Using Q-Learning to Determine the Optimal Routes Between Queen's Campus Buildings

School of Computing, Queen's University

Fall 2022

Introduction & Problem Formulation

For our final project we chose to build a reinforcement learning model that will optimize routes for Queen's University students going between locations on campus. To implement this we used data from Open Street Maps to collect all of the buildings on campus, stored as nodes, and every possible route between them, stored as the edges between nodes. In this graph implementation, the nodes are the states and the actions are the edges between them. We define the reward scheme as the time taken to take a path (edge) between two buildings (nodes). We then use a Q-learning method to train our model to learn the optimal routes between every building on campus. Once trained, our user is able to input a start and goal location and receive a visual map of the best path to take to get there.

State/Observation Space

The state space for the problem is discrete. The state space is defined by passing OSM a defined coordinate area and asking it to retrieve the nodes and ways in this space from their database. The area being used is Queen's campus, so these values are passed to OSM: [left = -76.5017059541, bottom = 44.2224495448, right = -76.4903552918, top = 44.2298231916]. There are only a finite number of nodes that exist in this area of OSM's map. The size of the state space is the number of nodes, which is 10732.

Action Space

The action space for our problem is discrete and includes all of the possible paths between each building on campus, which are the states. Therefore, within the bounding box we defined, we have 11437 unique actions.

Reward Scheme

The reward scheme for the problem encourages the agent to follow the shortest path from the start node to the goal node. The reward is the distance from the start node to the goal node, except this value is always negative or zero. The agent will receive a reward of 0 when it reaches the goal node. This can be shown as:

$$r = \begin{cases} 0, \textit{ for reaching a terminal state} \\ -d, \textit{ for reaching a node after traveling a distance } d \end{cases}$$

Methodology and Algorithm Description

Map Data Extraction

The first phase of our project implementation was data extraction. We used the Open Street Maps API to extract all of the buildings on campus and the paths between them. Open Street Maps (OSM) is a "collaborative project to create a free editable geographic database of the world"[1]. We used the Overpass API, a read-only API that allows us to select custom elements of OSM data through queries[2].

We used an OSM to networkx parser to import the OSM data and convert it into a networkx graph which removes all singular nodes from the data (see Figure 1). This ensured the agent would be moving in paths which are connected to the rest of the network and not disconnected.

[1] *OpenStreetMap*, https://www.openstreetmap.org/#map=2/71.3/-96.8.
[2] *Overpass API,* https://wiki.openstreetmap.org/wiki/Overpass_API#Introduction

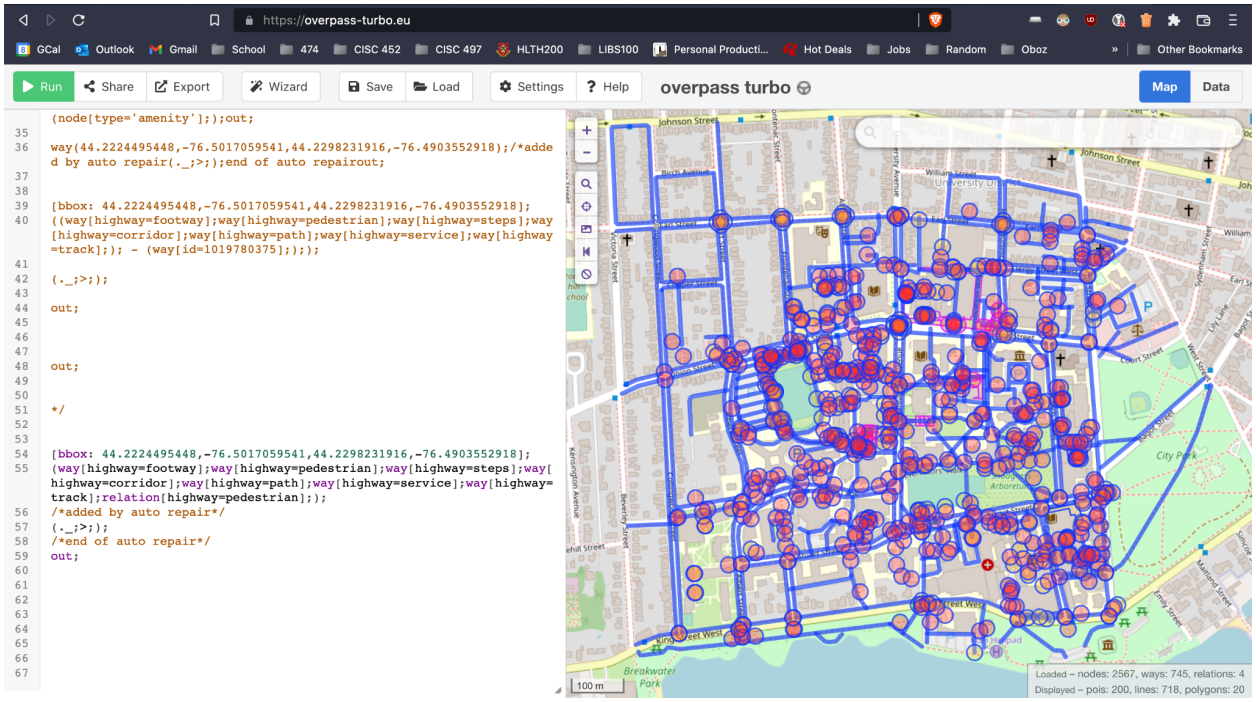

Figure 1: Extracted OSM Map Data

Implementing the Environment

We defined several classes that work together to create our environment. First we created “Node” and “Way” classes to define and retrieve the data within each. We represented nodes as tuples of latitude and longitude coordinates.

To create our graph of nodes and edges, we read and parsed OSM data to define it within an “OSM” class object. The OSM class is used to create a networkx graph that defines all the map data as Node and Way objects. Once initialized, it counts each time a node is used and builds a histogram that erases all the nodes that were not visited. This parses out nodes that will not contribute to our action space.

Finally we created a “CWorld” class that takes the OSM graph as input and defines start goal states. This contains the reward function and a function that takes a node as input and returns all of its neighboring nodes.

Using Q-Learning to Train Model

Since there may be several possible routes between two respective buildings, we define a Q-learning function to learn the optimal path between each pair of connected nodes in our environment. Q-learning algorithm is a model-free, off-policy method of reinforcement learning that finds the best course of action given the current state of the agent - this is an ideal method to approach solving our problem.
The Q-learning function takes the environment (“CWorld”), and alpha, gamma, epsilon and maximum number of episodes, and maximum number of steps as parameters as input. The Q-table is initialized by iterating through each node in the environment and setting the entries for all possible paths from each non-terminal node as 10. The Q-table entries for each possible path stemming from a terminal node are initialized as 0.

We define a get_next_state() function which takes a state (ie. a node) as input and using an epsilon-greedy policy, either 'exploits' (choosing the current node's path with maximum value) or 'explores' (choosing one of the current node's possible paths at random), returning the resulting action.

Using a train() function we loop for the max number of episodes, and for each state in the state space we call the get_next_state() function to return an action for the current state given our policy. We then take this action with the current state to get the next state and our reward, and update the Q-table entry Q(s,a) using the following update rule:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{current value}} \Big)}^{\text{temporal difference}}$$

Finally, we use an optimal_policy() function to return a list of the learned optimal policies for each node in our state space.

Training the Agent & Retrieving Optimal Routes

We define an Interface to define alpha, gamma and epsilon parameter values and create new instances of the CWorld (the environment) and the Q_Learning classes. After building the table we train the agent and return a list of the optimal policies. The user is able to use the interface to select a starting and goal location which are then used to retrieve the optimal route from the trained model.

Results

The results showed that the best hyperparameters and parameters for the model were an epsilon of 0.8, alpha of 0.2, gamma of 0.5, max steps set to 5000, and maximum episode set to 100. Using these values the agent was able to find optimal paths between start and goal nodes. When epsilon is increased the actions are more likely to be chosen exploratively.This was of great importance for our problem as our agent would never learn if we chose an epsilon of less than 0.5. When actions are chosen exploratively rather than exploitatively there isn't any value or logic guiding the choice, they are chosen completely randomly. When epsilon is decreased the actions are more likely to be chosen exploitatively, meaning they are decisions made based on what the agent has learned so far. When the alpha is increased the agent learns more from recent information rather than past information. When it is decreased the agent values past information more in its learning updates than recent information. Increasing the maximum episodes will let the agent learn for longer, but after a point this becomes redundant and can lead to overfitting. Additionally, our agent is not learning the right paths if there are thousands of steps in it's solution. Decreasing the maximum episodes can lead to poor training and an agent that hasn't properly learned how to navigate the environment to find the shortest path, producing poor results. We measured the performance of multiple testing agents by plotting the number of time steps in each episode (see Figure 2 below).

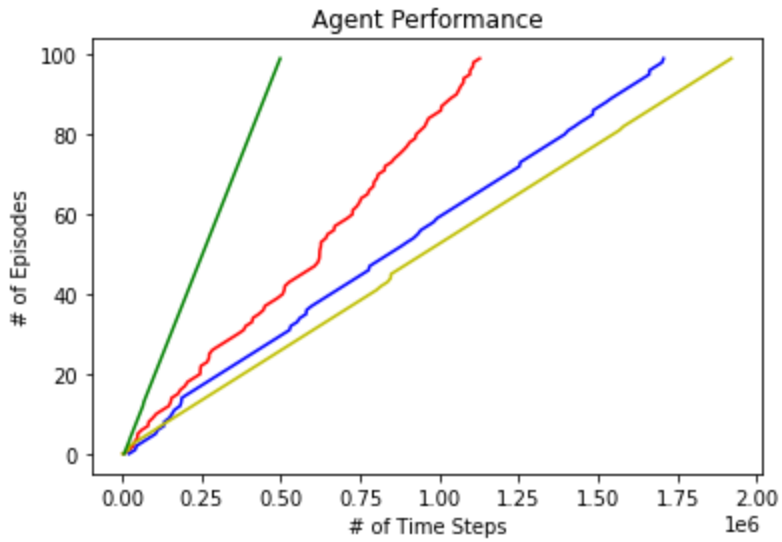

Figure 2: Agent Performance

Discussion

Challenges Faced

Many challenges were faced during implementation, but the biggest obstacles were representing the problem, working with map data, and a case where the agent oscillates between two nodes.

The first challenge was representing the problem. Two possible representations of the state space were considered: a gridworld where each cell represents a building, and a node-and-edge representation where each node is a building and each edge is a path. Although the nodes and edges representation felt less familiar than the gridworld, it resonated with applications of map building found during research, and worked seamlessly with the data which was retrieved from Open Street Maps. Once the nodes and edges representation of the map was chosen, the other problem encountered in the problem-representation stage was how to describe the values and policy. The initial problem asked the agent to find the shortest path between any two nodes. It was found that the solution would require each node to hold the best next action for each possible goal node. Furthermore, the policy would have to change for each pair of start nodes and goal nodes. This challenge steered the project away from representing the state space with an undirected graph to a directed graph, where there is a constant start node and a constant goal node.

A major challenge was working with the map data. Open Street Maps was determined to be the best source of node and way data because it was the easiest to use with python as there are several libraries available. There were difficulties determining the distance of a way, determining the ways connected to a node, and reducing the state and action spaces to make it easily navigable. The first problem was imperative to calculating accurate rewards and thus training the model properly. This was resolved with use of the haversine distance calculation, using the longitudes and latitudes of the given start and end nodes. This data was provided by the OSM data and thus was easily accessible. The second problem was solved with a library called networkx which was used to turn the data into a directed graph. The library has a function to find the neighbors. This was necessary to accurately find the resultant node after the

agent took a way. The networkx library was used again to solve the third problem. While turning the nodes and ways from the osm data into a directed graph with networkx, a curated selection of only ways that are roads could be added into the directed graph, as well as only certain data (i.e. the ids, longitudes, latitudes, distance of a way, and tags). Moreover, we originally used a Python Wrapper called Overpy. By specifying the bounding box we wanted to extract data from, we were able to narrow the area to the Queen's Campus. We also specified the type of nodes we wanted to extract; Open Street Maps contains data on many different features - in our case, we only wanted to look at Campus buildings so we filtered the nodes by those tagged 'building'. We soon figured out that there were many nodes and ways which did not connect to each other. This proved to be a problem as our agent kept running into the scenario of not being able to reach this goal. This problem was resolved by switching over to the networkx graph implementation which ensured that each node was connected correctly by parsing out nodes with no connections

Future Work

For future work it would be interesting to build on the reward function. For example, environmental circumstances like the time of day, weather, and traffic could have an effect. Getting historical and live data about the area being mapped and using it to consider the effects of different environmental factors on the time it would take to follow a certain path, rather than just the distance it is, would be beneficial to people in a hurry. It could drop the reward if it is predicted to take longer than expected without the environmental circumstances, and maybe raise the reward if it looks faster than expected. Furthermore, it'd be interesting if more ways were considered. For example, jaywalking could remove a large amount of time and distance in a path, but an increase in danger must be considered. Also, the mode of transportation would play a part in the time it takes to reach the goal and it'd be interesting to let the user choose this. Additionally, solving an undirected graph would be a future goal. This was part of the original problem and is something that would make the project useful. Lastly, producing an output visual of the map would be a very useful feature, unfortunately it was not feasible within this time period.