Lab 3: Frugal Maps

You are not logged in.

If you are a current student, please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

Table of Contents

1) Preparation

This lab assumes you have Python 3.6 or later installed on your machine (3.10 is recommended).

The following file contains code and other resources as a starting point for this lab: `lab03.zip`

You can also see and participate in online discussion about this lab in the "Labs" Category in the forum.

This lab is worth a total of 4 points.

Note that passing all of the tests on the server will require that your code runs reasonably efficiently and that it makes reasonably efficient use of memory.

The questions on this page (including your code submission) are due at 5pm Eastern on Friday, 25 Feb. Checkoffs are due at 10pm Eastern on Wednesday, 2 Mar.

Your Checkoff This Week

You are not logged in. Please log in to see information about your checkoff for this lab.

> **Reminder: Academic Integrity**
>
> Please also review the academic integrity policies before continuing. In particular, **note that you are not allowed to use any code other than that which you have written yourself, including code from online sources.**

2) Introduction

In this lab, you will be working with freely available real-world mapping data to solve the realistic large problem of finding the shortest (or fastest) path between two points on a map. You will implement the backend for a route-finding application, which we will be able to use to plan paths around Cambridge.

All of the software and data we are working with in this lab is freely available, including the mapping data, the images used to render the maps, and the software used to control the map display! In fact, a lot of the world's most widely used software is created in this same spirit of sharing and community. If you are interested in exploring the philosophical ideas behind these movements, a place to start would be the Wikipedia articles on the Free/Libre Software and Free-culture movements. Or talk with Adam (who is passionately interested in these ideas).

2.1) The Data

The data we'll use for this lab comes from OpenStreetMap, and for the bulk to this lab, we are working with data about Cambridge and the surrounding area (about 650 MB of data). While the raw data (downloaded from here) was specified in the OSM XML Format, we have done a little bit of pre-processing to convert the data to a format that is slightly easier to work with.

As in the original format, we have divided the data into two separate pieces: a list of "nodes" (representing individual locations) and a list of "ways" (representing roads or other kinds of connections between nodes). We have stored these data in large files that contain many pickled Python objects, one for each node/way (the function we used to do this is made available to you in the `util.py` file as `osm_to_serial_pickles`, in case you want to try it on your own data).

The Cambridge data set stores these in two files, which are available in the code distribution: `resources/cambridge.nodes` and `resources/cambridge.ways`.

We have also provided a helper function in `util.py` called `read_osm_data` (which has been imported into `lab.py` for you). Calling `read_osm_data` with a filename will produce an object over which you can loop to examine each node/way in turn. For example, you could use the following code to print all of the nodes in this database:

```
for node in read_osm_data('resources/cambridge.nodes'):
    print(node)
```

One important thing worth noting is that this object can only be looped over once (looping over the same data again would require calling `read_osm_data` again).

You can also get a single element out of this object using Python's built-in `next` function (if you want to grab a single entry rather than looping over the whole structure), for example:

```
data = read_osm_data('resources/cambridge.nodes')
print(next(data))
```

Each node (representing a location) is represented as a dictionary containing the following keys:

- `'id'` maps to an integer ID number for the node.
- `'lat'` maps to the node's latitude (in degrees).
- `'lon'` maps to the node's longitude (in degrees).

- `'tags'` maps to a dictionary containing additional information about the node, including information about the type of object represented by the node (traffic lights, speed limit signs, etc.).

(Note also that it is possible for multiple nodes to have the same location (latitude and longitude).)

Each way (representing an ordered sequence of connected nodes) is represented as a dictionary with the following keys:

- `'id'` maps to an integer ID number for the way.
- `'nodes'` maps to a list of integers representing the nodes that comprise the way (in order).
- `'tags'` maps to a dictionary containing additional information about the way (e.g., is this a one-way street? is it a highway or a pedestrian path? etc.).

Try printing out the nodes and ways to get a sense for the kind of information that is available there and then use Python to answer the following questions about the database.

How many total **nodes** are in the database?
(Hint: it may take a long time and use a large amount of memory to make a big list containing all the nodes; try looping over the result from `read_osm_data` instead).

Some of the nodes have a name associated with them (by virtue of having a `'name'` entry in their `'tags'` dictionary). How many of the nodes have a name?

What is the ID number of the node named `'77 Massachusetts Ave'`?

How many total **ways** are in the database?

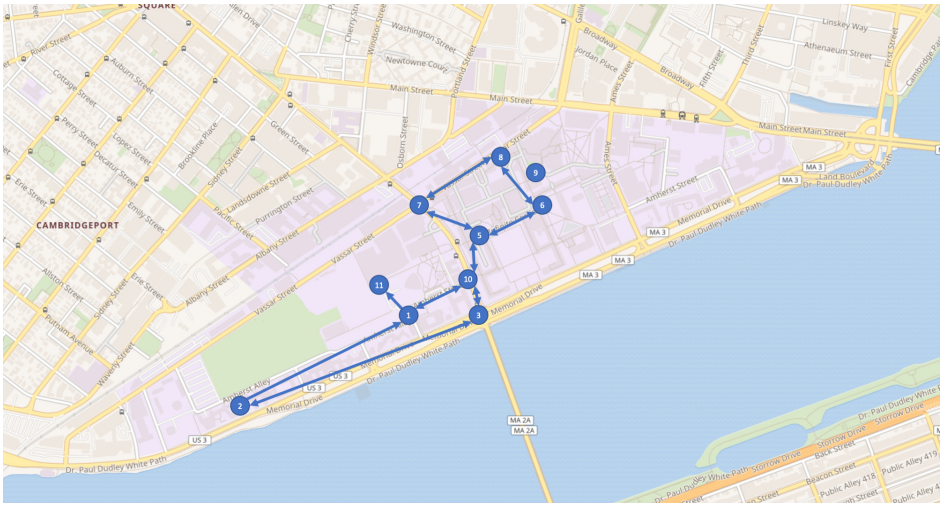
How many of these are one-way streets? (Hint: look at the value associated with the `'oneway'` key in the `'tags'` dictionary if it exists to make a decision; if that key doesn't exist, assume the road is two-way)

This might give you a sense of the scale of the database we're working with; there's a lot of information here!

2.2) Testing

We have provided three datasets for you to work with as you are testing your code, along with a visualization (see section 4.2).

As you are debugging, you may primarily wish to make use of the `mit` data set (containing some manually constructed nodes and ways), which is relatively small compared to the others (to the extent that it should be possible to compute some relevant results manually after printing out the nodes and ways contained therein). An image of the MIT dataset is shown below (click to see a larger version):



`midwest` is quite a bit larger, so it may not be possible to compute results manually from it, but it should load relatively quickly, so it can be used for testing on a slightly larger scale (using the visualization, for example).

`cambridge` is *really* big, and so working with it can take a while. As such, you may not want to do too much testing with it until you are reasonably sure that things are working. However, once things *are* working, it can be neat to plan paths through Cambridge and the surrounding area using the visualization.

3) Shortest Paths

Now that we've had a chance to get used to the format of the data set, we'll embark on our first of two tasks for the lab: finding the shortest path between two nodes. Ultimately, we will implement this as a function `find_short_path_nodes` in `lab.py`. This function takes three arguments:

- `internal_representation`: some structures of your creation (see below)
- `id1`: the ID of our starting node
- `id2`: the ID of our ending node

We are interested in returning the shortest path (in miles) between those two locations.

Ultimately, `find_short_path_nodes` should return a list of IDs (each corresponding to a single node) representing a path between the two given locations, where each pair of adjacent nodes must be connected by a way. If no such path exists, you should return `None`.

You can make use of the ideas introduced in the week 2 lecture and/or view a brief overview of one appropriate algorithm from the lecture.

3.1) Design Considerations

This procedure is fairly complicated, and so are the data we're given, so we'll clarify a few things and make a few small simplifying assumptions, as indicated below.

3.1.1) Road Types

The datasets we're working with contain information not only about roadways, but also about bicycle paths, pedestrian paths, buildings, etc. Since we are (for now, at least) planning paths for cars only, we will only consider a way to be valid for our purposes if it is a roadway (we're responsible citizens of the world, so we won't drive on a bike path or a pedestrian-only walkway).

Some ways in the dataset have a tag called `'highway'`, indicating that the path represents a path people can use to travel (as opposed to the outline of a building, a river, the outline of a park, etc.). We will use this tag to decide what kinds of ways to include in our results.

In particular, we'll only consider a way as part of our path-planning process if:

- it has a `'highway'` tag, **and**
- its `'highway'` tag is in the `ALLOWED_HIGHWAY_TYPES` set that is defined at the top of `lab.py`.

Ways that don't have these properties should be ignored completely.

> In the `cambridge` dataset, how many total nodes from the dataset exist as part of 'valid' ways as described above?

3.1.2) Connectedness

We will assume that we can travel from a node to another node if and only if there is a way that connects them. For example, if we have the following two ways in the database:

```
w1 = {'id': 1, 'nodes': [1, 2, 3], 'tags': {}}
w2 = {'id': 2, 'nodes': [5, 6, 7], 'tags': {'oneway': 'yes'}}
```

then moving from node `1` to node `2`, from `2` to `3`, from `3` to `2`, or from `2` to `1` are all OK because `w1` represents a bidirectional street. But, while moving from `5` to `6` or from `6` to `7` are OK, moving from `7` to `6` or from `6` to `5` are **not** OK because `w2` represents a one-way street (we're responsible citizens of the world, so we respect one-way restrictions).

Note that moving directly from node `1` to node `3` is **not** possible given the above, unless there is another way that directly connects those two nodes.

As we're planning paths between nodes, we'll want to make sure that we only consider a node as a possibility if it exists as part of a way (we can ignore all other nodes in the database).

3.1.3) Distance Measure

Throughout this lab, we will use an approximation for distance (in miles) that takes into account the approximate curvature of the earth. This approximation has been defined as a function `great_circle_distance` in `util.py` (and it has been imported into `lab.py` for your use).

You should not use any other distance measure for this lab, as the test cases expect you to use this measure of distance.

> What is the distance (in miles) between the following two locations, specified in terms of latitude and longitude?
> Location 1: `(42.363745, -71.100999)`
> Location 2: `(42.361283, -71.239677)`
>
> Enter your answer as accurately as Python provides it to you; don't round!

> In the `midwest` dataset, what is the distance (in miles) between the nodes with the following ID numbers?
> ID 1: `233941454`
> ID 2: `233947199`

> In the `midwest` dataset, there is a way with ID number `21705939`. If we were to follow that way from its beginning node to its ending node (and through all the intermediate nodes), how many miles would we have traveled?

3.2) Auxiliary Data Structures

As we saw in lab 2, the choice of internal data structures is going to be important; choosing the "right tools for the job," so to speak, can make a big difference, both in terms of the amount and clarity we have to write, and in terms of the efficiency of that code (in fact, this is even more true in this lab, which makes use of a *lot* more data than Lab 2).

As such, you are free to create whatever data structures you like by filling in the `build_internal_representation` function in `lab.py`. The object returned by this function will be passed in as the first argument to your path-finding functions.

The ultimate goal of this structure is to be able to answer questions quickly about the data that we'll need to answer repeatedly (without looping over the whole dataset). We set things up this way so that we can build these structures once and then use them multiple times to compute various results. As such, it is OK for this function to be a bit slow, so long as it saves time during the actual search process.

However, it may not be possible to store all of the nodes or all of the ways in memory (so if there are nodes and/or ways that you know will be irrelevant, you should not store them in memory).

3.3) Testing

After implementing `find_short_path_nodes`, your code should pass the first 13 test cases in `test.py`.

4) Starting and Ending Points

Often, when we are interested in path planning in real geographical contexts, our given starting point might not actually be a known point of interest (for example, it might be a location specified by GPS).

To this end, we will augment our system so that it can accept arbitrary locations for the starting and ending points, specified as `(latitude, longitude)` tuples.

We will do this by implementing a new function `find_short_path` in `lab.py`. This function should take three arguments:

- `map_rep`: the result of calling your `build_internal_representation` function (see above)
- `loc1`: a tuple of (latitude, longitude) of our starting location
- `loc2`: a tuple of (latitude, longitude) of our ending location

And this function should return a list of `(latitude, longitude)` tuples connecting the start and end location.

It is entirely possible that the locations passed to `find_short_path` will not correspond exactly to nodes in the dataset. As such, we will instead plan our shortest paths as follows, where a "relevant" way refers to any way that we will actually consider in our path planning (i.e., one that has not been ignored by the rules in section 3.1.1):

- finding the nearest node to `loc1` that is part of a relevant way (call this node n_1)
- finding the nearest node to `loc2` that is part of a relevant way (call this node n_2)
- finding the shortest path from n_1 to n_2 (in terms of miles)
- convert the resulting path into (latitude, longitude) tuples.

After finding n_1 and n_2, we will ignore `loc1` and `loc2` completely during the search process. They should not show up in the path, and they should not be considered when computing the total distance represented by the path.

> In the `midwest` dataset, what is the ID number of the nearest relevant node to location `(41.4452463, -89.3161394)`, i.e., the nearest node to that location that is part of a way we'll actually consider in our path-planning?

4.1) Testing

After implementing `find_short_path`, your code should pass the first 27 test cases in `test.py`.

4.2) Visualization

As with the last lab, we have provided a web-based interface that acts as a visualization for your code.

Here, we use leaflet to display map data from the Wikimedia foundation (more information about their Maps service is available here).

You can start the server by running `server.py` but providing the filename of one of the datasets as an argument, for example:

```
python3 server.py midwest
```

or

```
python3 server.py cambridge
```

(If you are not using a terminal to run your code, you can replace `sys.argv[1]` in `server.py` with a hard-coded name of a dataset, like `'midwest'`, which you can then change later to load in different datasets.)

This process will first build up the necessary internal representation for pathfinding by calling your `build_internal_representation` function, and then it will start a server. After the server has successfully started, you can interact with this application by navigating to `http://localhost:6009/` in your web browser. In that view, you can double-click on two locations to find and display a path between them.

Alternatively, you can manually call your path-finding procedure to generate a path and then pass its path to the provided `to_local_kml_url` function to receive a URL that will initialize to display the resulting path. For example, to show a path from Adam's high school to Timber Edge Alpaca Farms (where he worked in high school), you could run the following:

```
phs = (41.375288, -89.459541)
timber_edge = (41.452802, -89.443683)
map1 = build_internal_representation('resources/midwest.nodes', 'resources/midwest.ways')
print(to_local_kml_url(find_short_path(map1, phs, timber_edge)))
```

which should print out a URL that, when pasted into the browser, shows the path you found (assuming the server is running). If you started the server with the `midwest` dataset loaded, you can then double-click around that area to find other paths.

5) Improving Runtime With Heuristics

By now, your code should be working for planning paths, and it should be passing the test cases associated with `find_short_path`. In this section, we'll introduce an optimization that will allow us to speed up our searches while retaining the optimality of the paths we return.

The method for finding minimum-cost paths described on the page linked above involves searching radially outward from a starting point in terms of increasing cost, but it does not take into account information about where the goal is, and so it can waste time considering paths that don't really make sense (for example, paths that move off in the wrong direction, away from the goal position).

We consider those paths because our search process was deciding which paths to consider based on the total cost of the path so far, let's call it $g(n)$:

$g(n)$ = the path cost from the starting node to node n

When we have multiple paths we could consider, we always start with the one that has the smallest $g(n)$ value (the shortest total distance in miles), with no regard for whether that path was moving in a sensible direction or not. While this does guarantee that we end up with an optimal path, it wastes time.

We can do a bit better by introducing the notion of a *heuristic function* $h(n)$:

$h(n)$ = the estimated cost of the lowest-cost path from node n to the goal node

From this, we can derive a new function:

$$f(n) = g(n) + h(n)$$

Because $g(n)$ represents the path cost from the start node to node n, and $h(n)$ is the estimated cost of the lowest-cost path from n to the goal, we have:

$f(n)$ = the estimated cost of the lowest-cost solution involving n

Because the total cost of the path is the thing we're trying to optimize, we can make a slight change to the algorithm from above: when we have multiple paths to consider, we should always choose the one with the lowest $f(n)$ value as the next path to be considered (i.e., the one with the lowest estimated total path cost). By so doing, we can focus our attention on paths that kind of move toward the goal, and we avoid wasting time considering other paths. The most effective heuristics will be easy to compute but will be reasonably accurate estimates of the cost of the optimal path to the goal.

Note that the introduction of a heuristic does not change the **cost** of the paths, only the order in which we consider the various paths.

5.1) Heuristics and Optimality

Interestingly, as long as we're careful with the design of our heuristics, we can get this benefit without sacrificing the optimality of the path we return, so long as our heuristic has a couple of properties, which we call *admissibility* and *consistency*:

- a heuristic is *admissible* if it never overestimates the cost of the optimal path from any node to the goal, i.e., for all nodes n:

  $$h(n) \leq c^*(n)$$

  where $c^*(n)$ represents the actual cost of the least-cost path to the goal.

- a heuristic is *consistent* if, for each node n and each successor n' of that node, the heuristic evaluated at n is no more than the heuristic evaluated at n' plus the cost of traveling directly from n to n':

  $$h(n) \leq c(n, n') + h(n')$$

If these two properties hold for our heuristic function $h(n)$, then we are guaranteed that we will still find an optimal path if we sort our agenda according to $f(n) = g(n) + h(n)$.

5.2) A Heuristic For This Problem

Given the above, a reasonable heuristic in this domain is the distance directly from the given node to the goal node:

$$h(n) = \text{great\_circle\_distance}(n, \text{goal})$$

This function provides a decent estimate of our overall cost, it is admissible and consistent, and it is pretty fast to compute. As such, we should expect that it will do a decent job of improving the efficiency of our search procedure.

To test this theory, try running a search between the following two locations using the `cambridge` dataset, both *with* and *without* the heuristic, and make note of *the total number of paths we pull off of the agenda in each case*. How do those two numbers differ?

- Location 1: `(42.3858, -71.0783)`
- Location 2: `(42.5465, -71.1787)`

Be prepared to discuss these results, including how you computed them, during your checkoff conversation (or in your answers below if you are not scheduled for an in-person checkoff).

6) Need for Speed (Limits)

So far, our planning has been based purely on distance, but oftentimes, when planning a route between two locations, we are actually interested in the amount of *time* it will take to move from one location to another.

For the last part of this lab, you should implement `find_fast_path`, which, unlike `find_short_path`, should take into account speed limits (we're responsible citizens of the world, so we won't drive over the speed limit).

Some ways in the dataset store information about the speed limit along that way. That said, unfortunately, speed-limit information is somewhat sparse in OSM data (at least in these datasets), and so we'll have to guess a little bit for some of the roads. We have done a little bit of preprocessing of the data for you, to make extracting this information a little bit easier than it would otherwise be.

For each way, we'll determine the speed limit as follows:

- if the way has the `'maxspeed_mph'` tag, the corresponding value (an integer) represents the speed limit in miles per hour
- if that tag does not exist, look up the way's `'highway'` type in the `DEFAULT_SPEED_LIMIT_MPH` dictionary and use the corresponding value.

If two nodes are connected by more than one way with distinct speed limits, you should always prefer higher of the two speed limits.

Note that implementing this new function might require changing or reorganizing your code for `find_short_path` and/or for `build_internal_representation`. Even if it isn't required for functionality, you may also be presented with opportunities to refactor your `lab.py` to avoid duplicated code.

6.1) Heuristics

With this new notion of optimality, the heuristic function from earlier is no longer admissible (so we are no longer guaranteed to return an optimal path!).

As such, for this part of the lab, it is fine not to use a heuristic; but you may find it interesting to try to come up with an effective heuristic that is admissible and consistent given this new measure of cost, in order to speed up your search.

6.2) Visualization

If, after starting `server.py`, you open the following URL in your browser (note the difference from above), the web UI will use your `find_fast_path` instead of `find_short_path` for pathfinding when double-clicking:

```
http://localhost:6009/?type=fast
```

Try using both this and the shortest-path metric between a few different points on the Cambridge map. How would you expect those to differ? Do your results match your expectation?

During the checkoff, you will be asked to demonstrate your code running in the UI by finding both the shortest and fastest paths from Waltham, MA (west of Cambridge on the map) to Salem, MA (north and east of Cambridge) using the `cambridge` data set. As such, if you are coming in for an in-person checkoff, please make sure that you've got the server running with the `cambridge` dataset loaded when you ask for your checkoff.

7) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `submit-009-lab` script. The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ submit-009-lab -a lab03 /path/to/lab.py
```

Running that script should submit your file to be checked, and it should also provide some information about how and where to get feedback about your submission. Reloading this page after submitting will also show some additional information:

> You have not yet made any submissions to this assignment.

8) Checkoff

Once you are finished with the code, please come to office hours and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code and test cases in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions). In particular:

- Discuss your choice for the data structure that `build_internal_representation`. What other possibile choices could you have made instead, and why did you choose this representation? Did your representation change at all as you encountered new sections of the labs?
- Imagine using a BFS (as discussed in recitation 3) to find shortest paths, instead of this method. What would we expect to be different about the paths returned from BFS, versus the paths we're returning here? Try this experiment. What were the results? Do they match your expectations?
- How did the heuristic affect the speed of the search? How many nodes were expanded with and without the heuristic?
- How does your implementation of `find_fast_path` differ from `find_short_path`?
- Test your code running in the UI by finding both the shortest and fastest paths from Waltham, MA (west of Cambridge on the map) to Salem, MA (north and east of Cambridge) using the `cambridge` data set. What differentiates these paths? Why do they look the way they do?

> *You have not yet received this checkoff.*

9) (Optional) Extra Pieces

If you are interested, there are a number of ways you could modify or build on the code from this lab to do some other kinds of cool things! Here are a few ideas, and we're happy to help if you're interested to work on any of these (or on other ideas of your own!):

1. Import data from your own home state / home country! You can download data from http://download.bbbike.org/osm/ or https://download.geofabrik.de/ and use the `osm_to_serial_pickle` function in `util.py` to get convert the data to the format used in this lab.

   But be warned that if you're importing data from a country that uses a sensible measure of speed like `kph`, the speed limit calculations might be a bit off, since they assume MPH...so that might require some other adjustments.

2. Implement an admissible heuristic for the version of the code that takes speed limits into account.

3. Use the route planning for bicycles or pedestrians rather than motorists by modifying the "highway" types that are allowed. See this page for a list of "highway" types, and note also that many ways contain information about bicycle travel.

   Note that you probably won't be able to implement the 'speed limit' behavior in this case :)

4. Often times, there are considerations other than speed limit that factor in to what makes a path desirable. You could take this into account by introducing a penalty (in terms of cost) for paths that move through nodes that are labeled as traffic lights, or for any time we transition between ways, or something like that.

5. Depending on how you implemented things, you may find that a lot of the time spent planning paths between two locations is actually spent looking up the two nodes that we want to use as our starting and ending points. So you might try to find a way to speed that process up.

6. As it currently stands, we are ignoring a large class of tags called "relations" in the OSM data, which include restrictions on driving such as "no left turns." You could try modifying your code to take these relations into account.