Reinforcement Learning Algorithms to Find The Shortest Paths in Stochastic Graphs

Ian Sinclair [*] [1]

Project Summary

Graph theory has become a fundamental element of modern system's analysis. In particular, many problems in internet routing or data visualization can be simplified to a graph data structure. And so, there emerges a significant need to better understand complicated graphs, and more precisely, investigate efficiencies in traversing between nodes on a graph. (Tsai et al., 2021) Here we approach a machine learning algorithm, adapted from Q-learning, to find the minimum path between two fixed nodes within a fully connected graph structure.
Critically, there are already heuristic approaches such as Dijkstra's or A^* algorithms that are commonly used to query shortest paths. However, these suffer from time and space limitations for large graphs. And so, the development of a superior methods that meets the requirements of modern infrastructure is becoming critical.
We contribute to the investigation of machine learning on stochastic graph theory by implementing a reinforcement learning algorithm to a toy problem, or a smaller test graph. Particularly, we investigate a reinforcement learning strategy which allows an 'agent' to repetitively traverse subsets of the test graph in search of the destination node. Then, ideally the agent is able to optimize its route on each iteration, eventually resulting in an approximation for the minimum path created without necessarily needing to visit every node; and therefore, dramatically reducing the time complexity to find a solution. As an extension of this, we assume a Q-learning strategy to store the path options the agent can consider, then by designing a Q-table such that its optimization coincides the the shortest path between two nodes we can ensure that after enough iterations the agent will in fact find the shortest path. Unfortunately, our results are inconclusive, and so this paper adopts an educational standpoint on the strengths and theory of using machine learning to solve stochastic graphs.

[*]Equal contribution [1]Undergraduate: College of Natural Science and Mathematics, University of Denver.

University of Denver: Daniel Felix Ritchie School of Engineering & Computer Science.
ENCE 3631 Machine Learning. Dr. Zhihui Zhu. June 11, 2021

1. Introduction

Graph structures are an integral part of modern data science and architecture; in particular, they are used to model the connectivity between objects or attributes in large networks of information. This could be used to find similarities between individuals on a social network, or discovering different routes from one location to another on a GPS or digital road map. (Gilad, 2001). However, one of the more famous applications of graphs is to the internet routing problem, in which packets of data that are sent over the internet must be bounced off routers along the way, which creates a massive network of moving data. Critically, here, it costs time and energy to transport the data packets and so inefficiencies in the route from one location to the other can lead to large-scale end-to-end delays. (Ben-Ameur and Gourdin, 2003). As a result, there is much research investigating methods to find the optimal path between graph elements. Here we use the following terminology to introduce shortest path problems.

1.1. Basic Terminology:

We first define a particular location on a graph as a *node*, v, then the *node space* is a set containing every node, $V = \{v_1, v_2, ..., v_n\}$. Critically, here we assume the node space is finite; which is typically the case for real-world data architecture and visualization problems. However, is not necessarily true for all applications.
Next, an *edge* connects two nodes together, and is denoted (v_i, v_j). Which indicates a connection between node v_i to v_j. Additionally, we consider the edge space to be the set of all possible edges with a graph, $E = \{(v_i, v_j) | v_i \to v_j, i, j \leq n\}$, where n is the size of the node space.
Now note that the size of any particular edge can vary, or that is it possible for each edge to have a different length. And so it becomes necessary to define a method for calculating the length of each edge or the distance between nodes. Accordingly, we define each length separately as a weight, using a square weighting matrix, $W = [\delta_{i,j}]$, where each $\delta_{i,j}$ denotes the cost of travelling down edge (v_i, v_j). Clearly this matrix must be $n \times n$; however, because it is not guaranteed that every node will connect to every other node, there may exists weights that are impossible, but still

defined in the weighting matrix. In this case, we give the edge infinite weight, $\delta_{i,j} = \infty$. Which preserves the functionality of the weighting matrix.

Lastly, we can define a path as a set of nodes with connecting edges that takes some start node v_s to a destination node v_d. And so, let p be a path, then $p = \{v_s, v_{s+1}, ..., v_d\}$, where each node, v_i, v_{i+1} has a valid (non infinite) connecting edge, (v_i, v_{i+1}). Now, because of the interconnections of graph structures there may be multiple paths taking $v_s \rightarrow v_d$, and so we define the path space as $P = \{p_i | i = 1, 2, 3, ..., k\}$, or the set of all paths taking $v_s \rightarrow v_d$. From here, we can define the length of a particular path by the sum of edge weights between each node pair. $|p_i| = \sum_{v_k, v_{k+1} \in p_i} \delta_{k,k+1}$. Therefore, to select the optimal path, p^*, from the path space, we can define $p^* = \arg\min_{p_i \in P} \sum_{v_k, v_{k+1} \in p_i} \delta_{k,k+1}$. Or the path with the least total cost to traverse.

Figure 1 displays a visual representation of the listed terminology in an example graph. Form here, we have defined a method to describe graphs mathematically and can begin begin analyzing the attributes of modern application problems.

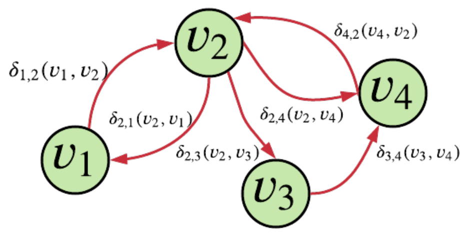

Figure 1. Graph Example: including basic terminology and labels

1.2. Routing Problem

In internet routing scenarios, data packets are bounced off routers to reach a final destination. This creates a massive graph network in which each router is a node and the accompanying cost of sending the data between routers is an edge. Additionally, the choice of route for each packet can significantly impact the time/energy required for it to reach its destination. From here, "The routing of the flows of traffic in an Internet network is completely determined by the choice of a routing protocol and the setting of its parameters....it most often reduces to avoid congestion and therefore to the ability to direct flow where network resources are available." (Ben-Ameur and Gourdin, 2003). As an extension, finding efficiencies, or minimum paths, between routers is critical; however, because many data packets are being sent simultaneously the traffic for a particular route may change over time. Therefore, the weight on a particular edge cannot be modeled by a constant δ.

1.2.1. Stochastic Graphs:

In a stochastic graph, the edge weights between any two nodes can vary by some probability distributed over time. And so, we redefine the weighing matrix. Such that we allow the cost to travel between nodes v_i, v_j by the continuous PDF, $f_{i,j} = \mathcal{P}(\delta_{i,j}(v_i, v_j) = x)$ for some random variable $x \in \mathbf{R}$. (Xia et al., 2019). Now, we can define the $n \times n$ weighting matrix, as $W = [f_{i,j}]$ for any node pair in the node space. Critically, we can still define non-existing edges as having infinite weight by $\mathrm{E}[f_{i,j}] = \infty$, for (v_i, v_j) is not a valid edge.

Lastly, we can define the length of a path as, $|p_i| = \sum_{v_k, v_{k+1} \in p_i} \mathrm{E}[f_{i,j}]$, or the sum of expected lengths from each distribution function between the nodes in the path. And so the optimal path is defined similarly by, $|p^*| = \arg\min_{p_i} \sum_{v_k, v_{k+1} \in p_i} \mathrm{E}[f_{i,j}]$.

2. Machine Learning and Shortest Paths

To approach our analysis of machine learning in modern data applications, we consider a toy problem, or a smaller more controllable graph than what would be expected in the real-world. This allows us to better define the relationships between hyper parameters before implementing the program into more complicated networks. In particular we consider a data set graph with 25 nodes and 160 edges, figure 2. Additionally, to add variability to the number of unique paths from the source to the destination node we choose to implement an undirected graph; such that, if (v_i, v_j) exists then (v_j, v_i) also exists and $f_{i,j} = f_{j,i}$. And so we denote, $(v_i, v_j), (v_j, v_i)$ with the same line.

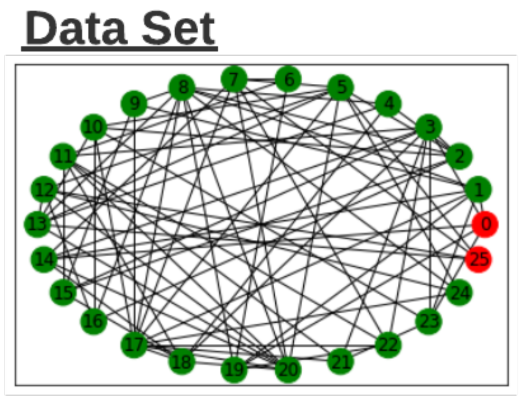

Figure 2. Visualization of Toy Problem Data Set.

Critically, within our data set, shown in figure 2, we desire to find the minimum path between node 0, and node 25, shown in red. Additionally, we take the expected edge weight for any valid edge to be a randomly generated integer on the interval $[1, 100]$. Note, within our model, the data set is engineering using a library import called

$networkX$, which is a common graph theory library. Now, we consider an implementation of a machine learning technique, reinforcement/Q-learning, in which the optimal path is learned as a behavioral policy over many iterations. However, we also desire a way to test the validity of the results obtained by the machine learning algorithm. And so, to work as a control group, we also implement a heuristic algorithm commonly used in modern shortest path problems.

2.1. Experimental Validity and Control Group

Because our machine learning algorithm is designed to reach a close approximation to the minimum path it is difficult to determine the complete accuracy of the final result. And so we consider a heuristic model, *Dijkstra's* algorithm as a control group. To which we compare the runtime vs accuracy of our machine learning algorithm against Dijkstra's algorithm over many independent trials. In particular we use the *NetworkX* build in implementation of Dijkstra's algorithm to run the control test. Importantly, Dijkstra's algorithm finds the shortest path from a source node to every other node and has therefore been a critical tool in modern shortest path problems. (Goldberg and Harrelson, 2005). However, it has a runtime of $T(n) = O(V + E)$, and a space complexity of $S(n) = (V^2)$ and so for large graphs this algorithm creates time/space bottlenecks in the computation of shortest paths. Thus, we consider a machine learning approach that may not be constrained by the same limitations.

2.2. Reinforcement Learning

In particular, we consider a reinforcement learning algorithm that reduces our graph structure data set to an *environment*, specifically, we define an agent that behaves similar to a human in that it traverses the graph, making decisions along the way and gets some kind of reward or punishment for good behavior, or making choices resulting in minimum paths. This it a globally iterative process so we can define an episode as a particular trial where the environment and agents position is reset and the agent start navigating the graph from the beginning, using past knowledge to try to standardize a method of behavior. This method of behavior is defined as a policy, π. As a result, the goal of reinforcement learning is to find the optimal policy that accumulates the most rewards. (Xia et al., 2019). Now critically, we desire a method that ensures the optimal policy coincides with the shortest path of our data set. In many machine learning models this is accomplished by the *Markov Decision Process* which is a natural extension from Markov Chains and allows the actions of an agent in a partially random environment to be represented mathematically.

2.3. Markov Decision Process (MDP)

The Markov Decision Process (MDP) can be defined by the tuple, $\mathcal{M} = (S, A, \rho, R, \gamma)$, where each elements represents a key component to the reinforcement learning processes. Furthermore, build connections for the following implementation on concepts from (Bäuerle and Rieder, 2011).

2.3.1. State Space, S

Within reinforcement learning, the agent must make decisions given events in the environment. These events can be categorized as *states*. Or that a *state*, s, is a particular position the agent can exist within the environment where it must make a decision. And so we can define the state space, S, as the set of every possible state, $S = \{s_1, ..., s_m\}$. Now; critically, we are interested in connections between nodes, and so the only location an agent can visit is a node. Therefore, we allow the state space to be equal to the node space, $S = V = \{v_1, ..., v_n\}$.

2.3.2. Action Space, A

Now, given a particular state, the agent can preform any one of a finite set of actions, a, that will move it to the next state. Here, because our state space is equivalent to the node space, it follows that the only actions to get from one node to another must be through an edge. Therefore, for a given state s on node v_i, we can define the set of actions, A_s as the set of outgoing edges from that node, $A_s = \{a_1, a_2, .., a_k\} = \{(v_i, v_j) | v_j \in V, (v_i, v_j) \quad exists\}$. In this manner, we can represent the action space A, as the set containing every action for every node. $A = \{A_{s_1}, A_{s_2}, ..., A_{s_n}\}$.

2.3.3. Discount Rate γ

When deciding the optimally of an action, the agent will build a value out of the rewards it expects to receive in the future for other sequential actions. So the discount rate, $\gamma \in [0, 1]$ determines how later rewards should affect the current decision. For instance, if the discount rate is 1, then all future rewards will have the same impact on the decision as the current reward. And if it's 0 the algorithm will only consider the most immediate rewards and so will be impulsive.

2.3.4. Reward Function R

Immediately after taking an action a reward is given to the agent which describes the strength of the choice. This creates the optimization problem to the algorithm. In particular we can define the reward as a discrete function that takes, $R : S \times A_s \times S \to \mathrm{R}$. Or, given a state s, an action a and a proposed new state, s', corresponding to the action, $R_a(s, s') \in \mathcal{R}$. (Xia et al., 2019). This is the motivational element of the reinforcement learning algorithm;

such that, between trials the agent will store the experienced rewards from the previous set of actions and optimize its next approach, or preferentially select routes that result in greater rewards. Therefore, we desire the reward based optimal route to coincide with the shortest-paths. And so, we allow the reward for each state action state pair to be proportional to the inverse of the expected edge weight, so cheaper/shorter edges offer a greater reward.

2.3.5. Optimal Policy, π^*

Returning to the goal of reinforcement learning, we want to find a policy to maximize the cumulative rewards for any state. And so given a state S and a policy π we can define the value function as the expectation following π of the sum of expected rewards for all the next decisions for each step after S. $V^{\pi}(s) = \mathrm{E}[\sum_{k=0}^{\infty} \gamma^t r_{t+k} | s_t = s]$, for some $s \in S$, and where t is the current event of the increasing order of events for a given trial, $t = 1, 2, 3, ...$ And so the optimal policy, is the π that maximizes this function for every state S. $\pi^* = \max_{\pi} V^{\pi}(s) = \max \mathrm{E}[\sum_{k=0}^{\infty} \gamma^t r_{t+k} | s_t = s]$. (Xia et al., 2019). Critically, here, the agents movements are completely dictated by the policy, so in theory, no new information about the graph can be collected after the first policy is created. Which can lead to a high degree of error.

2.3.6. State Transition Function, ρ

So now as a resolution, we get to the final component of the Markov decision tuple, the state transition function, $\rho : S \times A_s \times S \rightarrow [0, 1]$, (Xia et al., 2019). which given a current state, s, and action dictated by a policy, and a proposed new state, s' this function will give a binary probability distribute that the agent will follow the policy and move to the purposed s' state, or will move randomly. And so tuning ρ to find a balance between exploring the environment/ moving randomly and listening to the proposed policy is critical to the functionality of the algorithm.

2.4. Exploration vs Exploitation

The argument between exploration and exploitation is at the center of reinforcement learning. In particular, exploitation describes the against use of past knowledge to inform decision, so under a particular policy, the agent will always move by the purposed highest value. Whether or not the policy is optimal. And so this aspect favors convergence rate over accuracy. Whereas, exploration, the agent ignores the current policy and takes random actions or moves randomly throughout the graph. This results in many new possible rewards and paths being discovered but the convergence rate to find the minimum path is much much slower. And so in devising the hyper parameters for the algorithm, we must choose a good balance between exploitation and exploration.

2.4.1. Epsilon Greedy Strategy

A good and traditional way to do this is with an epsilon greedy strategy. In which a constant ϵ is set to some real number between 0 and 1. Then the probability that the agent moves randomly is set to ϵ, then the probability that the agent follows the policy is $1 - \epsilon$.
Additionally, we define the greedy strategy to reduce epsilon after each episode. We define a learning rate δ between 0 and 1 much smaller than ϵ, then after an episode, m a new $\epsilon' = \epsilon - m\delta$. This ensures that at the beginning of the learning process the agent prefers exploration but for a sufficiently large m the agent will start to follow the policy as it becomes optimized.

2.5. Q-Learning

Now to solidify the decision-making process in our model, we implement a Q-learning strategy where a matrix or q-table is constructed of the the possible state action pairs and the expected rewards for each. More specifically, we take each row to be a unique state, then each column to be a particular action associated with that state. The q-table is initialized as a $n \times k$ matrix, where k is the maximum number of outgoing edge for a single node across all nodes. Then, each of the k actions for each state are assigned of increasing order by the outgoing node. Furthermore, unavailable actions are given an untenable value, $-\infty$. Finally, each element is set to the quality of an action, or the expected reward. Then at each state, the agent only needs to pick the largest element in the corresponding row and preform the associated action. Then update the q-table with the new experienced rewards. And so, the optimization of the q-table results in a path between states that leads to the shortest path between the source and destination.

3. Results & Conclusions

Finding efficient paths between nodes on a graph is critical to many modern data applications. Here, we discuss a machine learning method using reinforcement and Q-learning techniques to find an adaptive algorithm that is capable of solving the bottleneck problem created by heuristic shortest path algorithms. Critically, by constructing each policy using a q-table, it may also able to reference multiple sub-optimal paths and so in the event that the length of a path changes, or the traffic on a route alters the standard pdf, the agent should also be able to adapt and quickly find new paths. Indicating that, after training, the algorithm is likely to be able to find new minimum paths faster than heuristic methods that need to fully re-calculate the path at every change in length. Unfortunately, at the time of writing our implementation of this Q-learning algorithm is inconclusive and unable to find the minimum path.

References

N. Bäuerle and U. Rieder. *Markov decision processes with applications to finance*. Springer Science & Business Media, 2011.

W. Ben-Ameur and E. Gourdin. Internet routing and related topology issues. *SIAM Journal on Discrete Mathematics*, 17(1):18–49, 2003.

E.-T. Gilad. Graph theory applications to gps networks. *GPS Solutions*, 5(1):31–38, 2001.

A. V. Goldberg and C. Harrelson. *Computing the shortest path: A search meets graph theory.*, volume 5. 2005.

K.-C. Tsai, Z. Zhuang, R. Lent, J. Wang, Q. Qi, L.-C. Wang, and Z. Han. Tensor-based reinforcement learning for network routing. *IEEE Journal of Selected Topics in Signal Processing*, 15(3):617–629, 2021. doi: 10.1109/JSTSP.2021.3055957.

W. Xia, C. Di, H. Guo, and S. Li. Reinforcement learning based stochastic shortest path finding in wireless sensor networks. *IEEE Access*, 7:157807–157817, 2019. doi: 10.1109/ACCESS.2019.2950055.

Supporting Materials

Python Q-Learning Implementation:

```
# -*- coding: utf-8 -*-
"""
Created on Sun May 30 21:55:04 2021

@author: IanSi
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import networkx as nx
import random
import time

"""
Initialize Graph Data set
"""
G = nx.gnm_random_graph(26,80,seed = 2021,directed = False)
destination = 25 #End of path

for i in G.nodes():
    G.nodes[i]['State'] = False
    G.nodes[i]['start'] = False
    G.nodes[i]['end'] = False

G.nodes[0]['start'] = True
G.nodes[24]['end'] = True


color_map = nx.get_node_attributes(G, name = "State")
for key in color_map:
        color_map[key] = "green"
color_map[0] = "red"
color_map[destination] = "red"


plt.figure(1)
colors = [color_map.get(node) for node in G.nodes()]
nx.draw_networkx(G,node_color = colors, node_size = 300, pos = nx.circular_layout(G),arrows = True, with_labels = True)


"""
Find Shortest Path using dijkstra's algorithm'
"""
Control_Path = nx.shortest_path(G, source=0, target=destination, weight=None, method='dijkstra')
for i in Control_Path:
    print(i)


"""
Initialize state and action space for Markov Decision Process (MDP)
"""
state_space = np.arange(nx.number_of_nodes(G))
#Fix action space
action_space = []

#Build Q-Table, is hard because action space does not have consistent rows...
maxActions = 0
for node in G:
    size = 0
    for i in nx.all_neighbors(G,node):
        size+=1
    if size > maxActions:
        maxActions = size

qtable = np.zeros((state_space.size, maxActions))

for node in G:
    size = 0
    for i in nx.all_neighbors(G,node):
        size+=1
    for j in range(size,maxActions,1):
        qtable[node][j] = -1

"""
Hyperparameters
"""
```

```
Total_episodes = 100
learning_rate = 0.8
max_steps = 50
gamma = 0.95
#Exploration Parameters
epsilon = 1
max_epsilon = 1
min_epsilon = 0.01
decay_rate = 0.01

"""
Q-learning Algorithm
"""
rewards = []

for episode in range(Total_episodes):
    Path = []
    state_prev = 0
    state = 0
    G.nodes[state]['State'] = True
    step = 0
    done = False
    total_rewards = 0

    for step in range(max_steps):
        G.nodes[state]['State'] = True
        color_map = nx.get_node_attributes(G, name = "State")
        for key in color_map:
            if color_map[key] == True:
                color_map[key] = "green"
            else:
                color_map[key] = "blue"
        #For current action (s) choose an action (a) by selecting a neighbor

        #Epsilon Splitting
        exp_exp_tradeoff = random.uniform(0, 1)

        if exp_exp_tradeoff > epsilon: #Exploitation
            QargMax = qtable[state][0]
            size = 0
            for i in nx.all_neighbors(G,state):
                if qtable[state][size] > QargMax:
                    QargMax = qtable[state][size]
                    new_state = i
                size+=1
            action = np.argmax(qtable[state,:])
        if exp_exp_tradeoff <= epsilon: #Exploration
            size = 0
            for i in nx.all_neighbors(G,state):
                size +=1
            edgeIndex = int(random.uniform(0, size))
            size = 0
            for i in nx.all_neighbors(G,state):
                if size == edgeIndex:
                    new_state = i
                    action = edgeIndex
                size += 1

        #Perform the action, record the reward and new state.
        #new_state, reward, done, info = env.step(action)

        reward = random.uniform(0, 100)

        if new_state == destination:
            done = True

        qtable[state, action] = qtable[state, action] + learning_rate
            * (reward + gamma * np.max(qtable[new_state, :]) - qtable[state, action])

        #Update Q-Function
        total_rewards += reward
        #Update state
        color_map[state] = "blue"
        Path.append(state)
        state = new_state
        #Check if destination has been reached
        if done == True:
            break
        epsilon = min_epsilon + (max_epsilon - min_epsilon)*np.exp(-decay_rate*episode)
        rewards.append(total_rewards)
```