travel similar graphs after learning from the first graph and possibly learning from subsequent graphs.

For future work, it would be good to consider more complicated distributions of Euclidean graphs. More complicated distributions may be distributions with graphs that have vertex set V such that $V \subset \mathbb{R}^2$, not just in $\mathbb{Z}^2$. Also, distributions may be more complicated using vertex sets V that are in higher dimensions (e.g. $\mathbb{R}^3$). We may also consider distributions of graphs that are non-Euclidean. We could also see how this Q-Learning model works for the other CO applications described in the source paper, such as Minimum Vertex Cover and Maximum Cut.

References

[1] **Focus Of Project:**

(Referenced in Introduction.)

Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, Le Song

Learning Combinatorial Optimization Algorithms over Graphs

Neural Processing Information Systems 5 April, 2017

[2] **For The Held-Karp Algorithm:**

(Referenced in Approximation Ratio as Error.)

Feidiao Yang, Tiancheng Jin, Tie-Yan Liu, Xiaoming Sun, Jialing Zhang

Boosting Dynamic Programming with Neural Networks for Solving NP-hard Problems

Proceedings of Machine Learning Research 95:726-739, 2018

7 Appendix

Code

Complete Graph

(Referenced in The Traveling Salesman Problem.)

```
from itertools import product
import numpy as np
from functools import reduce
import random

class CompleteGraph:
    def __init__(self, dist_matrix):
        '''
        :param dist_matrix: n by n matrix w where w[(i, j)] is the weight of the
            edge (i, j) and where w[(i, i)] = 0
        '''
        m, n = dist_matrix.shape
        if m != n:
            raise AttributeError("Matrix␣is␣Not␣Square")
        self.n = n
        self.vertices = list(range(self.n))
        self.edges = list(product(range(self.n)))
        self.w = (np.ones((n, n)) - np.eye(n)) * dist_matrix
        self.W = self.w.flatten()
        self.neighbors = dict([(i, set(self.filter(self.vertices, lambda j: j != i
            ))) for i in self.vertices])
        self.N = self.neighbor_matrix()
        self.U = self.getU()


    def closeWalk(self, w):
        '''
        :param w: a list of vertices to walk
        :return: if the beginning vertex is not the end vertex, the walk w with
            the beginning vertex appended to the end
        '''
        return w + [w[0]] if len(w) > 0 and w[-1] != w[0] else w

    def randomHamCycle(self):
        '''
        :return: a random list of vertices that represent a Hamiltonian cycle of
            the graph
        '''
```

```
    return self.closeWalk(random.sample(range(self.n), k = self.n))

def walkDistance(self, walk):
    '''
    :param walk: a list of vertices to walk
    :return: the sum of the weights of the edges to walk that corresponds to
        walking the given list of vertices
    '''
    def recurse(distance, current_v, toWalk):
        if len(toWalk) == 0:
            return distance
        else:
            next_v, remaining = toWalk[0], toWalk[1:]
            return recurse(distance + self.w[(current_v, next_v)], next_v,
                remaining)
    return 0 if len(walk) == 0 else recurse(0, walk[0], walk[1:])

def tourDistance(self, S):
    '''
    :param S: a list of vertices that represents a full solution for Q
        Learning
    :return: the total walk distance of the tour generated by S
    '''
    return self.walkDistance(self.closeWalk(S))

def filter(self, s, predicate):
    '''
    :param s: list of vertices
    :param predicate: a function that takes a vertex and returns a boolean
    :return: a sublist of s that is filtered by the predicate
    '''
    return reduce(lambda v, i: v + [i] if predicate(i) else v, s, [])

def neighbor_vec(self, v):
    '''
    :param v: vertex of graph
    :return: a binary vector that gives a 1 at index u if u is a neighbor of v
         and a 0 otherwise
    '''
    neighbors = self.neighbors[v]
    return np.vectorize(lambda i: int(i in neighbors))(self.vertices).reshape
        (-1, 1)

def neighbor_matrix(self):
    '''
```

```
    :return: square matrix where the vth column is the neighbor_vec of v
    '''
    return np.concatenate([self.neighbor_vec(v) for v in self.vertices], axis
        = 1)

def unit_vec(self, v):
    '''
    :param v: vertex of graph
    :return: a unit vector of the dimension of the number of vertices in the
        direction of vertex v
    '''
    return np.vectorize(lambda i: int(i == v))(self.vertices).reshape(-1, 1)

def neighbor_square(self, v):
    '''
    :param v: vertex of graph
    :return: square matrix where the vth column is the neighbor_vec of v and
        zeros elsewhere
    '''
    return np.outer(self.neighbor_vec(v), self.unit_vec(v).reshape(1, -1))

def getU(self):
    '''
    :return: a concatenation of each vertex neighbor_square along the 0th axis
    '''
    return np.concatenate([self.neighbor_square(v) for v in self.vertices],
        axis=0)
```

Euclidean Graph

(Referenced in The Traveling Salesman Problem.)

```
from CompleteGraph import CompleteGraph
import numpy as np
from TSP_HK import TSP_HK

class EuclideanGraph(CompleteGraph):
    def __init__(self, points, shortest_cycle = None, distance = None):
        '''
        :param points: the two-dimensional points to use for the graph
        :param shortest_cycle: an optimal permutation of vertices for the TSP
        :param distance: the minimal walk distance for the TSP
        '''
        self.points = points
        self.shortest_cycle = shortest_cycle
        self.distance = distance
        pt_array = np.array([list(point) for point in points])
        dist = lambda i, j: np.linalg.norm(pt_array[i, :] - pt_array[j, :])
        n = len(pt_array)
        dist_matrix = np.array([[dist(i, j) for j in range(n)] for i in range(n)])
        super().__init__(dist_matrix)

    def __repr__(self):
        return "Graph{}".format(set(self.points))

    def __str__(self):
        return "Graph{}".format(set(self.points))

    def thisWithShortestCycle(self):
        '''
        :return: the same graph with calculated best permutation and walk distance
            for the TSP
        '''
        S = TSP_HK().calculateWalk(self)
        d = self.walkDistance(self.closeWalk(S))
        return EuclideanGraph(self.points, shortest_cycle = S, distance = d)
```

Q-Learning Code

(Referenced in Q-Learning.)

```
    def R(self, C, t, n):
        '''
        :param C: vector of accumulated costs
        :param t: current time index
        :param n: number of steps into the future
        :return: change in rewards from current time to n steps in the future
        '''
        return C[t + n - 1] - C[t - 1]

    def dRelu_dz(self, z):
        '''
        :param z: argument vector of relu function
        :return: matrix that is the derivative of relu(z) with respect to z
        '''
        return np.diag(np.vectorize(lambda i: int(i > 0))(z)[:, 0])

    def dthetafunc_dtheta_1(self, r, Theta, G, S, i, v):
        '''
        :param r: horizontal vector that multiplies to derivative numerator
        :param Theta: object consisting of list of theta weights
        :param G: graph
        :param S: list of unique vertices of graph
        :param i: index of Theta list for the theta that is used for the
            derivative denominator
        :param v: vector column used in mu expression
        :return: vector or matrix representing a derivative
        '''
        if i in {0}:
            return r @ (self.x(G, S)[v] * np.eye(Theta.p))
        elif i in {1, 2}:
            return np.zeros((Theta.p, Theta.p))
        elif i in {3}:
            return np.zeros((1, Theta.p))

    def dthetafunc_dtheta_2(self, r, Theta, G, S, i, mu_list, v):
        '''
        :param r: horizontal vector that multiplies to derivative numerator
        :param Theta: object consisting of list of theta weights
        :param G: graph
        :param S: list of unique vertices of graph
        :param i: index of Theta list for the theta that is used for the
            derivative denominator
```

```
    :param mu_list: list of mu matrices where its last entry is the mu used in
        expression
    :param v: vector column used in mu expression
    :return: vector or matrix representing a derivative
    '''
    r_new = r @ Theta.theta_2
    neighbor_dmu = lambda u: self.dmu_dtheta(r_new, Theta, G, S, i, mu_list
        [:-1], u)
    if i in {0, 2, 3}:
        return sum([neighbor_dmu(u) for u in G.neighbors[v]])
    elif i in {1}:
        neighbor_outerproduct = lambda u: np.outer(mu_list[-1] @ G.unit_vec(u)
            , r)
        return sum([neighbor_dmu(u) + neighbor_outerproduct(u) for u in G.
            neighbors[v]])

def dthetafunc_dtheta_3(self, r, Theta, G, i, v):
    '''
    :param r: horizontal vector that multiplies to derivative numerator
    :param Theta: object consisting of list of theta weights
    :param G: graph
    :param i: index of Theta list for the theta that is used for the
        derivative denominator
    :param v: vector column used in mu expression
    :return: vector or matrix representing a derivative
    '''
    if i in {0}:
        return np.zeros((1, Theta.p))
    elif i in {1}:
        return np.zeros((Theta.p, Theta.p))
    elif i in {2}:
        return np.outer(self.relu(np.outer(Theta.theta_4, G.W)) @ G.U @ G.
            unit_vec(v), r)
    elif i in {3}:
        return (r @ Theta.theta_3 @ sum([self.dRelu_dz(Theta.theta_4 * G.w[(v,
             u)]) @
                                         (G.w[(v, u)] * np.eye(Theta.p, Theta.p)) for
                                             u in G.neighbors[v]]))

def dmu_dtheta(self, r, Theta, G, S, i, mu_list, v):
    '''
    :param r: horizontal vector that multiplies to derivative numerator
    :param Theta: object consisting of list of theta weights
    :param G: graph
    :param S: list of unique vertices of graph
```

```
    :param i: index of Theta list for the theta that is used for the
        derivative denominator
    :param mu_list: list of mu matrices where its last entry is the mu to
        calculate the current mu
    :param v: vector column used in mu expression
    :return: vector or matrix representing a derivative
    '''
    if len(mu_list) == 0:
        if i in {0, 3}:
            return np.zeros((1, Theta.p))
        elif i in {1, 2}:
            return np.zeros((Theta.p, Theta.p))
    else:
        z = (Theta.theta_1 * self.x(G, S)[v] + Theta.theta_2 @ mu_list[-1] @ G
            .N[:, v] +
        Theta.theta_3 @ self.relu(np.outer(Theta.theta_4, G.W)) @ G.U[:, v])
        r_new = r @ self.dRelu_dz(z)
        r_1 = self.dthetafunc_dtheta_1(r_new, Theta, G, S, i, v)
        r_2 = self.dthetafunc_dtheta_2(r_new, Theta, G, S, i, mu_list, v)
        r_3 = self.dthetafunc_dtheta_3(r_new, Theta, G, i, v)
        return r_1 + r_2 + r_3

def dQ_dtheta(self, Theta, G, S, v, i):
    '''
    :param Theta: object consisting of list of theta weights
    :param G: graph
    :param S: list of unique vertices of graph
    :param i: index of Theta list for the theta that is used for the
        derivative denominator
    :param v: column used in Q vector
    :return: vector or matrix representing a derivative
    '''
    Q_vec, mu_list = self.Q_vec_mu_list(Theta, G, S)
    ra_1 = Theta.theta_6 @ mu_list[-1] @ np.ones((G.n, 1))
    rb_1 = Theta.theta_7 @ mu_list[-1] @ G.unit_vec(v)
    if i in {0, 1, 2, 3}:
        ra_2 = Theta.theta_5a.reshape(1, -1) @ self.dRelu_dz(ra_1) @ Theta.
            theta_6
        ra = sum([self.dmu_dtheta(ra_2, Theta, G, S, i, mu_list[:-1], u) for u
             in G.vertices])
        rb_2 = Theta.theta_5b.reshape(1, -1) @ self.dRelu_dz(rb_1) @ Theta.
            theta_7
        rb = self.dmu_dtheta(rb_2, Theta, G, S, i, mu_list[:-1], v)
        return ra + rb
    elif i == 4:
```

```
            return self.relu(ra_1).reshape(1, -1)
        elif i == 5:
            return self.relu(rb_1).reshape(1, -1)
        elif i == 6:
            return np.outer(mu_list[-1] @ np.ones((G.n, 1)), Theta.theta_5a.
                reshape(1, -1) @ self.dRelu_dz(ra_1))
        elif i == 7:
            return np.outer(mu_list[-1] @ G.unit_vec(v), Theta.theta_5b.reshape(1,
                 -1) @ self.dRelu_dz(rb_1))

    def dQBest_dtheta(self, Theta, G, S, i):
        '''
        :param Theta: object consisting of list of theta weights
        :param G: graph
        :param S: list of unique vertices of graph
        :param i: index of Theta list for the theta that is used for the
            derivative denominator
        :return: vector or matrix representing a derivative
        '''
        S_not = self.S_not(G, S)
        if len(S_not) == 0:
            return 0
        else:
            (v, Q), mu_list = self.policy(Theta, G, S, S_not)
            return self.dQ_dtheta(Theta, G, S, v, i)

    def dz_dtheta(self, Theta, G, S_past, v_past, S, i):
        '''
        :param Theta: object consisting of list of theta weights
        :param G: graph
        :param S_past: list of unique vertices of graph n steps in the past
        :param v_past: vertex appended to S_past n steps in the past
        :param S: current list of unique vertices of graph
        :param i: index of Theta list for the theta that is used for the
            derivative denominator
        :return: vector or matrix representing a derivative
        '''
        return self.gamma * self.dQBest_dtheta(Theta, G, S, i) - self.dQ_dtheta(
            Theta, G, S_past, v_past, i)

    def dJ_dtheta(self, Theta, G, S_past, v_past, R_diff, S, i):
        '''
        :param Theta: object consisting of list of theta weights
        :param G: graph
        :param S_past: list of unique vertices of graph n steps in the past
```

```
    :param v_past: vertex appended to S_past n steps in the past
    :param R_diff: change in rewards from the past to the present
    :param S: current list of unique vertices of graph
    :param i: index of Theta list for the theta that is used for the
        derivative denominator
    :return: vector or matrix representing a derivative
    '''
    z = R_diff + self.gamma * self.QBest(Theta, G, S) - self.Q_vec_mu_list(
        Theta, G, S_past)[0][v_past]
    return z * self.dz_dtheta(Theta, G, S_past, v_past, S, i)

def updated_S(self, Theta, G, exploreOption = True):
    '''
    :param Theta: object consisting of list of theta weights
    :param G: graph
    :return: function that updates vertex list S
    '''
    def f(S):
        S_not = self.S_not(G, S)
        explore = exploreOption and (np.random.rand() < self.eps)
        v = np.random.choice(list(S_not)) if explore else self.vBest(Theta, G,
             S, S_not)
        return S + [v]
    return f

def updated_C(self, G, S):
    '''
    :param G: graph
    :param S: list of unique vertices of graph
    :return: function that updates cost list C
    '''
    def f(C):
        t = len(C)
        c_prev = 0 if len(C) == 0 else C[-1]
        c = c_prev + G.w[(S[t-1], S[0])] - G.w[(S[t-1], S[t])] - G.w[(S[t], S
            [0])]
        return C + [c]
    return f

def updated_M(self, G, S, C, t):
    '''
    :param G: graph
    :param S: list of unique vertices of graph
    :param C: list of accumulated costs S
    :param t: current time index
```

```
    :return: function that updates memory list M
    '''
    def f(M):
        return M + [(G, S[:(t-self.n)], S[t-self.n], self.R(C, t-self.n, self.
            n), S)] if t >= self.n else M
    return f

def updated_theta(self, Theta, B):
    '''
    :param Theta: object consisting of list of theta weights
    :param B: random batch from memory list
    :return: function that takes theta index and returns updated theta for
        that index
    '''
    def f(i):
        if i in {0, 1, 2, 3, 4, 5, 6, 7}:
            batch_gradient = 1 / len(B) * sum([self.dJ_dtheta(Theta, *b, i).T
                for b in B])
            return Theta.thetas[i] - self.alpha * batch_gradient
        else:
            return Theta.thetas[i]
    return f

def updated_Theta(self, M, t):
    '''
    :param M: memory list
    :param t: current time index
    :return: function that updates object Theta
    '''
    def f(Theta):
        if t >= self.n:
            r = min(len(M), self.beta)
            B = random.sample(M, r)
            theta_func = self.updated_theta(Theta, B)
            thetas_new = [theta_func(i) for i in range(8)]
            return ThetaObject(thetas_new)
        else:
            return Theta
    return f

def episode(self, Theta, M, G):
    '''
    :param Theta: object consisting of list of theta weights
    :param M: memory list
    :param G: graph
```

```
        :return: (Theta_new, M_new, S) where (Theta_new, M_new) are updated and S
            is the permutation used for graph G
        '''
        def QLearn_recurse(S, C, M, Theta, t):

            #print(50 * "=")
            #print("S: {}".format(S))
            #print("C: {}".format(C))
            #print("M: {}".format(M))
            #print("Theta:\n{}".format(Theta))
            #print("t: {}".format(t))

            if t == G.n:
                return Theta, M, S
            else:
                S_new = self.updated_S(Theta, G)(S)
                C_new = self.updated_C(G, S_new)(C)
                t = len(S)
                M_new = self.updated_M(G, S, C, t)(M)
                Theta_new = self.updated_Theta(M_new, t)(Theta)
                return QLearn_recurse(S_new, C_new, M_new, Theta_new, t + 1)

        return QLearn_recurse([], [], M, Theta, 0)

    def QLearning(self, Gs):
        '''
        :param Gs: list of graphs to use
        :return: (G_S_list, Theta) where G_S_list is a list of (G, S) for graph G
            and its corresponding
                 permutation S to use and Theta is the final list of weights after
                    Q Learning
        '''
        def appendEpisodeResults(results, G):
            G_S_list, (Theta, M) = results
            Theta_new, M_new, S = self.episode(Theta, M, G)
            return G_S_list + [(G, S)], (Theta_new, M_new)

        G_S_list_final, (Theta_final, _) = reduce(appendEpisodeResults, Gs, ([], (
            RandomThetaObject(self.p), [])))
        return G_S_list_final, Theta_final
```

Structure2Vec Architecture

(Referenced in Evaluating Q With Structure2Vec Neural Network.)

```
def x(self, G, S):
    '''
    :param G: graph
    :param S: list of unique vertices of graph
    :return: vector of binary values showing which vertices in G are in S
    '''
    return np.vectorize(lambda i: int(i in S))(G.vertices)

def relu(self, v):
    '''
    :param v: vector
    :return: vector showing the element-wise relu values of v
    '''
    return np.vectorize(lambda i: max(0, i))(v)

def F(self, Theta, G, S):
    '''
    :param Theta: object consisting of list of theta weights
    :param G: graph
    :param S: list of unique vertices of graph
    :return: function that takes a list of mu's and returns that list with an
        appended next mu
    '''
    x = self.x(G, S)
    def new_mu_list(mu_list):
        mu = mu_list[-1]
        r1 = np.outer(Theta.theta_1, x)
        r2 = Theta.theta_2 @ mu @ G.N
        r3 = Theta.theta_3 @ self.relu(np.outer(Theta.theta_4, G.W)) @ G.U
        return mu_list + [self.relu(r1 + r2 + r3)]
    return new_mu_list

def mu_list_final(self, Theta, G, S):
    '''
    :param Theta: object consisting of list of theta weights
    :param G: graph
    :param S: list of unique vertices of graph
    :return: final list of mu matrices
    '''
    F_func = self.F(Theta, G, S)
    mu_recurse = lambda mu_list, t: mu_list if t == self.T else mu_recurse(
        F_func(mu_list), t + 1)
```

```
    return mu_recurse([np.zeros((Theta.p, G.n))], 0)

def Q_vec_mu_list(self, Theta, G, S):
    '''
    :param Theta: object consisting of list of theta weights
    :param G: graph
    :param S: list of unique vertices of graph
    :return: a tuple of the Q vector for S and the mu list used to calculate Q
        vector
    '''
    mu_list = self.mu_list_final(Theta, G, S)
    mu = mu_list[-1]
    r1 = Theta.theta_5a.reshape(1, -1) @ self.relu(Theta.theta_6 @ mu @ np.
       ones((G.n, G.n)))
    r2 = Theta.theta_5b.reshape(1, -1) @ self.relu(Theta.theta_7 @ mu)
    return (r1 + r2)[0], mu_list

def S_not(self, G, S):
    '''
    :param G: graph
    :param S: list of unique vertices of graph
    :return: set of vertices in G but not in S
    '''
    return set(G.vertices).difference(S)

def policy(self, Theta, G, S, S_not = None):
    '''
    :param Theta: object consisting of list of theta weights
    :param G: graph
    :param S: list of unique vertices of graph
    :param S_not: (optional) set of vertices in G but not in S (to speed up
       calculation)
    :return: ((v, Q), mu_list) where (v, Q) is the (argmax, max) of the Q
       vector and mu_list was used for Q vector
    '''
    S_not = self.S_not(G, S) if S_not is None else S_not
    Qvec, mu_list = self.Q_vec_mu_list(Theta, G, S)
    vQs = [(v, Qvec[v]) for v in S_not]
    return reduce(lambda t1, t2: t2 if t1[0] is None or t2[1] > t1[1] else t1,
        vQs, (None, 0)), mu_list

def vBest(self, Theta, G, S, S_not = None):
    '''
    :param Theta: object consisting of list of theta weights
    :param G: graph
```

```
        :param S: list of unique vertices of graph
        :param S_not: (optional) set of vertices in G but not in S (to speed up
            calculation)
        :return: the best vertex to append to S if S_not is nonempty else None
        '''
        return self.policy(Theta, G, S, S_not)[0][0]

    def QBest(self, Theta, G, S, S_not = None):
        '''
        :param Theta: object consisting of list of theta weights
        :param G: graph
        :param S: list of unique vertices of graph
        :param S_not: (optional) set of vertices in G but not in S (to speed up
            calculation)
        :return: Q value of the best vertex to append to S if S_not is nonempty
            else 0
        '''
        return self.policy(Theta, G, S, S_not)[0][1]
```

Theta

(Referenced in Evaluating Q With Structure2Vec Neural Network.)

```
import numpy as np

class ThetaObject:
    def __init__(self, thetas):
        '''
        :param thetas: list of theta vectors/matrices to use
        '''
        self.thetas = thetas
        self.theta_1, self.theta_2, self.theta_3, self.theta_4 = thetas[:4]
        self.theta_5a, self.theta_5b, self.theta_6, self.theta_7 = thetas[4:]
        self.p = self.theta_1.shape[0]
        self.names = ["theta_1", "theta_2", "theta_3", "theta_4", "theta_5a", "
            theta_5b", "theta_6", "theta_7"]

    def __str__(self):
        r = "\n".join([20*"-" + "\n{}\n{}".format(name, theta) for (name, theta)
            in zip(self.names, self.thetas)])
        return r

    def __repr__(self):
        r = "\n".join([20*"-" + "\n{}\n{}".format(name, theta) for (name, theta)
            in zip(self.names, self.thetas)])
        return r
```

```
class RandomThetaObject (ThetaObject):
    def __init__(self, p):
        '''
        :param p: the dimension to use for the theta weights
        '''
        theta_1 = np.random.rand(p, 1)
        theta_2 = np.random.rand(p, p)
        theta_3 = np.random.rand(p, p)
        theta_4 = np.random.rand(p, 1)
        theta_5a = np.random.rand(p, 1)
        theta_5b = np.random.rand(p, 1)
        theta_6 = np.random.rand(p, p)
        theta_7 = np.random.rand(p, p)
        thetas = [theta_1, theta_2, theta_3, theta_4, theta_5a, theta_5b, theta_6,
            theta_7]
        super().__init__(thetas)
```

Euclidean Graph Distribution

(Referenced in 4-Fold Cross-Validation)

```
from EuclideanGraph import EuclideanGraph
import random

class EuclideanGraphDistribution:
    def __init__(self, pt_number_lim = (9, 9), xlim = (-5, 5), ylim = (-5, 5)):
        '''
        :param pt_number: the number of two-dimensional points/vertices to use
        :param xlim: (x_min, x_max) where x_min is the min x coordinate and x_max
            is the max x coordinate for each point
        :param ylim: (y_min, y_max) where y_min is the min y coordinate and y_max
            is the max x coordinate for each point
        '''
        self.pt_number_lim, self.xlim, self.ylim = pt_number_lim, xlim, ylim

    def randomGraph(self):
        '''
        :return: a random graph from the distribution
        '''
        n = random.randint(*self.pt_number_lim)
        points = [(random.randint(*self.xlim), random.randint(*self.ylim)) for i
            in range(n)]
        return EuclideanGraph(points)
```

Held-Karp Algorithm

(Referenced in Approximation Ratio as Error.)

```
from itertools import combinations
from functools import reduce

class TSP_HK:
    def __init__(self):
        pass

    def policy(self, G, Q_dict):
        '''
        :param G: graph
        :param Q_dict: tabular dynamic dictionary
        :return: function that takes path P and starting vertex c and returns (v,
            Q) where v is next best vertex to
        travel to and Q is the dynamic value of (P, c)
        '''
        def f(P, c):
            R = P - {c}
            vQs = [(u, G.w[(c, u)] + Q_dict[(R, u)]) for u in R]
            return reduce(lambda t1, t2: t2 if t1[0] is None or t2[1] < t1[1] else
                t1, vQs, (None, G.w[(c, 0)]))
        return f

    def vBest(self, G, Q_dict):
        '''
        :param G: graph
        :param Q_dict: tabular dynamic dictionary
        :return: function that takes path P and starting vertex c and returns the
            next best vertex to travel to
        '''
        return lambda P, c: self.policy(G, Q_dict)(P, c)[0]

    def QBest(self, G, Q_dict):
        '''
        :param G: graph
        :param Q_dict: tabular dynamic dictionary
        :return: function that takes path P and starting vertex c and returns the
            dynamic value of (P, c)
        '''
        return lambda P, c: self.policy(G, Q_dict)(P, c)[1]

    def getQfromP(self, G, Q_dict):
        '''
```

```
    :param G: graph
    :param Q_dict: tabular dynamic dictionary
    :return: function that takes path P and returns dictionary for keys that
        have (P, _)
    '''
    Q_func = self.QBest(G, Q_dict)
    return lambda P: dict([((P, c), Q_func(P, c)) for c in P])

def createQdict(self, G):
    '''
    :param G: graph
    :return: final dynamic dictionary of subproblems built from the bottom up
    '''
    #print("Creating Q Dict")
    neighbors = [v for v in G.vertices if v != 0]
    def createLevel(Q_dict, i):
        #print("Creating Level {}".format(i))
        if i == G.n:
            return Q_dict
        else:
            X = [frozenset(x) for x in combinations(neighbors, i)]
            Q_dict_func = self.getQfromP(G, Q_dict)
            Q_dict_new = reduce(lambda Q, P: Q | Q_dict_func(P), X, Q_dict)
            return createLevel(Q_dict_new, i + 1)
    return createLevel({}, 1)

def S_not(self, G, S):
    '''
    :param G: graph
    :param S: list of unique vertices of graph
    :return: set of vertices in G but not in S
    '''
    return set(G.vertices).difference(S)

def calculateWalkfromQ(self, G, Q_dict):
    '''
    :param G: graph
    :param Q_dict: dynamic dictionary that contains the subproblems
    :return: permutation of vertices of G that give the shortest Hamiltonian
        cycle
    '''
    #print("Calculating Walk From Q")
    def updated_S(i, S):
        if i == G.n:
            return S
```

```
        else:
            S_not = self.S_not(G, S)
            c = S[-1]
            P = frozenset(S_not).union({c})
            v = self.vBest(G, Q_dict)(P, c)
            return updated_S(i + 1, S + [v])
    return updated_S(1, [0])

def calculateWalk(self, G):
    '''
    :param G: graph
    :return: permutation of vertices of G that give the shortest Hamiltonian
        cycle
    '''
    Q_dict = self.createQdict(G)
    return self.calculateWalkfromQ(G, Q_dict)
```

Walked Graphs

(Referenced in Approximation Ratio as Error.)

```
from EuclideanGraph import EuclideanGraph
import pandas as pd

class WalkedGraphs:
    def __init__(self, file, graphDist = None, n = None):
        '''
        :param file: the file to read from or to write to for the walked graphs
        :param graphDist: the graph distribution used to generate graphs (if the
            file is not already created)
        :param n: the number of graphs to generate (if the file is not already
            created)
        '''
        self.graphs = self.getGraphs(file, graphDist, n)

    def getGraphsFromCSV(self, file):
        '''
        :param file: the file to get the graphs from
        :return: list of graphs that have a calculated permutation S and distance
            d best for TSP
        '''
        df = pd.read_csv(file, index_col=0)
        m = len(df["Graph"].unique())
        def graph(i):
            graph_df = df.loc[lambda df: df["Graph"] == i]
            ordered_points = list(zip(list(graph_df["x"]), list(graph_df["y"])))
            S = list(range(len(ordered_points)))
            d = graph_df["d(G)"].iloc[0]
            return EuclideanGraph(points=ordered_points, shortest_cycle=S,
                distance=d)
        return [graph(i) for i in range(m)]

    def createGraphs(self, graphDist, n):
        '''
        :param graphDist: graph distribution to generate graphs
        :param n: the number of graphs to generate
        :return: list of graphs that have a calculated permutation S and distance
            d best for TSP
        '''
        return [graphDist.randomGraph().thisWithShortestCycle() for i in range(n)]

    def toCSV(self, graphs, file):
        '''
```

```
    :param graphs: list of graphs that have a calculated permutation S and
        distance d best for TSP
    :param file: file to store the graphs
    :return: the dataframe used to store graphs (side effect: saved this
        dataframe to the file)
    '''
    def graphDF(i):
        G = graphs[i]
        ordered_points = [G.points[j] for j in G.shortest_cycle]
        d = G.distance
        xs, ys = tuple(zip(*ordered_points))
        return pd.DataFrame(data = {"Graph": G.n * [i], "x": xs, "y": ys, "d(G
            )": G.n * [d]})
    df = pd.concat([graphDF(i) for i in range(len(graphs))], axis=0).
        reset_index(drop=True)
    df.to_csv(file)
    return df

def getGraphs(self, file, graphDist, n):
    '''
    :param file: the file to get the graphs from
    :param graphDist: the graph distribution used to generate graphs (if the
        file is not already created)
    :param n: the number of graphs to generate (if the file is not already
        created)
    :return: list of graphs that have a calculated permutation S and distance
        d best for TSP
    '''
    try:
        return self.getGraphsFromCSV(file)
    except FileNotFoundError:
        graphs = self.createGraphs(graphDist, n)
        self.toCSV(graphs, file)
        return self.getGraphsFromCSV(file)
```

Analysis 1

(Referenced in Tuning Hyperparameters.)

```
from TSP_RL import TSP_RL
from functools import reduce
from itertools import product
import time
import pandas as pd
import os
from WalkedGraphs import WalkedGraphs

class Analysis_1:
    def __init__(self, hyp_range_dict, k, graph_file = None, graphDist = None, n =
        None):
        '''
        :param hyp_range_dict: the range of values to use for each hyperparameter
        :param k: the number of parts to use for our partition
        :param graph_file: the file to use for our graph data
        :param graphDist: the graph distribution used to generate graphs (if the
            file is not already created)
        :param n: the number of graphs to generate (if the file is not already
            created)
        '''
        self.folder = lambda i: "\\".join([os.getcwd(), "Analysis_{}".format(i)])
        graph_file = "\\".join([self.folder(1), "WalkedGraphs.csv"]) if graph_file
            is None else graph_file
        self.hyp_range_dict = hyp_range_dict
        self.hyp_names = ["p", "T", "eps", "n", "alpha", "beta", "gamma"]
        hyp_combos = product(*[hyp_range_dict[name] for name in self.hyp_names])
        self.hyp_dicts = [dict(zip(self.hyp_names, hyp_values)) for hyp_values in
            hyp_combos]
        self.graphs = WalkedGraphs(graph_file, graphDist, n).graphs
        self.n = len(self.graphs)
        self.train_test_dict = self.get_train_test_dict(k=k)

    def partition(self, k):
        '''
        :param k: the number of parts to use for our partition
        :return: a partition of our graph indices
        '''
        (q, r) = (self.n // k, self.n % k)
        def f(i, j, p):
            return p if i == k else f(i + 1, j + q + int(i < r), p + [list(range(j
                , j + q + int(i < r)))])
        return f(0, 0, [])
```

```
def get_train_test_dict(self, k):
    '''
    :param k: the number of parts to use for our partition
    :return: a dictionary that has key value pairs of (i, train_index(i),
        test_index(i)) for each i in our k parts
    '''
    partition = self.partition(k)
    train_index = lambda i: reduce(lambda l1, l2: l1 + l2, partition[:i] +
        partition[(i+1):])
    test_index = lambda i: partition[i]
    return dict([(i, (train_index(i), test_index(i))) for i in range(k)])

def approx_ratio(self, model, Theta):
    '''
    :param model: TSP Q Learning model to use
    :param Theta: object consisting of list of theta weights to use for Q
        function to approximate best walk
    :return: approximation ratio for calculated walk
    '''
    def f(G):
        S = model.calculateWalk(Theta, G)
        return G.tourDistance(S) / G.distance
    return f

def hyp_error(self, hyp_dict):
    '''
    :param hyp_dict: assignment of hyperparameters to use for the model
    :return: average error, as approximation ratio, of each fold
    '''
    hyp_params = [hyp_dict[k] for k in self.hyp_names]
    tsp = TSP_RL(*hyp_params)
    def fold_error(fold):
        train, test = self.train_test_dict[fold]
        trainGs, testGs = [self.graphs[i] for i in train], [self.graphs[i] for
            i in test]
        _, Theta = tsp.QLearning(trainGs)
        approx_ratio_func = self.approx_ratio(tsp, Theta)
        return 1 / len(test) * sum([approx_ratio_func(G) for G in testGs])

    return 1 / len(self.train_test_dict) * sum([fold_error(fold) for fold in
        self.train_test_dict.keys()])

def getErrorDf(self, append = True):
    '''
```

```
:param append: the decision of whether to append to a preexisting error
    dataframe
:return: an error dataframe that has been made from a preexisting error
    dataframe or made from scratch
'''
def error_row(i):
    hyp_dict = self.hyp_dicts[i]
    hyps = [hyp_dict[name] for name in self.hyp_names]
    error = self.hyp_error(hyp_dict)
    row = hyps + [error]
    df = pd.DataFrame.from_dict(data = {i: row}, orient = "index", columns
        = self.hyp_names + ["Error"])
    df.to_csv("\\".join([self.folder(1), "Error_{}.csv".format(i)]))
    return pd.DataFrame.from_dict(data = {i: row}, orient = "index",
        columns = self.hyp_names + ["Error"])

start_time = time.time()

if append:
    prev_df = pd.read_csv("\\".join([os.getcwd(), "Error", "Error.csv"]),
        index_col=0)
    i = prev_df.index[-1] + 1
    new_row = error_row(i)
    df = pd.concat([prev_df, new_row], axis=0)
    df.to_csv("\\".join([self.folder(1), "Error.csv"]))
else:
    df = error_row(0)
    df.to_csv("\\".join([self.folder(1), "Error.csv"]))

print("Time␣Elapsed:␣{}␣Minutes".format((time.time() - start_time) / 60))
return df
```

Analysis 2

(Referenced in Average Error Grouped By Hyperparameter To Find "Best" Model.)

```
from Analysis_1 import Analysis_1
import pandas as pd
from functools import reduce
import matplotlib.pyplot as plt
import os

class Analysis_2(Analysis_1):
    def __init__(self, graph_file = None, graphDist = None, n = None):
        '''
        :param file: the file to read from or to write to for the walked graphs
        :param graphDist: the graph distribution used to generate graphs (if the
            file is not already created)
        :param n: the number of graphs to generate (if the file is not already
            created)
        '''
        hyp_range_dict = {"p": [3, 4], "T": [2, 1], "eps": [0.01, 0.05], "n": [2,
            3],
                          "alpha": [0.01, 0.1], "beta": [5, 10], "gamma": [0.9, 1]}
        graph_file = "\\".join([os.getcwd(), "Analysis_1", "OurWalkedGraphs.csv"])
             if graph_file is None else graph_file
        super().__init__(hyp_range_dict, 4, graph_file = graph_file, graphDist=
            graphDist, n = n)
        self.error_file = "\\".join([self.folder(1), "Error.csv"])

    def groupedErrorDf(self):
        '''
        :return: function that takes a hyperparameter and returns dataframe
            showing mean error grouped by hyperparameter
        '''
        df = pd.read_csv(self.error_file, index_col=0)
        return lambda name: df.groupby(by = [name])["Error"].mean()

    def groupedErrorDfs(self):
        '''
        :return: dictionary that has key-value pairs (hyp, groupedErrorDf(hyp))
        '''
        groupedErrorDf_func = self.groupedErrorDf()
        def appendDf(df_dict, name):
            grouped_df = groupedErrorDf_func(name)
            grouped_df.to_csv("\\".join([self.folder(2), "Error_By_{}.csv".format(
                name)]))
            return df_dict | {name: grouped_df}
```

```
    return reduce(appendDf, self.hyp_names, {})

def plots(self):
    '''
    :return: None (side effect: shows bar charts of average approximation
        error grouped by each hyperparameter)
    '''
    df_dict = self.groupedErrorDfs()
    def plot(hyp_index):
        fig, axs = plt.subplots(2, 2, figsize = (4, 2))
        hyp_names_title = [self.hyp_names[i] for i in hyp_index]
        fig.suptitle("Average␣Approximation␣Ratio␣By␣Hyperparameter␣(For␣{})".
            format(hyp_names_title))
        for i in range(len(hyp_index)):
            r, c = i // 2, i % 2
            hyp = self.hyp_names[hyp_index[i]]
            df = df_dict[hyp]
            ax = axs[r, c]
            ax.set_xlabel(hyp)
            ax.set_ylim([1, 1.1])
            ax.bar([str(x) for x in df.index], df)
        plt.show()
    for index in [list(range(4)), list(range(4, 7))]:
        plot(index)

def bestParams(self):
    '''
    :return: dictionary that shows the best assignment of values for each
        hyperparameter
    '''
    df_dict = self.groupedErrorDfs()
    def best_param(name):
        grouped_df = df_dict[name]
        min_error = min(grouped_df)
        return grouped_df.loc[lambda df: df == min_error].index[0]
    return dict([(name, best_param(name)) for name in df_dict.keys()])

def bestRows(self):
    '''
    :return: error dataframe filtered by the rows having a perfect
        approximation ratio
    '''
    df = pd.read_csv(self.error_file, index_col=0)
    best_rows = df.loc[lambda df: df["Error"] == 1]
    best_rows.to_csv("\\".join([self.folder(2), "Error_Best.csv"]))
```

```
        return best_rows
```

Analysis 3

(Referenced in Q-Learning With "Best" Model Over 3 Episodes.)

```
from Analysis_2 import Analysis_2
from TSP_RL import TSP_RL
import pandas as pd
from EuclideanGraph import EuclideanGraph
import matplotlib.pyplot as plt
from WalkedGraphs import WalkedGraphs

class Analysis_3 (Analysis_2):
    def __init__(self, graph_file = None, graphDist = None, n = None):
        '''
        :param graph_file: the file to read from or to write to for the walked
            graphs
        :param graphDist: the graph distribution used to generate graphs (if the
            file is not already created)
        :param n: the number of graphs to generate (if the file is not already
            created)
        '''
        super().__init__(graph_file, graphDist, n)
        self.best_hyp_dict = {"p": 4, "T": 2, "eps": 0.01, "n": 3, "alpha": 0.1, "
            beta": 5, "gamma": 0.9}
        self.best_hyp_params = [self.best_hyp_dict[k] for k in self.hyp_names]

    def inversePerm(self, S):
        '''
        :param S: permutation
        :return: the inverse permutation of S
        '''
        index = list(range(len(S)))
        inverseDict = dict(list(zip(S, index)))
        return [inverseDict[i] for i in index]

    def newFile(self, file, note, ext = ".csv"):
        '''
        :param file: file to create a new file from
        :param note: additional note used to name new file
        :param ext: extension of file
        :return: new file
        '''
        file_path = file.split("\\")
        csv_file = file_path[-1].strip(".csv")
```

```
        return "\\".join(file_path[:-1] + ["{}␣({})".format(csv_file, note) + ext
            ])

    def graphDF(self, G_S_list):
        '''
        :param G_S_list: list of (G, S) where G is a graph and S is permutation to
             approximate TSP
        :return: dataframe that stores the graph information as well as S and walk
             distance that approximate TSP
        '''
        def f(i):
            G, S = G_S_list[i]
            ordered_points = [G.points[j] for j in G.shortest_cycle]
            xs, ys = tuple(zip(*ordered_points))
            actualDist = G.distance
            approxOrder = self.inversePerm(S)
            approxDist = G.tourDistance(S)
            approxRatio = approxDist / actualDist
            return pd.DataFrame(data={"Graph": G.n * [i], "x": xs, "y": ys, "
                approxOrder": approxOrder,
                "actualDist": G.n * [actualDist], "approxDist": G.n * [approxDist],
                     "approxRatio": G.n * [approxRatio]})
        return f

    def toCSV(self, graph_file, graphDist = None, n = None):
        '''
        :param graph_file: the file to read from or to write to for the walked
            graphs
        :param graphDist: the graph distribution used to generate graphs (if the
            file is not already created)
        :param n: the number of graphs to generate (if the file is not already
            created)
        :return: dataframe of graphs with approximated best S and walk distance (
            side effect: stores this in a new file)
        '''
        graphs = WalkedGraphs(graph_file, graphDist, n).graphs
        tsp = TSP_RL(*self.best_hyp_params)
        G_S_list, _ = tsp.QLearning(graphs)
        graphDF_func = self.graphDF(G_S_list)
        df = pd.concat([graphDF_func(i) for i in range(len(G_S_list))], axis=0).
            reset_index(drop=True)
        output_file = self.newFile(graph_file, "approxS")
        df.to_csv(output_file)
        return df
```

```
def getG_S_rFromCSV(self, file):
    '''
    :param file: file to get (G, S, r) list from
    :return: (G, S, r) list where G is graph, S is calculated permutation, and
        r is approximation ratio from S
    '''
    df = pd.read_csv(file, index_col=0)
    m = len(df["Graph"].unique())
    def G_S_r(i):
        graph_df = df.loc[lambda df: df["Graph"] == i]
        ordered_points = list(zip(list(graph_df["x"]), list(graph_df["y"])))
        S_true = list(range(len(ordered_points)))
        d_true = graph_df["actualDist"].iloc[0]
        S_approx = self.inversePerm(graph_df["approxOrder"])
        approx_ratio = graph_df["approxRatio"].iloc[0]
        return EuclideanGraph(points=ordered_points, shortest_cycle=S_true,
            distance=d_true), S_approx, approx_ratio
    return [G_S_r(i) for i in range(m)]

def getG_S_r(self, graph_file, graphDist, n):
    '''
    :param graph_file: the file to read from or to write to for the walked
        graphs
    :param graphDist: the graph distribution used to generate graphs (if the
        file is not already created)
    :param n: the number of graphs to generate (if the file is not already
        created)
    :return: (G, S, r) list where G is graph, S is calculated permutation, and
        r is approximation ratio from S
    '''
    try:
        return self.getG_S_rFromCSV(self.newFile(graph_file, "approxS"))
    except FileNotFoundError:
        self.toCSV(graph_file, graphDist, n)
        return self.getG_S_rFromCSV(self.newFile(graph_file, "approxS"))

def drawGraphs(self, graph_file, graphDist, n):
    '''
    :param graph_file: the file to read from or to write to for the walked
        graphs
    :param graphDist: the graph distribution used to generate graphs (if the
        file is not already created)
    :param n: the number of graphs to generate (if the file is not already
        created)
```

```
    :return: None (side effect: saves and shows plots of graphs drawn with
        optimal path along with calculated path)
    '''
    G_S_r_list = self.getG_S_r(graph_file, graphDist, n)
    fig, axs = plt.subplots(1, n)
    #fig.suptitle("Learned Paths of Graphs over {} Episodes".format(n))

    def graph_plot(i):
        G, S, r = G_S_r_list[i]
        ax = axs[i]
        ordered_points = G.points
        closed = lambda z: z + (z[0],)
        x_true, y_true = tuple(zip(*ordered_points))
        ax.set_title("Episode␣{}".format(i + 1))
        ax.set_xlabel("Approx␣Ratio:␣{}".format(round(r, 2)))
        ax.plot(x_true, y_true, "o")
        x_approx, y_approx = tuple(zip(*[G.points[j] for j in S]))
        ax.plot(closed(x_approx), closed(y_approx), "-", label="Approximated")
        ax.plot(closed(x_true), closed(y_true), "-", label = "Optimal")
        ax.legend()
        #plt.savefig(self.newFile(graph_file, "Graph {} Plot".format(i), ext
            =".png"))
        #plt.show()
    [graph_plot(i) for i in range(len(G_S_r_list))]
    plt.savefig(self.newFile(graph_file, "Learned␣Paths", ext=".png"))
    plt.show()
```

Dataframes

Walked Graphs Dataframe for Cross-Validation

(Referenced in 4-Fold Cross-Validation)

	Graph	x	y	d(G)
0	0	2	0	28.634036
1	0	4	-3	28.634036
2	0	3	-4	28.634036
3	0	1	-3	28.634036
4	0	-1	-4	28.634036
5	0	-1	0	28.634036
6	0	-4	3	28.634036
7	0	-5	3	28.634036
8	0	-3	5	28.634036
9	1	1	5	29.235448
10	1	5	0	29.235448
11	1	2	-4	29.235448
12	1	1	-1	29.235448
13	1	-1	0	29.235448
14	1	-4	0	29.235448
15	1	-4	3	29.235448
16	1	-1	5	29.235448
17	1	0	4	29.235448
18	2	-4	5	37.239927
19	2	4	5	37.239927
20	2	4	3	37.239927
21	2	0	1	37.239927
22	2	2	-1	37.239927
23	2	1	-2	37.239927
24	2	1	-4	37.239927
25	2	-5	-5	37.239927
26	2	-3	2	37.239927
27	3	-5	0	29.534266
28	3	-1	1	29.534266
29	3	-1	2	29.534266
30	3	0	4	29.534266
31	3	2	5	29.534266
32	3	3	-2	29.534266
33	3	4	-3	29.534266
34	3	2	-5	29.534266
35	3	1	-4	29.534266
36	4	-2	-1	28.408629
37	4	0	4	28.408629
38	4	-3	3	28.408629
39	4	-5	-3	28.408629
40	4	-5	-5	28.408629
41	4	-3	-5	28.408629
42	4	-1	-4	28.408629
43	4	1	-5	28.408629
44	4	0	-3	28.408629
45	5	-5	2	28.482606
46	5	-5	-1	28.482606
47	5	-3	-4	28.482606
48	5	-2	-1	28.482606
49	5	0	0	28.482606
50	5	-1	2	28.482606
51	5	1	4	28.482606
52	5	2	5	28.482606
53	5	-5	5	28.482606
54	6	1	1	27.001397
55	6	2	-1	27.001397
56	6	3	-2	27.001397
57	6	0	-5	27.001397
58	6	-4	-4	27.001397
59	6	-5	1	27.001397
60	6	-4	2	27.001397
61	6	-3	2	27.001397
62	6	1	4	27.001397
63	7	4	4	29.460961
64	7	0	5	29.460961
65	7	0	3	29.460961
66	7	-4	-3	29.460961
67	7	-4	-3	29.460961
68	7	-3	-4	29.460961
69	7	0	-4	29.460961
70	7	1	-5	29.460961
71	7	5	2	29.460961

Models With $\rho = 1$ (Referenced in Average Error Grouped By Hyperparameter To Find "Best" Model)

	p	T	eps	n	alpha	beta	gamma	Error
1	3	2	0.010000	2	0.010000	5	1.000000	1.000000
2	3	2	0.010000	2	0.010000	10	0.900000	1.000000
3	3	2	0.010000	2	0.010000	10	1.000000	1.000000
5	3	2	0.010000	2	0.100000	5	1.000000	1.000000
6	3	2	0.010000	2	0.100000	10	0.900000	1.000000
8	3	2	0.010000	3	0.010000	5	0.900000	1.000000
9	3	2	0.010000	3	0.010000	5	1.000000	1.000000
10	3	2	0.010000	3	0.010000	10	0.900000	1.000000
11	3	2	0.010000	3	0.010000	10	1.000000	1.000000
13	3	2	0.010000	3	0.100000	5	1.000000	1.000000
15	3	2	0.010000	3	0.100000	10	1.000000	1.000000
32	3	1	0.010000	2	0.010000	5	0.900000	1.000000
34	3	1	0.010000	2	0.010000	10	0.900000	1.000000
36	3	1	0.010000	2	0.100000	5	0.900000	1.000000
37	3	1	0.010000	2	0.100000	5	1.000000	1.000000
42	3	1	0.010000	3	0.010000	10	0.900000	1.000000
44	3	1	0.010000	3	0.100000	5	0.900000	1.000000
48	3	1	0.050000	2	0.010000	5	0.900000	1.000000
57	3	1	0.050000	3	0.010000	5	1.000000	1.000000
65	4	2	0.010000	2	0.010000	5	1.000000	1.000000
67	4	2	0.010000	2	0.010000	10	1.000000	1.000000
68	4	2	0.010000	2	0.100000	5	0.900000	1.000000
69	4	2	0.010000	2	0.100000	5	1.000000	1.000000
71	4	2	0.010000	2	0.100000	10	1.000000	1.000000
72	4	2	0.010000	3	0.010000	5	0.900000	1.000000
73	4	2	0.010000	3	0.010000	5	1.000000	1.000000
74	4	2	0.010000	3	0.010000	10	0.900000	1.000000
75	4	2	0.010000	3	0.010000	10	1.000000	1.000000
76	4	2	0.010000	3	0.100000	5	0.900000	1.000000
77	4	2	0.010000	3	0.100000	5	1.000000	1.000000
78	4	2	0.010000	3	0.100000	10	0.900000	1.000000
79	4	2	0.010000	3	0.100000	10	1.000000	1.000000
98	4	1	0.010000	2	0.010000	10	0.900000	1.000000
103	4	1	0.010000	2	0.100000	10	1.000000	1.000000
104	4	1	0.010000	3	0.010000	5	0.900000	1.000000
105	4	1	0.010000	3	0.010000	5	1.000000	1.000000
109	4	1	0.010000	3	0.100000	5	1.000000	1.000000
114	4	1	0.050000	2	0.010000	10	0.900000	1.000000
118	4	1	0.050000	2	0.100000	10	0.900000	1.000000

Gradient Calculation

(Referenced in Q-Learning.)

Notation

- $N \in \{0,1\}^{m \times m}$ such that $N[:, v] \in \{0,1\}^{m \times 1}$ gives the neighbors of v
- $E_v \in \{0,1\}^{m \times 1}$ is the unit vector associated with vertex v
- W_{flat} is the flattened weight matrix w
- $U = [[N[:, v] \otimes E_v^\intercal]_{v \in V}] \in \{0,1\}^{m^2 \times m}$

Calculating $r \cdot \dfrac{\partial(\theta_1 \ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:, u] + \theta_3 \mathbf{relu}(\theta_4 \otimes W_{flat}) U[:, u])}{\partial \theta_1}$
For $r \in \mathbb{R}^{1 \times p}, u \in V, t > 0$

$$
\begin{aligned}
& r \cdot \frac{\partial(\theta_1 \ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:, u] + \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U[:, u])}{\partial \theta_1} \\
&= r \cdot \frac{\partial \theta_1 \ X_S[u]}{\partial \theta_1} + r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:, u]}{\partial \theta_1} + r \cdot \frac{\partial \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U[:, u])}{\partial \theta_1} \\
&= r \cdot \frac{\partial \theta_1 \ X_S[u]}{\partial \theta_1} + r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:, u]}{\partial \theta_1} + 0_{1 \times p} \\
&= X_S[u] * r + \sum_{q \in \mathcal{N}(u)} \underbrace{[r \cdot \theta_2]}_{\in \mathbb{R}^{1 \times p}} \cdot \frac{\partial (\mu_S^{(t-1)})_q}{\partial \theta_1}
\end{aligned}
$$

Calculating $r \cdot \dfrac{\partial(\theta_1\ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:,u] + \theta_3 \mathbf{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_2}$

For $r \in \mathbb{R}^{1\times p}, u \in V, t > 0$

$$
\begin{aligned}
& r \cdot \frac{\partial(\theta_1\ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:,u] + \theta_3 \mathrm{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_2} \\
&= r \cdot \frac{\partial \theta_1\ X_S[u]}{\partial \theta_2} + r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:,u]}{\partial \theta_2} + r \cdot \frac{\partial \theta_3 \mathrm{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_2} \\
&= 0_{p\times p} + r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:,u]}{\partial \theta_2} + 0_{p\times p} \\
&= r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:,u]}{\partial \theta_2} \\
&= \sum_{q \in \mathcal{N}(u)} \{ \underbrace{[r \cdot \theta_2]}_{\in \mathbb{R}^{1\times p}} \cdot \frac{\partial (\mu_S^{(t-1)})_q}{\partial \theta_2} \} + r \cdot \frac{\partial \theta_2}{\partial \theta_2} \cdot [\mu_S^{(t-1)} N[:,u]] \\
&= \sum_{q \in \mathcal{N}(u)} \{ \underbrace{[r \cdot \theta_2]}_{\in \mathbb{R}^{1\times p}} \cdot \frac{\partial (\mu_S^{(t-1)})_q}{\partial \theta_2} \} + [\mu_S^{(t-1)} N[:,u]] \otimes r
\end{aligned}
$$

Calculating $r \cdot \dfrac{\partial(\theta_1\ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:,u] + \theta_3 \mathbf{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_3}$

For $r \in \mathbb{R}^{1\times p}, u \in V, t > 0$

$$
\begin{aligned}
& r \cdot \frac{\partial(\theta_1\ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:,u] + \theta_3 \mathrm{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_3} \\
&= r \cdot \frac{\partial \theta_1\ X_S[u]}{\partial \theta_3} + r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:,u]}{\partial \theta_3} + r \cdot \frac{\partial \theta_3 \mathrm{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_3} \\
&= 0_{p\times p} + r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:,u]}{\partial \theta_3} + r \cdot \frac{\partial \theta_3 \mathrm{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_3} \\
&= r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:,u]}{\partial \theta_3} + r \cdot \frac{\partial \theta_3 \mathrm{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_3} \\
&= \sum_{q \in \mathcal{N}(u)} \{ \underbrace{[r \cdot \theta_2]}_{\in \mathbb{R}^{1\times p}} \cdot \frac{\partial (\mu_S^{(t-1)})_q}{\partial \theta_3} \} + r \cdot \frac{\partial \theta_3}{\partial \theta_3} \cdot [\mathrm{relu}(\theta_4 \otimes W_{flat}) U[:,u]] \\
&= \sum_{q \in \mathcal{N}(u)} \{ \underbrace{[r \cdot \theta_2]}_{\in \mathbb{R}^{1\times p}} \cdot \frac{\partial (\mu_S^{(t-1)})_q}{\partial \theta_3} \} + [\mathrm{relu}(\theta_4 \otimes W_{flat}) U[:,u])] \otimes r
\end{aligned}
$$

Calculating $r \cdot \dfrac{\partial(\theta_1\ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:,u] + \theta_3 \mathbf{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_4}$

For $r \in \mathbb{R}^{1\times p}, u \in V, t > 0$

$$
\begin{aligned}
& r \cdot \frac{\partial(\theta_1\ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:,u] + \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_4} \\
&= r \cdot \frac{\partial \theta_1\ X_S[u]}{\partial \theta_4} + r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:,u]}{\partial \theta_4} + r \cdot \frac{\partial \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_4} \\
&= 0_{p\times p} + r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:,u]}{\partial \theta_4} + r \cdot \frac{\partial \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_4} \\
&= r \cdot \frac{\partial \theta_2 \mu_S^{(t-1)} N[:,u]}{\partial \theta_4} + r \cdot \frac{\partial \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_4} \\
&= \sum_{q \in \mathcal{N}(u)} \{ \underbrace{[r \cdot \theta_2]}_{\in \mathbb{R}^{1\times p}} \cdot \frac{\partial (\mu_S^{(t-1)})_q}{\partial \theta_4} \} + \sum_{q \in \mathcal{N}(u)} \{ [r \cdot \theta_3] \cdot \frac{\partial \text{relu}(\theta_4 * w(u,q))}{\partial \theta_4} \} \\
&= \sum_{q \in \mathcal{N}(u)} \{ \underbrace{[r \cdot \theta_2]}_{\in \mathbb{R}^{1\times p}} \cdot \frac{\partial (\mu_S^{(t-1)})_q}{\partial \theta_4} \} + \sum_{q \in \mathcal{N}(u)} \{ w(u,q) * [r \cdot \theta_3 \cdot \frac{\partial \text{relu}(\theta_4 * w(u,q))}{\partial \theta_4 * w(u,q)}] \}
\end{aligned}
$$

Calculating $r \cdot \dfrac{\partial (\mu_S^{(t)})_u}{\partial \theta_i}$ **For** $i \in \{1,2,3,4\}, r \in \mathbb{R}^{1\times p}, u \in V, t > 0$

$$
\begin{aligned}
r \cdot \frac{\partial (\mu_S^{(t)})_u}{\partial \theta_i} &= r \cdot \frac{\partial (\text{relu}(\theta_1 \otimes X_S + \theta_2 \mu_S^{(t-1)} N + \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U))_u}{\partial \theta_i} \\
&= r \cdot \frac{\partial \text{relu}(\theta_1\ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:,u] + \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_i} \\
&= \underbrace{[r \cdot \frac{\partial \text{relu}(\theta_1\ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:,u] + \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial(\theta_1\ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:,u] + \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U[:,u])}]}_{\in \mathbb{R}^{1\times p}} \\
&\qquad \cdot \frac{\partial(\theta_1\ X_S[u] + \theta_2 \mu_S^{(t-1)} N[:,u] + \theta_3 \text{relu}(\theta_4 \otimes W_{flat}) U[:,u])}{\partial \theta_i}
\end{aligned}
$$

Calculating $\frac{\partial \hat{Q}_\Theta(S,v)}{\partial \theta_i}$ For $i \in \{1,2,3,4\}$

$$
\begin{aligned}
\frac{\partial \hat{Q}_\Theta(S,v)}{\partial \theta_i} &= \frac{\partial(\theta_{5a}^\intercal \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1}) + \theta_{5b}^\intercal \text{relu}(\theta_7 \mu_S^{(T)}) E_v)}{\partial \theta_i} \\
&= \frac{\partial \theta_{5a}^\intercal \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1})}{\partial \theta_i} + \frac{\partial \theta_{5b}^\intercal \text{relu}(\theta_7 \mu_S^{(T)}) E_v}{\partial \theta_i} \\
&= \frac{\partial \theta_{5a}^\intercal \text{relu}(\theta_6 \sum_{u\in V} (\mu_S^{(T)})_u)}{\partial \theta_i} + \frac{\partial \theta_{5b}^\intercal \text{relu}(\theta_7 (\mu_S^{(T)})_v)}{\partial \theta_i} \\
&= [\theta_{5a}^\intercal \cdot \frac{\partial \text{relu}(\sum_{u\in V} \theta_6 (\mu_S^{(T)})_v)}{\partial (\sum_{u\in V} \theta_6 (\mu_S^{(T)})_u)} \cdot \theta_6] \cdot \sum_{u\in V} \frac{\partial (\mu_S^{(T)})_u}{\partial \theta_i} \} + [\theta_{5b}^\intercal \cdot \frac{\partial \text{relu}(\theta_7 (\mu_S^{(T)})_v)}{\partial \theta_7 (\mu_S^{(T)})_v} \cdot \theta_7] \cdot \frac{\partial (\mu_S^{(T)})_v}{\partial \theta_i} \\
&= \sum_{u\in V} \{ \underbrace{[\theta_{5a}^\intercal \cdot \frac{\partial \text{relu}(\sum_{u\in V} \theta_6 (\mu_S^{(T)})_v)}{\partial (\sum_{u\in V} \theta_6 (\mu_S^{(T)})_u)} \cdot \theta_6]}_{\in \mathbb{R}^{1\times p}} \cdot \frac{\partial (\mu_S^{(T)})_u}{\partial \theta_i} \} + \underbrace{[\theta_{5b}^\intercal \cdot \frac{\partial \text{relu}(\theta_7 (\mu_S^{(T)})_v)}{\partial \theta_7 (\mu_S^{(T)})_v} \cdot \theta_7]}_{\in \mathbb{R}^{1\times p}} \cdot \frac{\partial (\mu_S^{(T)})_v}{\partial \theta_i}
\end{aligned}
$$

Calculating $\frac{\partial \hat{Q}_\Theta(S,v)}{\partial \theta_{5a}}$

$$
\begin{aligned}
\frac{\partial \hat{Q}_\Theta(S,v)}{\partial \theta_{5a}} &= \frac{\partial(\theta_{5a}^\intercal \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1}) + \theta_{5b}^\intercal \text{relu}(\theta_7 \mu_S^{(T)}) E_v)}{\partial \theta_{5a}} \\
&= \frac{\partial \theta_{5a}^\intercal \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1})}{\partial \theta_{5a}} + \frac{\partial \theta_{5b}^\intercal \text{relu}(\theta_7 \mu_S^{(T)}) E_v}{\partial \theta_{5a}} \\
&= \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1})^\intercal + 0_{1\times p} \\
&= \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1})^\intercal
\end{aligned}
$$

Calculating $\frac{\partial \hat{Q}_\Theta(S,v)}{\partial \theta_{5b}}$

$$\begin{aligned}
\frac{\partial \hat{Q}_\Theta(S,v)}{\partial \theta_{5b}} &= \frac{\partial(\theta_{5b}^\intercal \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1}) + \theta_{5b}^\intercal \text{relu}(\theta_7 \mu_S^{(T)}) E_v)}{\partial \theta_{5b}} \\
&= \frac{\partial \theta_{5b}^\intercal \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1})}{\partial \theta_{5b}} + \frac{\partial \theta_{5b}^\intercal \text{relu}(\theta_7 \mu_S^{(T)}) E_v}{\partial \theta_{5b}} \\
&= 0_{1\times p} + (\text{relu}(\theta_7 \mu_S^{(T)}) E_v)^\intercal \\
&= (\text{relu}(\theta_7 \mu_S^{(T)}) E_v)^\intercal
\end{aligned}$$

Calculating $\frac{\partial \hat{Q}_\Theta(S,v)}{\partial \theta_6}$

$$\begin{aligned}
\frac{\partial \hat{Q}_\Theta(S,v)}{\partial \theta_6} &= \frac{\partial(\theta_{5b}^\intercal \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1}) + \theta_{5b}^\intercal \text{relu}(\theta_7 \mu_S^{(T)}) E_v)}{\partial \theta_6} \\
&= \frac{\partial \theta_{5b}^\intercal \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1})}{\partial \theta_6} + \frac{\partial \theta_{5b}^\intercal \text{relu}(\theta_7 \mu_S^{(T)}) E_v}{\partial \theta_6} \\
&= [\theta_{5a}^\intercal \cdot \frac{\partial \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1})}{\partial \theta_6 \mu_S^{(T)} 1_{m\times 1}}] \cdot \frac{\partial \theta_6}{\partial \theta_6} \cdot [\mu_S^{(T)} 1_{m\times 1}] + 0_{p\times p} \\
&= [\theta_{5a}^\intercal \cdot \frac{\partial \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1})}{\partial \theta_6 \mu_S^{(T)} 1_{m\times 1}}] \cdot \frac{\partial \theta_6}{\partial \theta_6} \cdot [\mu_S^{(T)} 1_{m\times 1}] \\
&= [\mu_S^{(T)} 1_{m\times 1}] \otimes [\theta_{5a}^\intercal \cdot \frac{\partial \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1})}{\partial \theta_6 \mu_S^{(T)} 1_{m\times 1}}]
\end{aligned}$$

Calculating $\frac{\partial \hat{Q}_\Theta(S,v)}{\partial \theta_7}$

$$\begin{aligned}
\frac{\partial \hat{Q}_\Theta(S,v)}{\partial \theta_7} &= \frac{\partial(\theta_{5b}^\intercal \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1}) + \theta_{5b}^\intercal \text{relu}(\theta_7 \mu_S^{(T)}) E_v)}{\partial \theta_7} \\
&= \frac{\partial \theta_{5b}^\intercal \text{relu}(\theta_6 \mu_S^{(T)} 1_{m\times 1})}{\partial \theta_7} + \frac{\partial \theta_{5b}^\intercal \text{relu}(\theta_7 \mu_S^{(T)}) E_v}{\partial \theta_7} \\
&= 0_{p\times p} + [\theta_{5b}^\intercal \cdot \frac{\partial \text{relu}(\theta_7 \mu_S^{(T)})}{\theta_7 \mu_S^{(T)}}] \cdot \frac{\partial \theta_7}{\partial \theta_7} \cdot [\mu_S^{(T)} E_v] \\
&= [\theta_{5b}^\intercal \cdot \frac{\partial \text{relu}(\theta_7 \mu_S^{(T)})}{\theta_7 \mu_S^{(T)}}] \cdot \frac{\partial \theta_7}{\partial \theta_7} \cdot [\mu_S^{(T)} E_v] \\
&= [\mu_S^{(T)} E_v] \otimes [\theta_{5b}^\intercal \cdot \frac{\partial \text{relu}(\theta_7 \mu_S^{(T)})}{\theta_7 \mu_S^{(T)}}]
\end{aligned}$$

Calculating $\frac{\partial \max_{v'\in \overline{S}} \hat{Q}_\Theta(S,v')}{\partial \theta_i}$ **For** $i \in \{1,2,3,4,5a,5b,6,7\}$

- If $\overline{S} = \emptyset$,
  - Then, $\frac{\partial \max_{v'\in \overline{S}} \hat{Q}_\Theta(S,v')}{\partial \theta_i} = 0$
- Else
  - Let $v^* := \text{argmax}_{v'\in \overline{S}} \hat{Q}_\Theta(S,v')$
  - Then, $\frac{\partial \max_{v'\in \overline{S}} \hat{Q}_\Theta(S,v')}{\partial \theta_i} = \frac{\partial \hat{Q}_\Theta(S,v^*)}{\partial \theta_i}$

Calculating $\frac{\partial J(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t; \Theta)}{\partial \theta_i}$ **For** $i \in \{1,2,3,4,5a,5b,6,7\}$

$$\begin{aligned}
&\frac{\partial J(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t; \Theta)}{\partial \theta_i} \\
&= (R_{t-n,t} + \max_{v'\in \overline{S_t}} \hat{Q}_\Theta(S_t,v') - \hat{Q}_\Theta(S_{t-n}, v_{t-n})) \cdot \frac{\partial(R_{t-n,t} + \max_{v'\in \overline{S}} \hat{Q}_\Theta(S_t,v') - \hat{Q}_\Theta(S_{t-n}, v_{t-n}))}{\partial \theta_i} \\
&= (R_{t-n,t} + \max_{v'\in \overline{S_t}} \hat{Q}_\Theta(S_t,v') - \hat{Q}_\Theta(S_{t-n}, v_{t-n})) \cdot (\frac{\partial R_{t-n,t}}{\partial \theta_i} + \frac{\partial \max_{v'\in \overline{S}} \hat{Q}_\Theta(S_t,v')}{\partial \theta_i} - \frac{\partial \hat{Q}_\Theta(S_{t-n}, v_{t-n}))}{\partial \theta_i} \\
&= (R_{t-n,t} + \max_{v'\in \overline{S_t}} \hat{Q}_\Theta(S_t,v') - \hat{Q}_\Theta(S_{t-n}, v_{t-n})) \cdot (\frac{\partial \max_{v'\in \overline{S}} \hat{Q}_\Theta(S_t,v')}{\partial \theta_i} - \frac{\partial \hat{Q}_\Theta(S_{t-n}, v_{t-n})}{\partial \theta_i})
\end{aligned}$$