Machine Learning Engineer Nanodegree

Capstone: Building Running Routes with Reinforcement Learning

Frank Corrigan
October 4, 2017

Definition

Project Overview

Approximately 17 million individuals in the US (according to 2015 data from runningUSA.org) identify themselves as runners. These are people that have participated, at least once, in an organized running race. This cohort is continuously searching for recommendations on how to run faster, who to run with, and where to find more enjoyable runs. That final question is the topic of focus in this capstone project. A particularly common method of deciding where to run is to ask other runners. Recently, websites such as Strava or MapMyRun have made the answer to this question more shareable. Any runner can throw a GPS-enabled watch on their arm, go for a run, upload the data to their site of choice, and other runners and view what routes other runners are running most often. The most popular routes are regarded as the best routes to run.

However, not every runner enjoys the same scenery or can take the time to commute to those best run locations. It would be advantageous if we could find the best or optimal run according to personal taste directly from where we are staying or living. Since time is finite and exploration of the options for running routes seems infinite, machine learning can assist in determining these optimal paths. The problem of considering which route to take is not unique to runners and running paths - optimal path finding problems exist and have been studied in depth in a variety of fields including naval ship ocean traversal, automobile navigation through road networks, logistics (recall the infamous traveling salesman problem), and even in a related activity, cycling. An optimal path, depending upon its application, could mean the shortest path, the most fuel efficient path, or the fastest path. Each of these fields lends information for how to think about determining the best running route from an individual's current location.

Leigh M. Chinitz had a similar idea many years ago. In 2004, he registered a US patent on an algorithm that created closed loops for cyclers. The website that leverages this algorithm, RouteLoops.com, asks the user for a starting location and target mileage and creates a looped route the cycler (or walker) can follow. Additionally, there are some high level preferences you can select such as road selection type. This research is helpful in understanding a good method for creating loops from the same start and end location. However, the RouteLoops algorithm is explicit and doesn’t incorporate preferences such as “run past water” or “go past historical monuments.” RouteLoops.com also struggles with short distances since repeating at least part of the course may be mandatory. I should also note that MapMyRun.com also has a demo product that creates loops for runners, but in it’s current form (April 2017) it only works in Paris, France. You can upgrade to use it in other cities.

On a personal note, I've been a competitive runner for many years through high school, college, and ever since. When I moved up to Melrose, MA -- 10 miles outside Boston -- a few months ago I was delighted and overwhelmed by the numerous options of trails I could explore. I do love exploration when I run, but as I worked through

Udacity's Smartcab Reinforcement Learning project I began to realize that running a new trail system and finding the best loops is very similar to the q-learning algorithm I was creating to teach a car to drive. I wondered if my computer could help me find the best trails in a much timelier fashion than just running every day. From years of experience, I have good intuition for what constitutes a good trail or a good run. If I can map a reward system to attributes of the trail system, we can use simulation and reinforcement learning to determine the best routes even before lacing up our sneakers.

- http://www.runningusa.org/statistics
- http://users.iems.northwestern.edu/~dolira/dolinskaya_dissertation.pdf
- https://en.wikipedia.org/wiki/Travelling_salesman_problem
- http://patft.uspto.gov/netacgi/nph-Parser?u=%2Fnetahtml%2Fsrchnum.htm&Sect1=PTO1&Sect2=HITOFF&p=1&r=1&l=50&f=G&d=PALL&s1=7162363.PN.&OS=PN/7162363&RS=PN/7162363

Problem Statement

The problem statement in it's simplest question form is "tell me the best (or optimal) running route from my current location." A runner wants to run the best runs as soon as possible, but it takes time to learn where they are and what your surroundings have to offer. Where are the trails or bike paths that will allow me to enjoy nature or avoid intersections and getting hit by cars? Where are the enormous uphills that I'd like to avoid (or attack) today? Are there historical structures or tourist attractions that I'd like to run past and see in this area? Is the neighborhood safe for me to run through? Are there restrooms or water fountains I can pass along the way? Each one of these questions, in total, is a piece of this problem.

A 10 mile run could have dozens of different split point decisions that need to be made. If a runner ran a trail system 1000 times, they'd be able to pinpoint the most rewarding runs given the characteristics of that area. However, 1000 runs could take several years! The goal is to speed up this learning process for each individual runner. Applying reinforcement learning to this problem will allow the runner to know the optimal decision at each split point in order to maximize the utility of a running loop based on a desired target mileage number. We can measure that by A) plotting on a map what the algorithm suggests as the best path and B) inspecting each decision at different points with knowledge of what lies ahead in that best path. If this works for me -- in the MIddlesex Fells Reservation -- it should work anywhere we can obtain map data for.

The optimal route -- for this particular solution -- will create a loop (start and end at same location) and maximize miles on "trails" (which can include hiking trails, running paths, or bike paths) rather than roads, include picturesque water views, and incorporate the highest number of historical or natural landmarks possible. It should be noted that a true loop is not required, although nice. A figure 8 is also desirable as is a loop where you “run to the reservoir, run around it, and run back the same way you came.” Balancing other variables will result in alternative solutions to this problem. For instance, if an individual is recovering from a hamstring injury they will want to avoid hills and as such minimize elevation gain to yield the optimal route. This implementation will focus on mileage on trails, aesthetically pleasing or educational surroundings.

The algorithm under development will include 2 user inputs. A starting location and a target mileage for the run. Simultaneously, we'll use a reinforcement learning algorithm to determine the best route for the target mileage. Ultimately, you'll have a policy table for each possible move that will still loop you from the same location and route your run through the best places the area has to offer (+trails, +views, +history). This entirely answers the question, if I'm at location X on this 5 mile run, which direction should I go next in order to maximize utility[1]? We'll be able to look at the optimal policy the algorithm determines, look at the path and its contents on a map, and determine if the algorithm has learned what we would consider the best policy.

A successful implementation will yield a running route that is within one-quarter mile of the target mileage for the run and will include the maximum number of waterfront paths and historic landmarks given the vicinity of the user's location. This may be challenging, if not impossible, in some cases. For example, the desired start location may be in Nebraska where there is only a single road in two directions for 10+ with no trails, no water, and no historic landmarks in the area. Since success in that instance would not be achievable, this implementation will narrow the geographic boundaries considered for training to a portion of Malden, Melrose, and Winchester, Massachusetts the majority of which is called the Middlesex Fells Reservation. A successful algorithm will have the ability to render the best (optimal) running route from most any location.

Metrics

Until we have a computer-human feedback loop in place where the algorithm suggests a loop, the runner runs it, and they rate how awesome the route was we won't have the perfect data to measure (and improve) the success of the algorithm. Instead, to start, we'll consider 2 criteria for the success of the algorithm.

In the proposal statement for this project, I suggested the time taken to compute the optimal path should not exceed 30 seconds. To quote myself, "I don't even want to wait more than 30 seconds for my Garmin (GPS watch) to find a satellite connection and most runners won't want to wait more than a few seconds for a suggested run… they'll just go." After months of testing, this time limit seems foolish the way I originally envisioned. When considering the user (customer) we'll have to take an alternative approach to calculating the optimal path at the time of use (perhaps another algorithm is needed to predict where a user will want to start/end their run from). In the dataset used for this project there are 15,000 nodes (15K decision points -- see visualization in Section: Analysis - Exploratory Visualization). While the combinations for how those nodes can be connected on a loop is finite, it certainly exceeds our ability to simulate and learn optimal path policy in 30 seconds[2]. In the results below, I've included the time it takes for our algorithm to converge based on a varying mileage parameter (as well as my understanding of convergence in this problem).

The second evaluation metric is the quality of the run. For this, I will use myself since I am familiar with the Fells (I live in Melrose, MA next to the Fells and have run there every day for the past year) and I will be able to give a subjective assessment of how good the route is relative to other routes I know I could take. I will list criteria that define a "good run" below in the benchmark section. I will also test a start point where I grew up (in New York) since I have run countless miles from that start location as well. Despite its unscientific nature, this will give a better assessment of the quality of the suggested run than anything else available.

Analysis

[1] Utility being used as a measure of usefulness to the runner.

[2] Udacity Reviewer #1 recognized that this is heavily reliant on the available hardware. In bullet #5 of the Implementation section below I outline how I've implemented this project in order to speed things up when no longer testing on a single processor.

Data Exploration

The dataset for this project will come from OpenStreetMaps.org. This open source geospatial data can be downloaded in .OSM format, which is very similar to XML. Further, there are many good resources on how to extract and manipulate these files. The map for this project will include parts of Malden, Melrose, and Winchester, MA. which is primarily the Middlesex Fells Reservation area. Technically, the bounds include `<bounds minlat="42.4251000" minlon="-71.1318000" maxlat="42.4702000" maxlon="-71.0699000"/>`.

This data includes both nodes and ways. Nodes -- which include latitude and longitude coordinates -- identify points on a map and ways -- which include collections of nodes -- identify links between nodes. Nodes and ways are tagged with different types of identifiers. For example, a node may be tagged with v="traffic_signal" meaning there is a traffic light at those coordinates. Waypoints have significantly more tags. One very common tag is k="name" usually accompanied by a street name. The starting location for this project will be Stone Place in Melrose, MA - where I currently reside. Three other tags that will be leveraged to assign rewards to different ways are

1. "highway" equals "path" or "footpath" which will indicate that the way is a running path prefered over the road.
2. "tourism" equals "alpine_hut", "wilderness_hut", or "viewpoint" which indicates either natural-setting shelter (usually in cool places) or a lookout where you can see something neat (in this case it will be a skyline view of Boston). This tag also includes "artwork" (it's always neat to run by a Banksy).
3. "historic" equals anything usually means something cool. Count it.
4. "natural" equals anything. There are a variety of tags to tell us about the way here including "peak" (is it a mountain peak) and "water" or "spring" (does it go past a body of water).

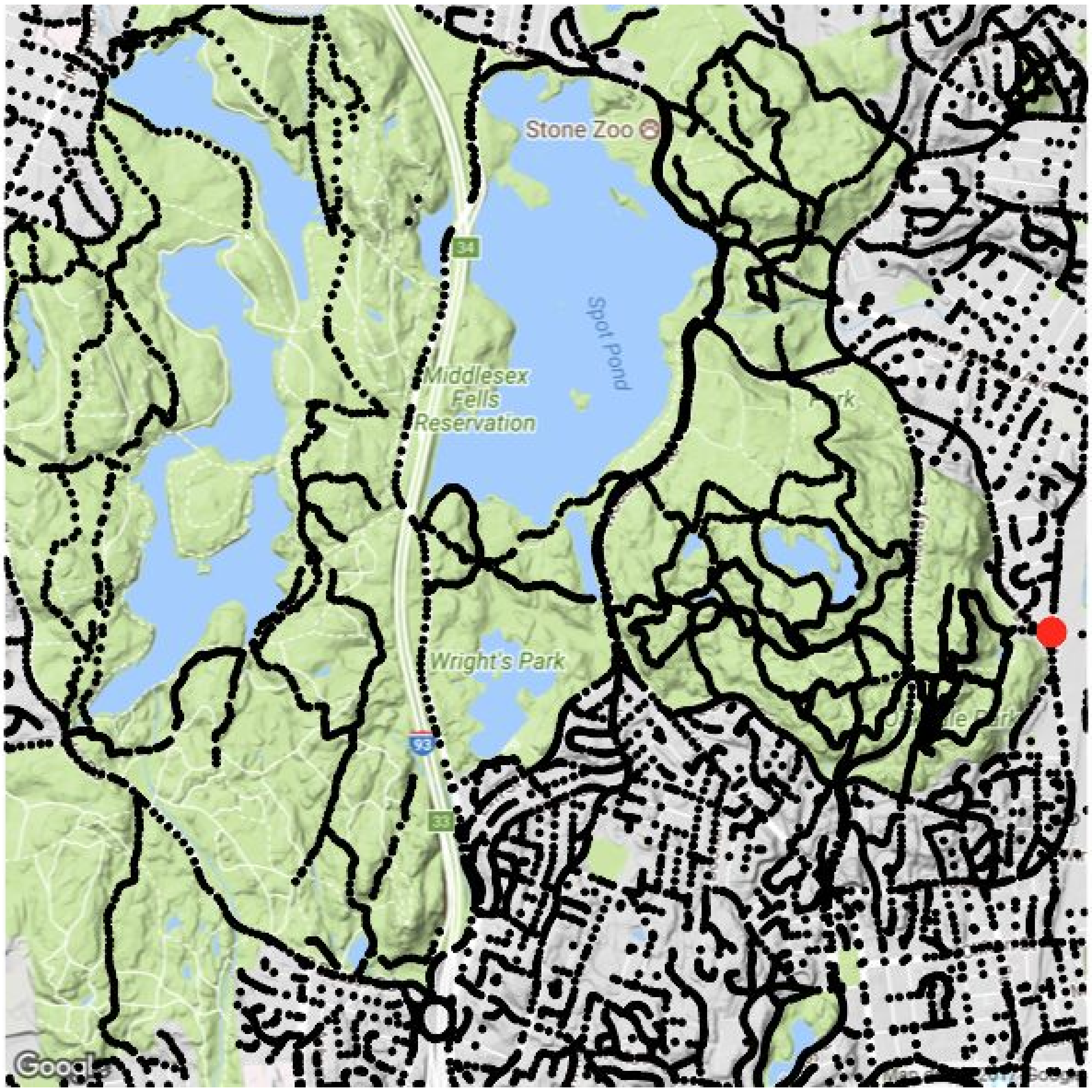
The world is vast, and so is the data that represents it. This file is approximately 28MBs and contains 127,443 nodes. This contains a lot more than roads and paths; it also includes data points for building structures too. After filtering out only roads and paths (which would be ways available for running) using OSMFILTER the file is 5.6MB with *only* 24860 nodes. Still huge. I explain further attempts to scale down the data in Section: Methodology - Data Preprocessing, but this is the base dataset used.

This dataset is built by dozens of individual contributors. While the dataset used in this project is very well tagged -- lots of people hike this trail system -- some more remote areas (and even trails in my home suburban town) may not have the same quality of data. A lack of tags will severely impact the quality of the results - garbage in, garbage out.

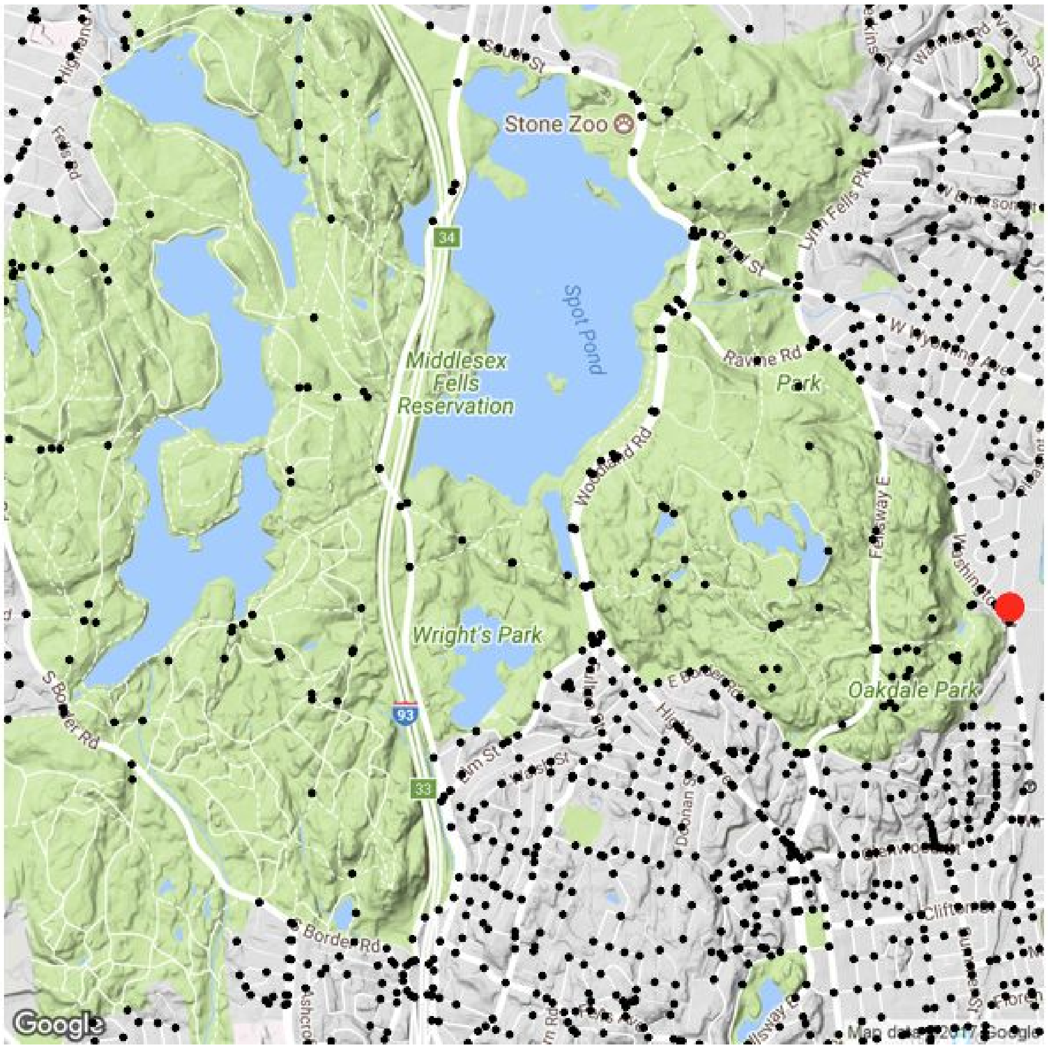
- http://www.openstreetmap.org
- http://www.openstreetmap.org/#map=13/42.4530/-71.1269
- http://wiki.openstreetmap.org/wiki/Hiking

Exploratory Visualization

The map below shows the region where we are training our algorithm. Each black dot represents a data point (a node). The red dot is the starting location. Although the area is well tagged, you can see that some paths in the Fells are still “unchartered”. In the lower left hand corner you can see the white line trails that are not overlapped by black dots.



The above map shows all nodes. This is too many nodes to iterate over as it makes convergence near impossible. In order to reduce the size of the dataset we can capture the same information by filtering to only intersections (decision points). The below map shows intersections only -- considerably less dots than the map above.



Algorithms and Techniques

This implementation builds a q-learning algorithm. Here is a screenshot from Wikipedia showing the basic algorithm:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

That equation ultimately gives us a policy for what we should do at each node. Imagine you are at a specific point on a run. In order to decide where to go next you assess your environment -- where did I just come from, how much further do I want to go, what are my options and where will they take me relative to my start location -- and make a decision. However, there is one last component of that decision. You would want to know not only the immediate result of your decision but also how that decision impacts your future movements. This equation, updating the policy of a specific movement over many iterations, allows us to incorporate both immediate and future information at the same time. To make this more concrete, imagine you are at a crossroads and you have 2 options. Each of those options are x miles away from the start. Option 1 will eventually lead to a busy road with many traffic lights and lots of cars (not good). Option 2 will eventually lead to a beautiful waterfall with soft ground surrounded by pine trees (good). That's future information. The iteration cycle will help the algorithm incorporate that future information into the decision it makes about the current movement and eventually guide us away from the road and to that waterfall.

Below defines the major parameters of the model.

- **States & Rewards**. The reward of a state is determined by a few different variables interacting at the same time. A state describes where the runner is, where the runner was before they moved to their current position, where they are going next, and the duration (a binary variable telling if the running is halfway through the run or not). The foundation of the reward is the feature of the way -- is it on trail, does it have a nice scenic view, does it go past water, or does it go past a historic landmark. The tags in the data tell us this information. The reward is adjusted to further teach the algorithm how to run. First, if the duration is low (meaning the running is less than halfway through the run) and the runner moves to the further point from the start location (see Determining Distance below), the reward goes up. Similarly, if the duration is high and the runner moves to the closest point from the start location, the reward goes up. Second, if the runner goes backwards, they incur a large deduction. This is all learned by leveraging the geospatial component of our dataset. Although we don't mind if we repeat part of the course, we certainly don't want to turn around -- continuous, fluid, motion is always a goal for runners.
- **Determining Distance**. We use two distance calculations. First, we implement a simple trigonometric calculation (as the crow flies) to get a sense of where nodes are relative to the start location which is used to determine duration. Second, we implement a local routing service[3] to inform the algorithm how far a specific node is from the start location (route wise). This second component helps us decide, of the next options I can take on my run, which one is closest and which one is furthest from where I started.
- **Alpha, Gamma, and Epsilon**. There are 3 parameters that manipulate how this algorithm learns. This environment is not deterministic -- meaning the reward structure changes from run to run depending upon the path selection -- and as such, we use an alpha 0.8 in the algorithm. This allows us to learn incrementally the information learned during a movement into our optimal policy (note that decaying the alpha with a variety of functions did not yield superior results). We use a gamma of 0.3 because the future matters. However, future Q-values should nudge the algorithm and not direct it. Finally epsilon, we decay this parameter over the length of the simulation which encourages the algorithm to explore a lot in the earlier stages but exploit the knowledge gained as it gets closer to the conclusion of the simulation. Without the decay, more frequently than not, you end up with "gaps" in the optimal policy -- it doesn't know what to do at a certain location.

Benchmark

Since no single source will serve as a benchmark of the output of this model (according to my research this is a new application of q-learning), I will leverage an existing tool, RouteLoops.com, and my pseudo-professional experience as a competitive runner (just in case you want to check: https://www.athlinks.com/athletes/122932884).

[3] http://project-osrm.org/

RouteLoops will benchmark time to calculate a route. While it doesn't have to be as fast as RouteLoops, it will have to be within "range" of the RouteLoop builder. RouteLoops can build random loops within seconds hitting target mileages requested by the user. This algorithm should perform similarly while providing a more personalized, optimal route. To make this more clear, my original expectation was to build a route within 30 seconds and RouteLoops.com dictates we should build loops in under 10 seconds. There is a huge caveat here. Computation time depends heavily on hardware. I am building and testing this model on my laptop (3.1 GHz Intel Core i7, 16 GB 1867 MHz DDR3), while RouteLoops is probably running their software on multiple servers. So, my expectation is that I can build a qualitatively acceptable route at a rate of 60 seconds per mile[4].

We can compare quality of the algorithm by observing decisions it makes. There are a few ways to assess a "good route":

- The route starts and ends at the same location.
- The route incorporates known surrounding reward-granting features.
    - For each route, we will count the average number of legs per mile that have reward-granting features which, all else being equal, will allow us to assess if one route is better than another.
    - Additionally, I will show a list of the types of features that were encountered on that particular route.
- The route creates a loop of some form, for example:
    - True loop where runner starts and ends at same location and repeats no segments.
    - Figure eight loop where the runner starts and ends at same location and forms a figure eight.
    - Pseudo loop where runner runs to the lake, loops the lake, and returns back home on the same path traversed up to the lake.
- The route mileage is within one quarter mile of the target mileage (+/- 0.25 miles is negligible).

Out and back loops are OK, and in real life are extremely practical when trying to hit a target mileage. I did many runs starting at Iona College and there were 2 core trail runs. One was out to a lake, loop the lake, and back the same way you came (Nature Study). The other was simply an out and back (Leatherstocking) -- unless you wanted to do a 14 mile run (which was typically reserved for Sundays) the only option if you wanted to hit that particular bit of trail was an out and back. But we are attempting to leverage data so we can run more interesting routes and avoid doing so many out and backs. All of these criteria are the equivalent of an accuracy measurement in a more traditional machine learning problem.

Methodology

Data Preprocessing

Fundamentally, the success of this project is attributable to data preprocessing. Without understanding and scaling down the right input dataset, the algorithm results would never be sufficient.

[4] The way I've set this up is each policy table I build is built with 100 iterations. The variable component is how many policy tables we build and ensemble together. For instance, when building a 1 mile loop I will run the algorithm 10 times resulting in 10 policy tables and then for each key in our table we find the mean value from all 10 tables.

Connecting the q-learning algorithm with the osm data is really about trying to parse the xml file into a useable format that the algorithm can consume. We needed a few items...

- **Neighbor nodes**. For every single node visited, we needed to know it's relative position on the way (aka path aka route) so the algorithm could decide where it came from and where to go next. We wrote a function that takes a node as the input and returns its surrounding "neighbor" nodes.
- **Geospatial coordinates**. Each node is assigned lat/lon coordinates. Each node is part of a way. In order to figure out the distance of a node from the last and future position, as well as its distance from the start location, we needed to pull out the coordinates.
- **Obtaining Rewards**. For every single node visited, we needed the foundational reward. In the dataset, each way is comprised of a list of nodes and has a series of tags. For example, k="highway" v="footway" indicates that the way is a path for walking (not vehicles). The algorithm needed to know the reward for each.

When you download map data from openstreetmap.org, you start with A LOT of information. Not all of this information is relevant for this project. The initial data file that was downloaded was 28MBs with 120K+ nodes. Not all of this is "runnable." You are not going to run through a building, you are probably not going to run down a service street or driveway, and you don't need to know the relationship between a street and a particular building. I needed to strip that away to make the algorithm work and work faster. So I used OsmFilter to discard a lot of unuseful information from the dataset. For the record, it took me months to realize this was the case and as a result I spend many hours trying to tweak the algorithm to unuseful data. The lesson here -- one that I already knew but needed to relearn -- is that you should really know your data before trying to build a model on it. This is the command I used in my terminal:

```
./osmfilter map --keep="highway= and name=" --drop="highway=service">streets_to_run.osm
```

This tells the tool to parse the file keeping only streets with a name and removing any street designated a service road (so I don't end up running around parking lots all day). I document this process in the section below Section: Methodology - Implementation.

This brings the file from 28MBs with 120K+ nodes to 3.5MBs with 15K nodes (see the visual above to get a sense of what this means). While this is still big and requires a large amount of computation time it certainly helped yield reasonable results.

This is a huge step, however, it's not enough. Before running the data points through the algorithm I also removed all nodes that aren't decisions points -- intersections. You can see this visually in the Exploratory Visualization section above and I describe how I do it in the Implementation section below.

Implementation

In order to complete this project, 5 main steps were taken:

1. **Obtain the data**. The data was downloaded from openstreetmap.org (https://www.openstreetmap.org/export#map=15/42.4440/-71.0719) for boundaries surrounding 1000 Stone Place in Melrose, MA.
2. **Filter the data**. As noted above an .osm file contains way more information than we need for this project

which drives up the computation time and produces poor results if we use the whole file. In order to scale the file down to only what we need the following steps were taken:

a. *Install OsmFilter*. A few different methods can be found on the OsmFilter Wiki (http://wiki.openstreetmap.org/wiki/Osmfilter), but I used the command line install: `wget -O - http://m.m.i24.cc/osmfilter.c |cc -x c - -O3 -o osmfilter`
b. Place the .osm file from step 1 in the same directory where you installed OsmFilter. The following terminal command will remove all data except roads and paths we deem acceptable for running and output to a new file: `/osmfilter map --keep="highway= and name=" --drop="highway=service">streets_to_run.osm.` You now have the data file upon which you'll pull data called streets_to_run.osm.
c. As noted above, filtering the osm data to only roads and paths was not sufficient. I also needed to remove nodes that weren't decision points aka intersections. To do this, I wrote a function called extract_intersections which parses the osm file and looks for nodes that occur in more than 2 ways. This would indicate an intersection. From that I create a list of intersection nodes and use that to cross reference when writing other functions such as get_next_nodes that identifies the neighbor nodes of a target node. Warning: this means you can only start your run from an intersection (to be improved in future iterations).

3. **Set up routing service**. When we want to know the distance between a node and the start location, we need to understand the route via street path, not as the crow flies. For this, we need to call a routing service. Openstreetmap.org provides an open source service that I downloaded, ran locally, and was able to call thousands upon thousands of times without worrying about API limitations. In order to do that: note - i was parsing the json incorrectly for a few months..
   a. I took this course to understand how APIs work: https://www.udemy.com/rest-api-flask-and-python/learn/v4/t/lecture/5960174?start=0
   b. Clone the OSRM repository: https://github.com/Project-OSRM/osrm-backend/blob/master/docs/http.md
   c. Cd into that directory and get the map where you want to call routes, in my case this is in the state of MA in the United States: **`wget http://download.geofabrik.de/north-america/us/massachusetts-latest.osm.pbf`**. You can obtain other maps from https://www.geofabrik.de/. Now we need to build the routing system:
   d. Install Docker. https://www.docker.com/. You can install this anywhere on your machine you please.
   e. Build the routing system. Make sure you are in the OSRM directory and use the following commands (note that we use the /opt/foot.lua option so that the routes we build are walking based - otherwise it would route us around the trails rather than through the trails.
      i. `docker run -t -v $(pwd):/data osrm/osrm-backend osrm-extract -p /opt/foot.lua /data/massachusetts-latest.osm.pbf`
      ii. `docker run -t -v $(pwd):/data osrm/osrm-backend osrm-contract /data/massachusetts-latest.osrm`
   f. Use this command to run the routing service locally -- when we see the message in the second bullet we are ready to roll:
      i. `docker run -t -i -p 5000:5000 -v $(pwd):/data osrm/osrm-backend osrm-routed /data/massachusetts-latest.osrm`

    ii. [info] running and waiting for requests

  g. Now we can use python request package to call the API and parse the returned data:

    i. `url = 'http://127.0.0.1:5000/route/v1/walking/{};{}?steps=true'.format(end_coords, start_coords)`

4. **Build algorithm.** The below steps detail the logic for the algorithm:
  a. Prior to starting simulation decide the number of miles and starting location. Then start the simulation.
  b. Determine location
  c. Determine next nodes that are available to run to
  d. If exploring, chose a random node. If exploiting, choose next node with highest score (i.e. highest q-value) in our policy table. In latter case, if no next node has a highest score (i.e. they are all 0), choose random node.
  e. Move to next node and determine reward of that move:
    i. If chosen path is on trail, passes water, or passes a historic monument; increase reward.
    ii. If chosen node is returning to the node that the runner immediately came from; decrement reward.
    iii. If the duration is high (in latter half of the run) and node chosen is the closest possible option to the start location; increase reward.
    iv. If the duration is high (in latter half of the run) and node chosen is the starting location; increase reward dramatically.
  f. Get max q-value of all possible nodes following the chosen node. For example, if A was the location and point B was the node chose and the next options from B are to points C and D, we need to know the higher of the two q-values of moving from B-C and B-D.
  g. Update policy table using this equation:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

  h. Set current location to the node chosen and repeat from Step B until runner returns to the starting location.
  i. Repeat Steps B - H until algorithm converges.

5. **Test algorithm, revise parameters, and ensemble the results.** At this point you have a basic working model that allows us to run many iterations while tweaking parameters. Additionally, since there is randomness at play I run the algorithm several times and obtain multiple policy tables for the same parameters. Then, I average the results of those tables. This method A) allows me to control for instances where pure random chance produced a poor route and B) will allow me to run the algorithm much faster when more robust hardware is being utilized[5]. Let me offer a real life analogy. Imagine a runner is attempting to learn a trail system originating at origin A. She does 25 runs over the course of 50 days. Perhaps she's learned the optimal route. If she does 100 runs (75 additional runs and 75 additional days) she will surely learn the best route. But, imagine instead that this runners, on day 26 meets up with 3

[5] Instead of doing 1,000 iterations resulting in a single policy table (which might take 10 minutes), I can run 100 iterations 10 times on 10 different machines and average the results together (which might take 1 minute to run the algorithm and another few seconds to aggregate the policy tables).

other runners who have also done 25 runs originating at point A. They can combine their knowledge of the trail system to learn the optimal run on day 26 instead of them all waiting another 75 days to learn the optimal policy. Effectively, we've reduced the time to learn the optimal policy by about 75%.

With the base version in place, we were ready to refine parameters and hone in the usefulness of the results which is discussed in the next section.

Additional:

To test New York location follow these steps:

1. Prep the OSM file to iterate over
   a. Download map from https://www.openstreetmap.org/export#map=14/40.8720/-73.3146 - might need to use overpass API
   b. Filter out building structure nodes and other crap with osmfilter: `./osmfilter raw_ny_osm --keep="highway= and name=" --drop="highway=service"> ny_streets.osm`
2. Prep geofabrik.pbf to use the open source routing machine
   a. `wget http://download.geofabrik.de/north-america/us/new-york-latest.osm.pbf`
   b. `docker run -t -v $(pwd):/data osrm/osrm-backend osrm-extract -p /opt/foot.lua /data/new-york-latest.osm.pbf`
   c. `docker run -t -v $(pwd):/data osrm/osrm-backend osrm-contract /data/new-york-latest.osrm`
   d. `docker run -t -i -p 5000:5000 -v $(pwd):/data osrm/osrm-backend osrm-routed /data/new-york-latest.osrm`
3. Fire up the osrm calling the NY map and then run the engine.py file using ny_streets.osm

We will use the results of this in Section IV. Results - Model Evaluation and Validation

Refinement

Using two start locations as testing points and a variety of target mileages, the algorithm performs OK as shown down below in the results. This is certainly only an initial solution to the problem with the caveat that the calculation time is not within goal. In order to determine how many iterations are required to achieve an acceptable result and as a consequence how much time is required to run those iterations, I experimented with a 1 mile run and varied the number of tables built in the simulation (remember, each policy table is built on 100 iterations and then averaged (mean) together to obtain a master policy table). The below table summarizes the findings of this experiment.

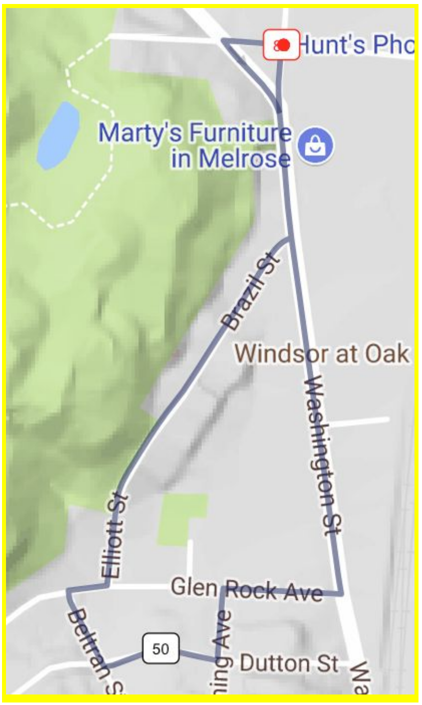
Note: at times, we encounter "Couldn't Determine Policy[6] if trials are too low.

# of Iterations	# of Tables	Time Elapsed	Acceptable Results[7]
100	2	1 min, 55 sec	No. Loop not in trail. Miles = 0.98

[6] Without enough trials, there is a possibility that the algorithm doesn't create a policy for a particular movement. In the event that the run generator (function that creates best run based on optimal policy table) needs information about a movement that doesn't exist in our policy table, the generator will fail.
[7] Acceptable Results means that the mileage is close to target mileage and the run forms some type of loop.

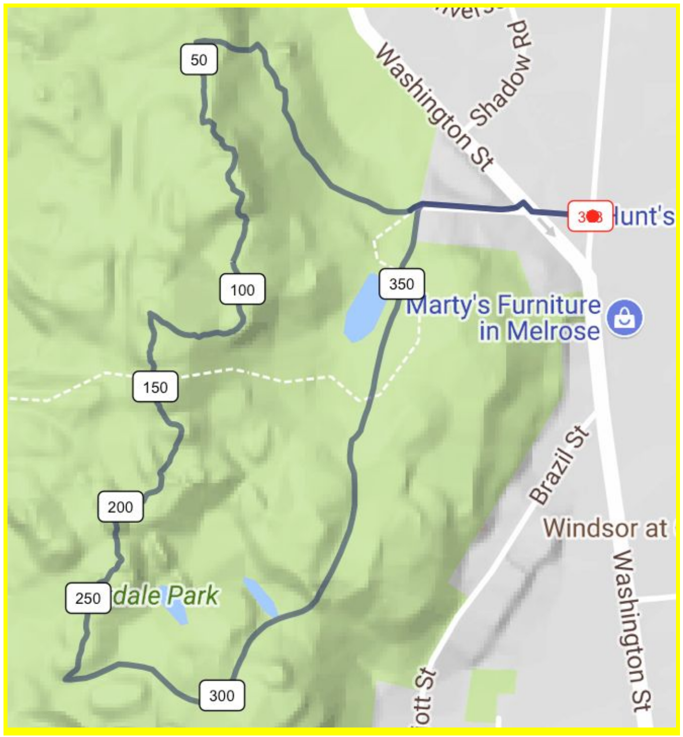
100	4	4 min, 13 sec	No. Loop in trail. Miles = 1.36
100	6	5 min, 3 sec	No. Loop not in trail. Miles = 1.26
100	8	7 min, 54 sec	Yes. Loop in trail. Miles = 1.00
100	10	10 min, 15 sec	No. Loop not in trail. Miles = 1.07
100	12	10 min, 54 sec	Yes. Loop in trail. Miles = 1.00

100 trials per Iteration, 2 Tables:



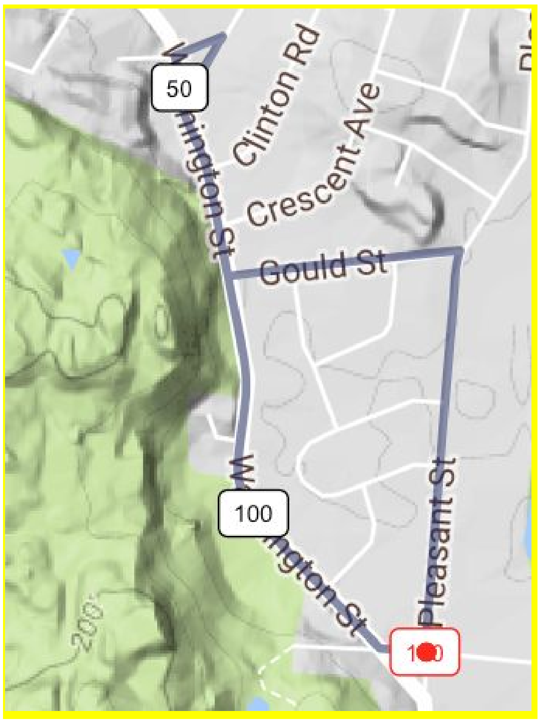
Unacceptable. Not a terrible start, but doesn't find the trail. Mileage = 0.98 miles.

100 trials per Iteration, 4 Tables:



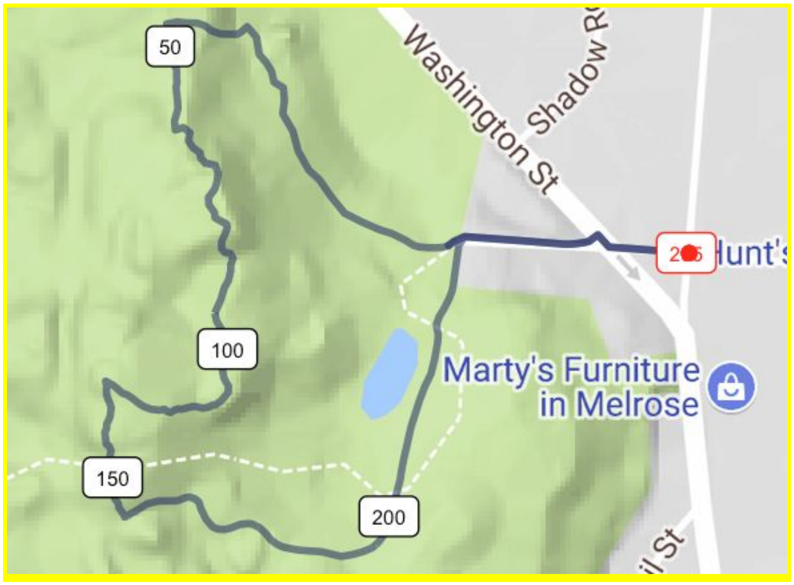
Not the mileage we want at 1.36 miles, but this is a very good loop in the trails.

100 trials per Iteration, 6 Tables:



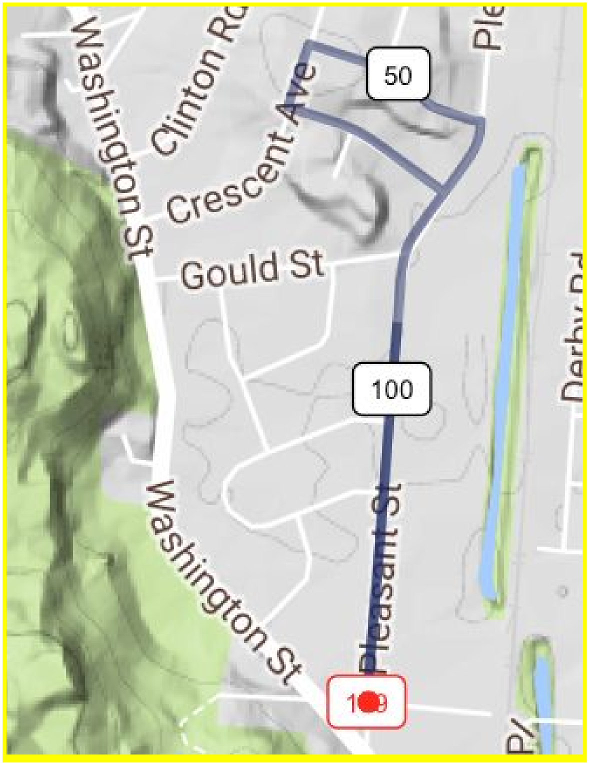
Ok. It does create loop-ish run, but doesn't find the trail. Miles = 1.26.

100 trials per Iteration, 8 Tables:



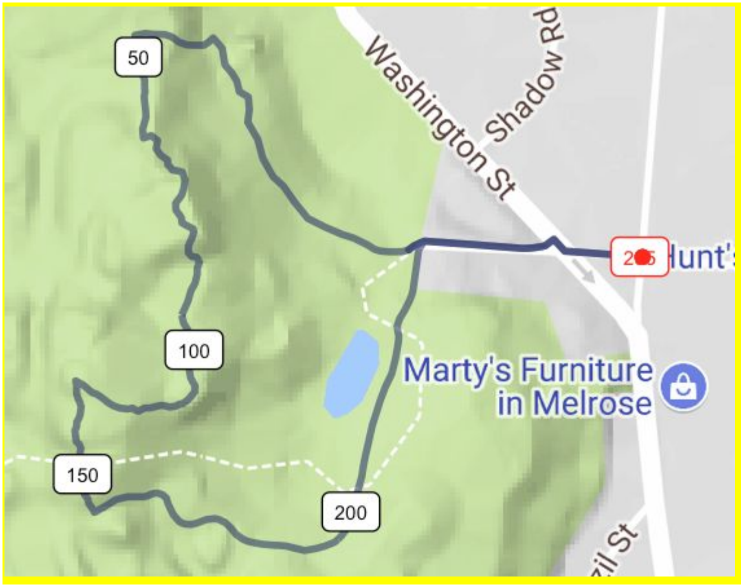
Perfect. Finds trail & creates nice loop. Returns back to start location and mileage = 1.00 (target = 1.0).

100 trials per Iteration, 10 Tables:



Good loop, but not in the trails. Returns to origin point. Mileage perfect at 1.07 miles. FYI - no matter how many times I run this at 10 tables.. The result is always bizarre.

100 trials per Iteration, 12 Tables:



Perfect. Finds trail & creates nice loop. Returns back to start location and mileage = 1.00 (target = 1.0).

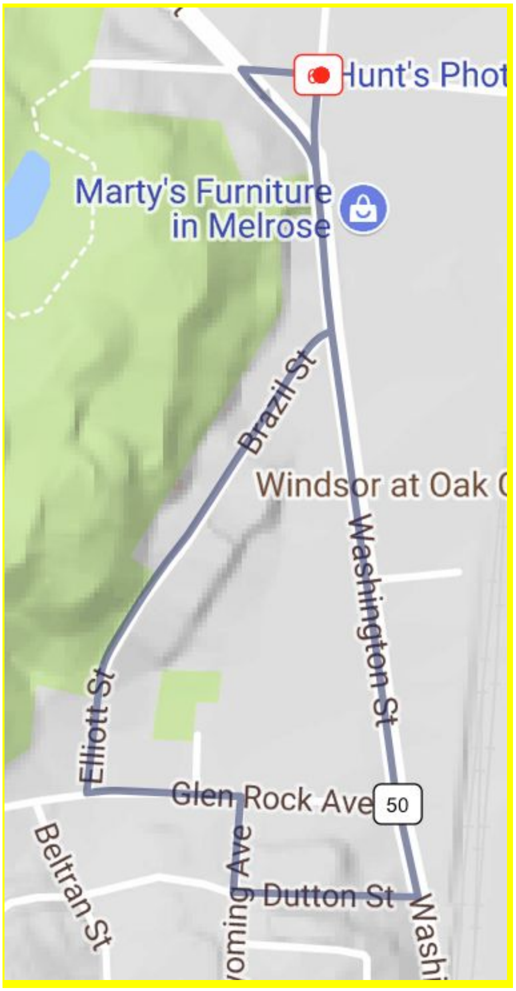
As you can see in the table above, it takes 7 minutes, 54 seconds to get the result we want at 1 mile. Longer runs (which I show below in Free Form Visualization) would require more time while shorter runs would require less. I will note that that time should be split by 8 since it is the average of 8 policy table and, like noted above, with a distributed environment we can run these 8 tables simultaneously. Hence, our time should be viewed slightly less than <1 minute. This is above the goal threshold I set out when starting this project but it is very close. FYI - it's likely that even on a runner's easiest of easy days they are running no less than 3 miles which will increase this calculation time. This is an opportunity for improvement discussed further in Section V.

Just for comparison, I wanted to see what happened when I used less iterations with the same number of tables. Below is the result of using 50 iterations and ensembling 12 tables. As expected, the result is worse than the result where we use nearly double the computation time.

# of Iterations	# of Tables	Time Elapsed	Acceptable Results[8]
50	8	3 min, 45 sec	No. Good loop, not on trail. Miles = 0.97
200	2	4 min, 28 sec	Yes, Good trail loop. Miles = 1.00

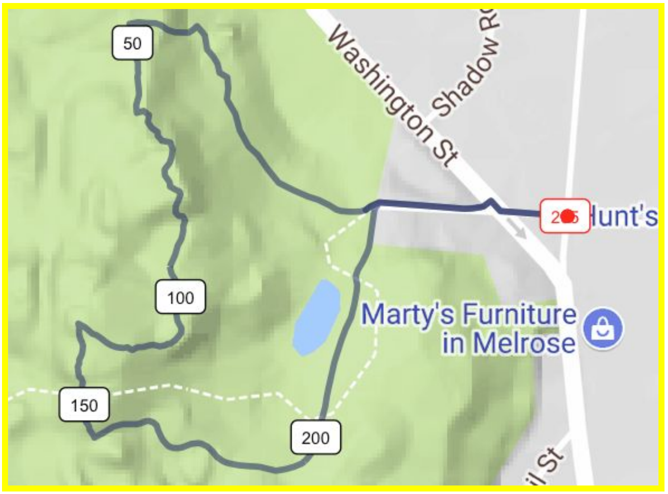
50 trials per Iteration, 8 Tables:

[8] Acceptable Results means that the mileage is close to target mileage and the run forms some type of loop.



It’s a good loop. However, it doesn’t have enough trials to find the benefits of the trail. Yields the right mileage of 0.97 miles.

200 trials per Iteration, 2 Tables:



Perfect. Finds trail & creates nice loop. Returns back to start location and mileage = 1.00 (target = 1.0).

Note that this is the same loop as running 100 trials with 8 tables which took 7 minutes and 54 seconds to run. Running 200 trials and 2 tables, on the other hand, took 4 minutes and 28 seconds. Why not use the one that takes total time? The point of using the table method is to reduce the time the algorithm would take to create a route when we are using parallel processing. Hence, with the 100 trials method it took less than 1 minutes and with the 200 trials method it took over 2 minutes. This means that for further testing we will use the 100 trials and 8 tables method.

There were a few parameters that contributed significantly to the tuning of the algorithm.

- **Rewards**. There are a few different items to learn and the balance between different rewards could sway the results significantly. For instance, if the runner is 50% of the way through the run and they can move further away from the start location to consume more aesthetic reward (see a waterfall), should they or should they turn back. The algorithm puts heavier emphasis on going back to hit target mileage and as such a higher reward on turning back to the start location when duration is high.
    - **Going Backwards**. We rarely want to turn around on a run… although I do out and backs more often than I want to. Perhaps a good loop generating tool can reduce the number of times I do this :) In our algorithm, if the movement is forward (not going back to a point we were just at) a reward is received.
    - **Returning Home.** If the agent is either halfway through the run by distance from origin or 50% of the way through the run by total miles, we encourage the agent to go to the next node that is closest to the origin with a hefty reward
- **Epsilon.** Critical to decay the epsilon. I used a concave downward shape sloping to 0 that allows the algorithm to explore a lot in the initial stages and then exploit the best path (and learn more about the best path) as the number of trails comes to an end. Without leveraging the exploitation phase at the end, lots of simulations ended up where a certain location wouldn't have the necessary q-value mapped to it and as such the results would break. The runner would get to point B in a state of high duration, but there wouldn't be a q-value in our policy table for that event and as such the runner wouldn't know what to do. Repeatedly exploiting the q-table toward the end of the simulation ensured that we would populate the q-values through the end of the run
- **Gamma**. I repeatedly saw bizarre results when using a high gamma > 0.5. Hence using a low value typically and consistently yielded better results since, again, the gamma should nudge the algorithm rather than direct it. In the end, I settle on 0.3 (rather than 0.0) as the gamma because future movements are important. This is logical.
- **Alpha**. The reward structure is not deterministic because the reward of a particular movement depends on the path the runner has come from. For example, imagine 2 paths that both contain the movement from point X to point Y. In the first path, this movement may be high reward but in the second path may be no reward. As such, we need to use an alpha less than one that will ultimately lead to finding the best reward even when part of different paths. After attempting to use a linear decay function for alpha, I found that using a static 0.8 was equally, if not more, effective.

Unfortunately, tweaking these parameters was more of a trial and error approach that took months. I would tweak the parameters, run the simulation, examine the results and examine the policy table… then tweak a little bit more. I tried all sorts of clever workarounds in an attempt to reduce the training time needed… most were unfruitful.

Results

Model Evaluation and Validation

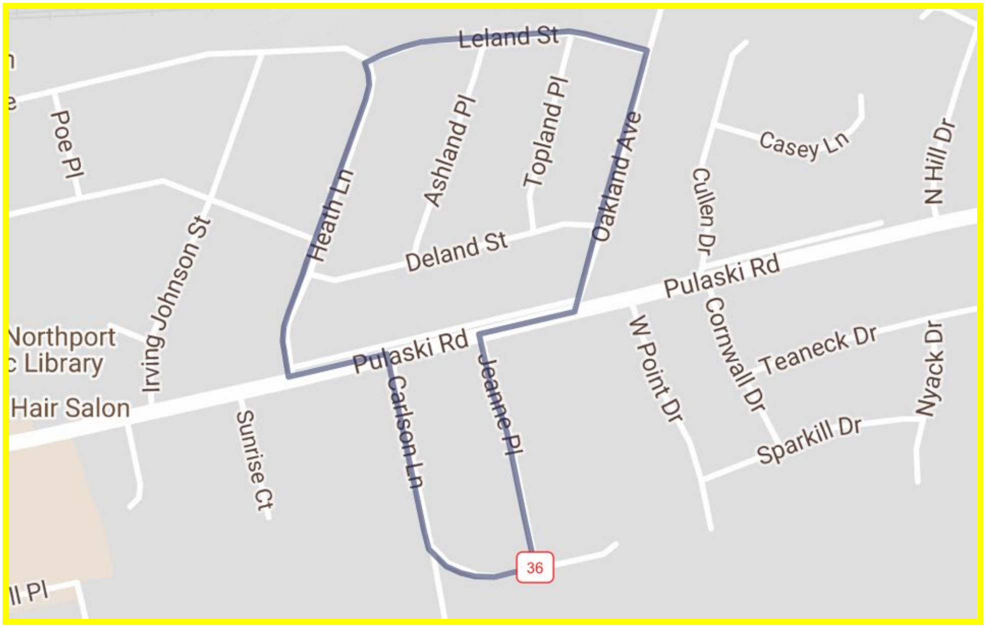
As stated in the introduction, I recognized a strong parallel between the q-learning approach and the manner in which runners learn a trail system when I was running in Middlesex Fells last year and simultaneously taking Udacity's Reinforcement Learning course. It took some convincing on my part to my peers that this algorithm was an appropriate solution to the problem of "where should I run?" as seen in this Udacity forum thread:

https://discussions.udacity.com/t/rl-capstone-project/231147/10

This project now proves that q-learning can be used to solve this problem, but not necessarily at the desired speed (yet).

Although the model was built and tested from a single start location, in order to show the robustness of the model I test the results on 2 different start locations. Note that I finalized the model on the initial test location (Stone Place, Melrose MA.) before testing it on the other random start location (Jeanne Pl. in East Northport NY). I used 1 mile as the test target mileage. Here's what happened:

Starting at Jeanne Pl. in East Northport NY (where it intersects with Squire Dr.) here is what the 1 mile loop would look like after 400 trials. Note that the closest marked trail that I can think of qualitatively is about 1 mile away… so I wouldn't expect to hit trails here. Interestingly, when I look at the openstreetmap data for my home town, the trail coverage is poor -- I may need to fix that eventually. This is the result...



The result is sufficient because even though we can't get to a trail, it still creates a decent loop for a 1 mile run (actually comes in at 0.98 miles). This tells me we are OK to use 100 iterations per mile. With constant usage and feedback from actual runners, we'll be able to vastly improve the quality of the paths this algorithm creates.

Justification

The first benchmark stated was a comparison to routes created on RouteLoops.com and primarily used for timing. First of all, RouteLoops.com can't even create a route under 3 miles. This algorithm can (partly due to the fact that I don't strictly desire a loop, but mostly because the algorithm is more robust). RouteLoops.com never plots a route through the woods or trails which for many runners is the best option. This algorithm can. With that said, RouteLoops.com produces road loops much quicker than our standing algorithm.

The second benchmark is quality of run. All of the criteria of a "good run" listed in section Analysis: Benchmarks are being met. Routes A) start and end in the same location (the return home mechanism is working), B) seek out reward-granting features (reward data is being identified in the OSM data), C) are loops or pseudo-loops (the don't go backwards mechanism is working), and D) are within a quarter mile of target mileage (duration parameter is working). Best of all, testing has shown that these parameters are being met across geographic regions and at multiple length of runs.

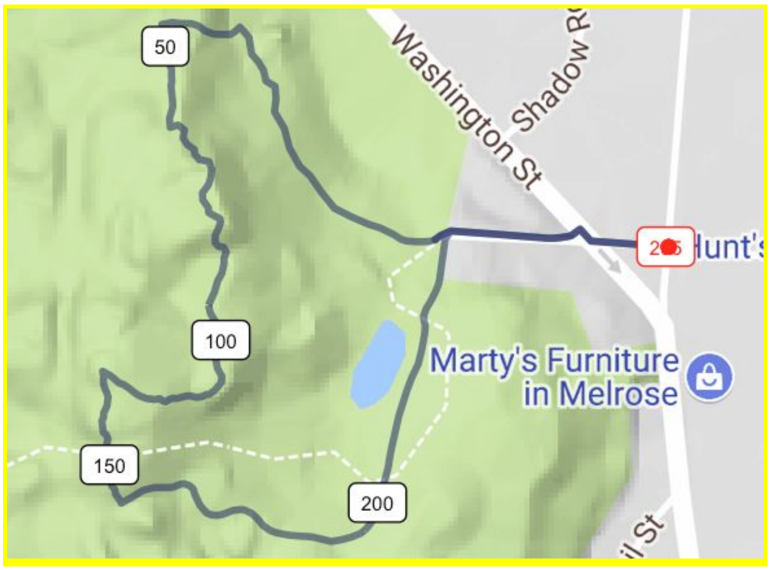
If we go back to the original question -- where should I run? -- there is no doubt that our algorithm's results give the user a much better answer than either of our benchmarks or even using a combination of those benchmarks. Routeloops can show me road loops I may want to run. However, our results instantly creates the route that maximizes the miles on a trail and shows the runner cool stuff.

Conclusion

Free-Form Visualization

The testing above was based on a 1 mile loop. The below visualizations show us what happens when we increase the target mileage of the run. For each mileage band, I included how long it took the algorithm to run and provide brief commentary on the results.

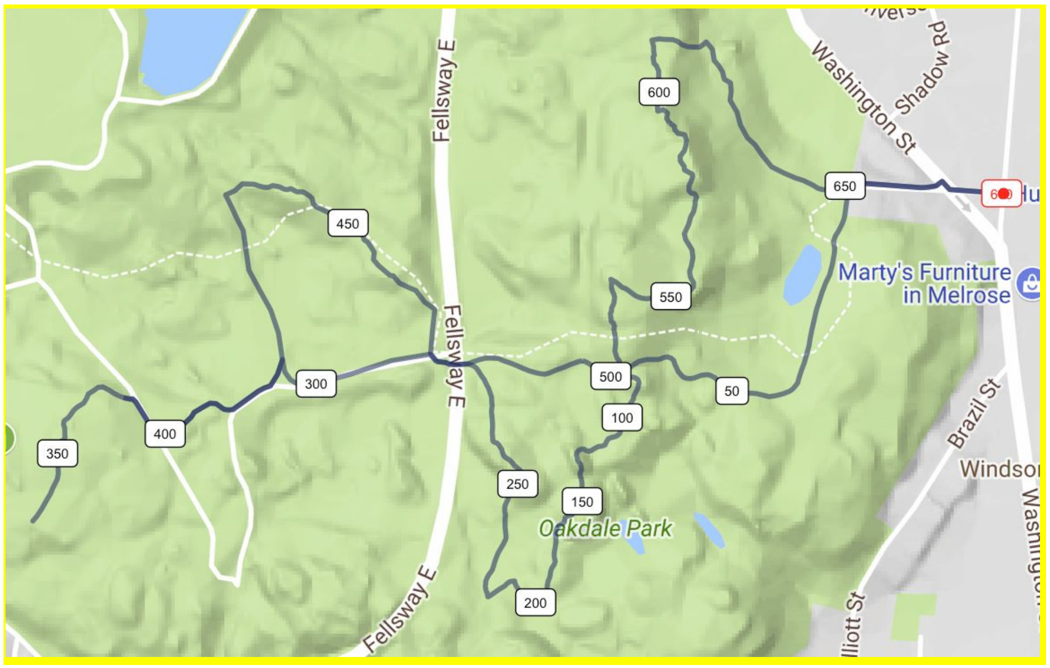
1 mile(s) target (100 iterations, 8 tables):



1.0 mile result. Total time = 7 min, 54 sec (55-60 seconds per table)

Perfect. Finds trail & creates nice loop. Returns back to start location and mileage = 1.00 (target = 1.0).

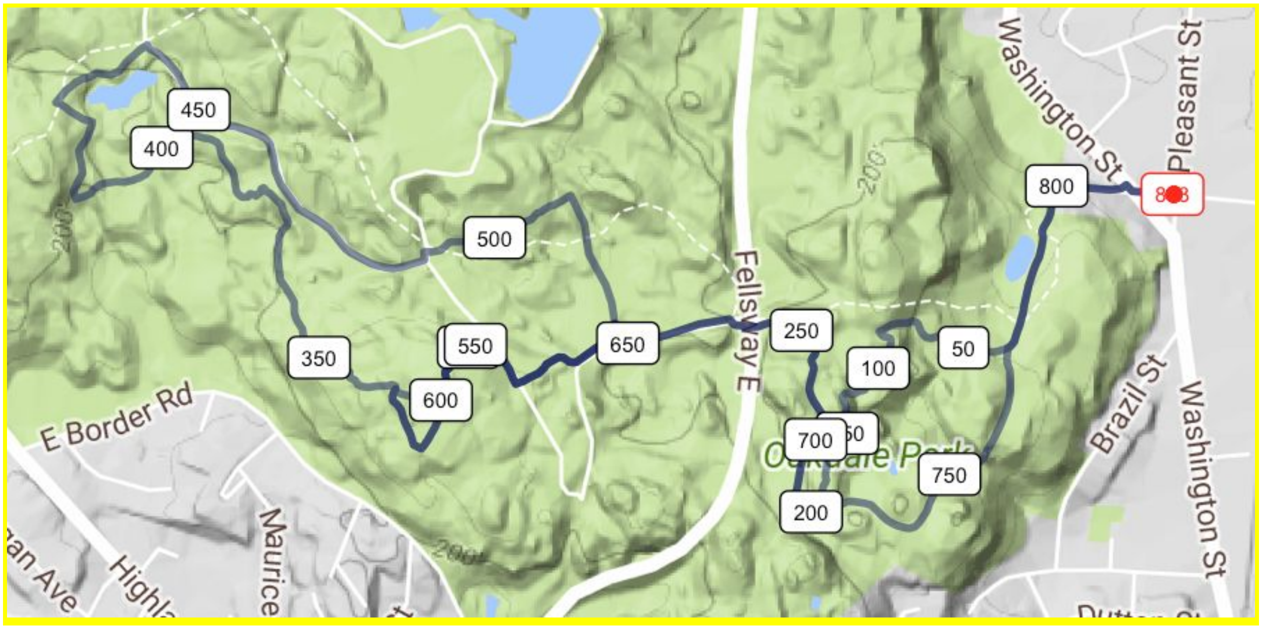
2 mile(s) target (100 iterations, 8 tables):



2.57 mile result. Total time = 15 min 19 sec (1 minute, 54 seconds per table)

Very nice route. Some out and back mini-segments, but interesting. Yielded 2.57 miles, a bit higher than requested.

3 mile(s) target (100 iterations, 8 tables):



4.01 mile result - 28 minutes, 6 seconds (3 min, 30 seconds per table)

Pretty interesting route. Definitely finds trail. Does a loop and some inner-looping. Over shoots the target mileage by 1 mile.

Reflection

My initial reaction to thinking about the time and energy that has already gone into this project is that I should have just done a Kaggle contest for my capstone project. However, I do believe that I now understand more about geospatial data and reinforcement learning than 99.9% of data analysts out there which feels good. There are a multitude of applications for this including automobile navigation. Another idea is leveraging this logic so people can find the safest parts of town (especially when they are in unfamiliar neighborhoods) to walk through, but this would require an additional set of crime and location data.

Improvement

There is certainly room for improvement.

First, the time this takes to calculate is not sufficient. I'm not exactly sure what kind of deep learning can be applied here, but I believe that is the next step. As soon as I get the sign off on this project I will be applying the the Deep Learning Nanodegree to see if I can make that happen.

Second I do wish that I found more of a silver bullet to this problem. Could I combine with other data to run multiple models at the same time with varying parameters and then average the results of the models? It's possible this would greatly improve the resulting paths and decrease time to train the models.

Resources

- https://www.darrinward.com/lat-long/. Originally used this site to plot map points and routes before

writing final r script to visualize routes.

- http://www.gpsvisualizer.com/map_input. Another site I was originally using to plot multiple points on a map.
- https://www.visualcinnamon.com/2014/03/running-paths-in-amsterdam-step-2.html. Used this tutorial to figure out how to draw routes on map in r.
- https://arxiv.org/pdf/0903.4930.pdf. Read this paper to see if there was a way to speed up the learning process of the q-learning algorithm. In the end, I did not implement this technique.
- http://wiki.openstreetmap.org/wiki/Overpass_API/Language_Guide. When attempting to filter dataset to include only 'runnable' nodes, I considered using Overpass_API. In the end I used OsmFilter to scale down the input dataset.
- http://wiki.openstreetmap.org/wiki/Osmfilter. Osm files contain nodes of streets/paths AND buildings. I only wanted runnable ways and used OsmFilter to scale down original file.
- https://stackoverflow.com/questions/82831/how-do-i-check-whether-a-file-exists-using-python. In order to speed up the many iterations of training, every time I ran the algorithm I would save the distances between nodes to a file and read it back in the next time. In doing this I didn't have to call my routing service so many times - it was faster to read the data point from a dictionary if I had done so in the past.
- http://wiki.openstreetmap.org/wiki/Key:historic. Use the OSM wiki to better understand the tagging system I used for rewards.
- https://github.com/Project-OSRM/osrm-backend. The routing service is the backbone of this project. I used OSRM to run a local instance and continuous pull API requests hundreds of thousands (maybe millions?) of times over the course of developing this project.