Getting Started Tips

How to Get Started

Step 1: Make sure your code compiles and that you can open localhost:4567 and see the blank screen. If you get a 404 error, you can open map.html manually using IntelliJ.

Step 2: Read through the spec. It will probably feel like there are a bunch of holes in your understanding. This is OK. It'll take a while to feel everything out.

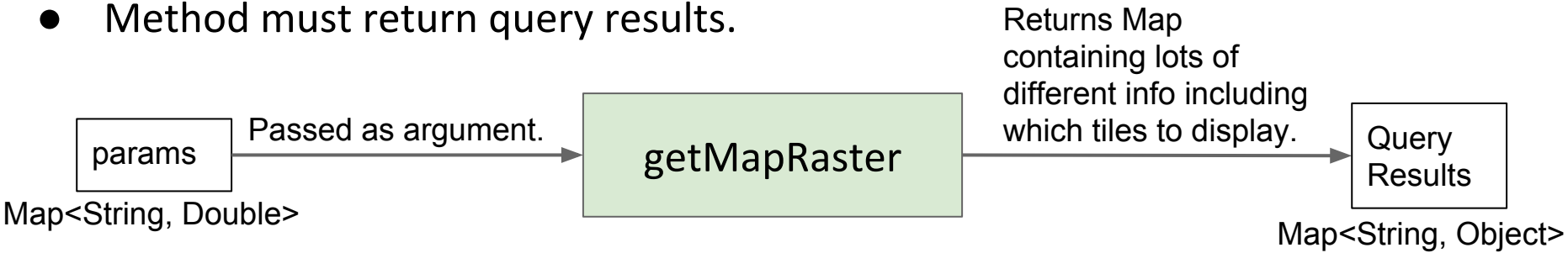
- These slides will give you some additional tips in understanding what everything means.
- Throughout the assignment, feel free to write a little bit of code to help bolster your understanding (e.g. printing parameters, writing helper methods), even before you understand the full picture.

Rastering (Conceptual)

The getMapRaster Method

The hardest part of this project is getting started on getMapRaster.
Let's acquaint ourselves with the API of this method.

- Method is given `params` (what user wants).
- Method must return query results.

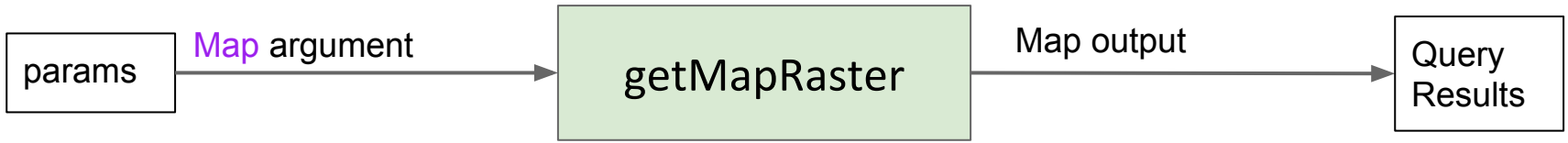


Example:

Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

This means:

- The user is requesting to see everything with longitude (a.k.a. x coordinates) -122.24053 and -122.241632, and latitude (a.k.a. y coordinates) 37.8765 and 37.87548. This is our **query box**.
- The user's browser window display is 892 pixels wide by 875 pixels tall.



Example:

You can do this assignment without ever knowing this fact, but it makes the numbers on the coming slides nicer.

Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

This means:

- The user is requesting to see everything with longitude (a.k.a. x coordinates) -122.24053 and -122.241632, and latitude (a.k.a. y coordinates) 37.8765 and 37.87548. This is our **query box**.

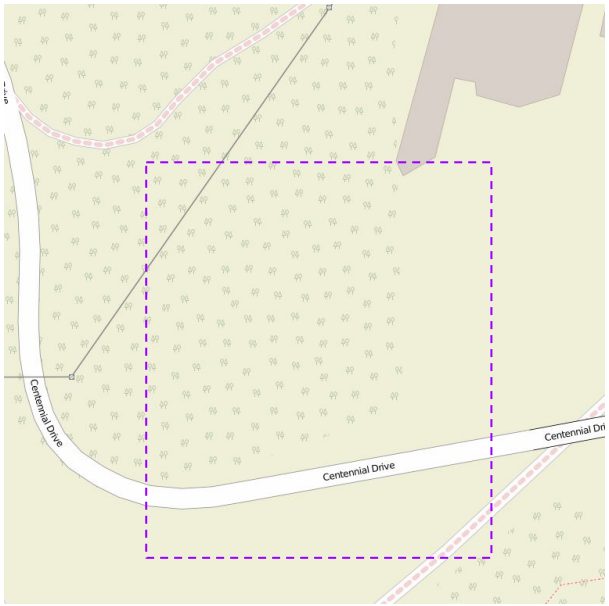
At our latitude (38 degrees north), each degree of longitude is S_L = 288,200 feet.

- User wants a box `(122.241632 - 122.24053)` x S_L = 317 feet wide.
- User wants it drawn on a screen space that is 892 pixels wide.
- User thus wants an image that is roughly `317 / 892 = 0.355` feet per pixel.

High Level Results

Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

Ultimately, what our user wants to see is some region of the world, outlined as a dotted purple box (the query box) below.



Two Critical Requirements:

- Must draw entire query box (ok to go over).
- Must have sufficient pixels/distance (LonDPP).

As before, distance per pixel (dpp) calculation:

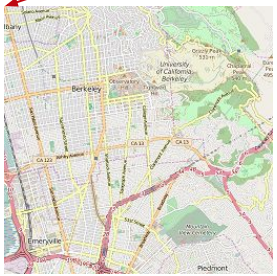
- xDist = 122.24053 -122.241632 = 0.00110199999
- lonDPP = xDist / w = 0.0011019 / 892 = 1.23542×10^{-6} lon/pixel = 0.355 feet/pixel

The Input Files

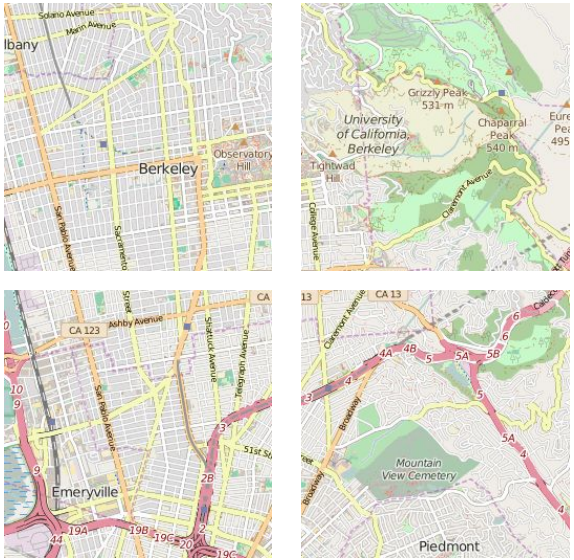
We've provided a large number of images at 8 different levels of zoom.

- d0_x0_y0.png is the entire world.
- d1_x0_y0.png/d1_x1_y0.png/d1_x0_y1.png/d1_x1_y1.png are the northwest, northeast, southwest, and southeast corners of the world.
- All images are 256 x 256.

(-122.29980, 37.89220)



(-122.21191, 37.82280)



https://sp18.datastructur.es/materials/proj/proj3/FileDisplayDemo.html

High Level Results

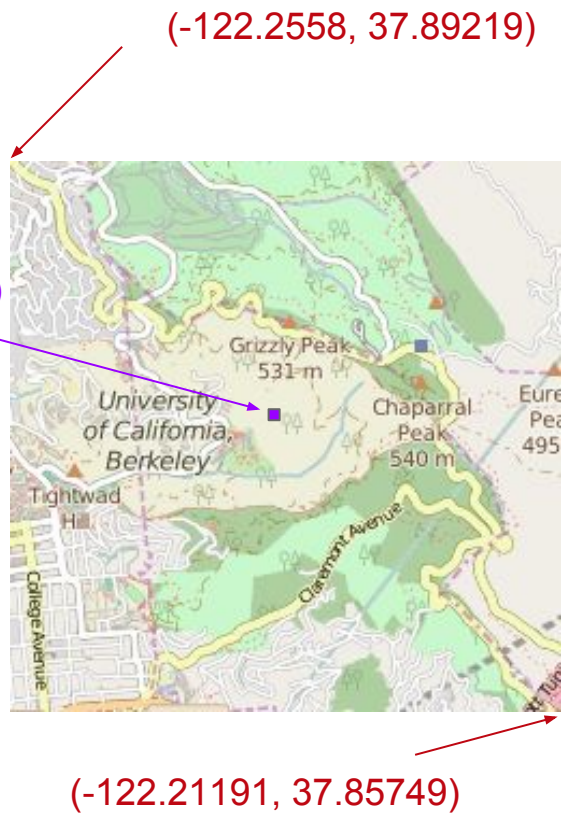
Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

Goal:

- Display images that cover desired range at resolution of at least 0.355 feet/pixel.

Example:

- Could we use d1_x1_y0.png (shown at right)? Attributes:
  - Top left corner: (-122.2558, 37.89219)
  - Bottom right corner: (-122.21191, 37.85749)
- d1_x1_y0.png covers desired area, but LonDPP is:
  - S_L * (122.2558 - 122.21191) / 256 = ~49 feet per pixel.
- Resolution is much too low.



High Level Results

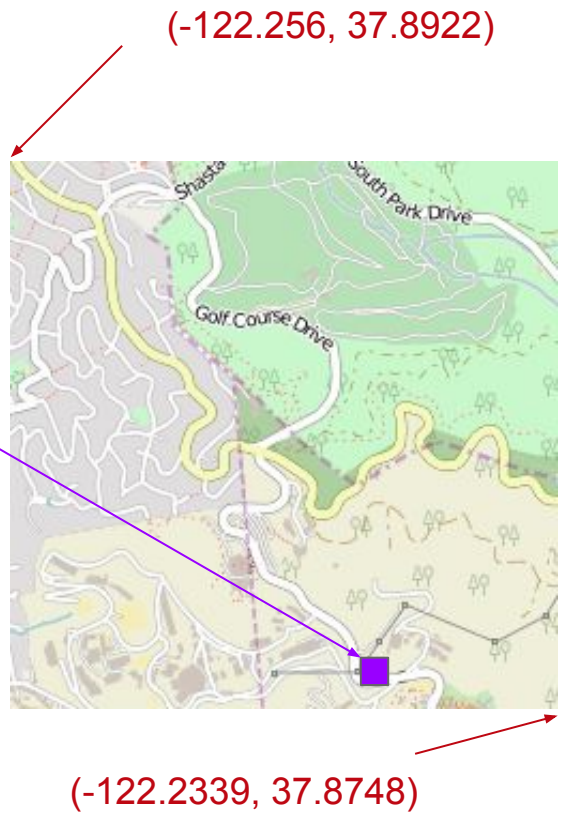
Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

Goal:

- Display images that cover desired range at resolution of at least 0.355 feet/pixel.

Example 2:

- Could we use d2_x2_y0.png? Attributes:
  - Top left corner: (-122.256, 37.8922)
  - Bottom right corner: (-122.2339, 37.8748)
- d2_x2_y0.png covers desired area, but LonDPP is:
  - S_L * (122.256 -122.2339) / 256 = ~25 feet per pixel.
- Resolution is still much too low!



Test Your Understanding

Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

Goal:

- Display images that cover desired range at a resolution of at least 0.355 feet/pixel.

Observations:

- Our depth 1 file `d1_x1_y0` had a resolution of ~49 feet/pixel.
- Our depth 2 file `d2_x2_y0` had a resolution of ~25 feet/pixel.

What depth file would be just barely “good enough” for the user’s request?

Test Your Understanding

Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

Goal:

- Display images that cover desired range at a resolution of at least 0.355 feet/pixel.

Observations:

- Our depth 1 file `d1_x1_y0` had a resolution of ~49 feet/pixel.
- Our depth 2 file `d2_x2_y0` had a resolution of ~25 feet/pixel.

What depth file would be just barely "good enough" for the user's request?

- Depth 9, but provided file only go to depth 7, so use depth 7 instead!

This is approximate. If you just stick this in your code without knowing what this all means, you will have a bad time.

Resolution = ~49 / 2^(D-1)

For D = 8, we get: ~49 / 2^8 = ~0.38 feet/pixel (too coarse)

For D = 9, we get: ~49/2^8 = 0.19 feet/pixel

High Level Results

Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

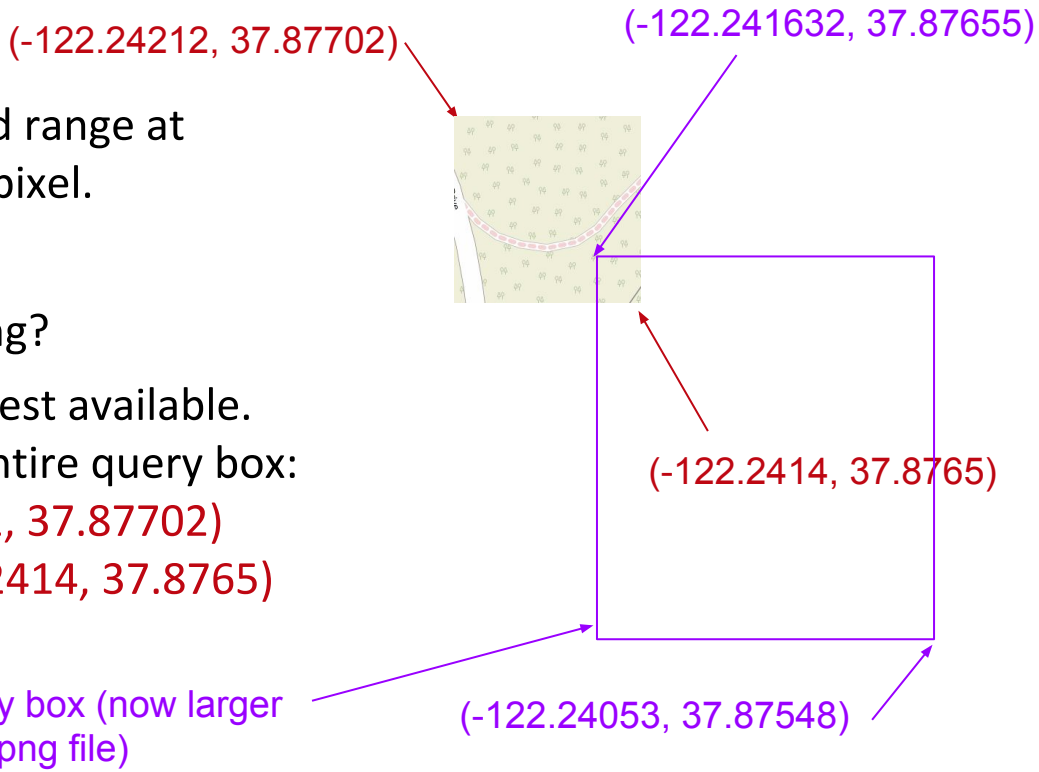
Goal:

- Display images that cover desired range at resolution of at least 0.355 feet/pixel.

Example 3: Could we use `d7_x84_y28.png`?

- Resolution is 0.773 feet / pixel. Best available.
- However, image doesn't cover entire query box:
  - Top left corner: (-122.24212, 37.87702)
  - Bottom right corner: (-122.2414, 37.8765)

We need more images!



High Level Results

Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

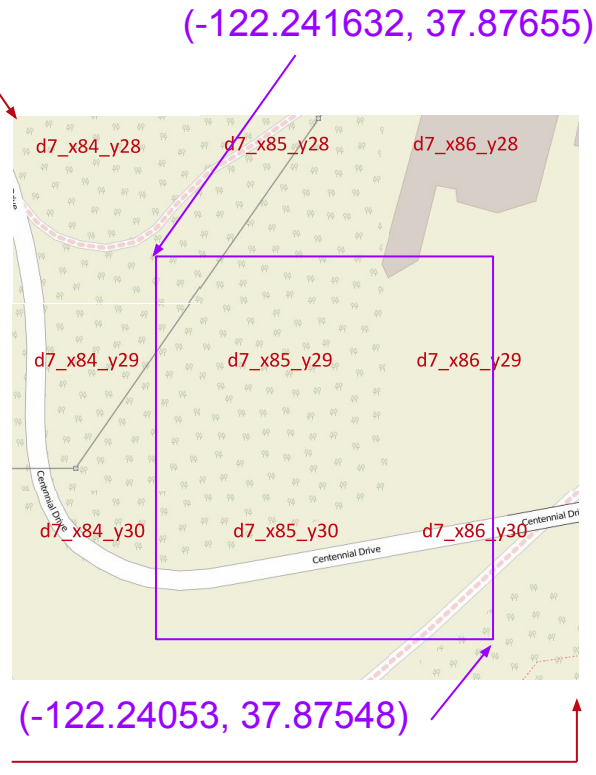
Goal:

- Display images that cover desired range at resolution of at least 0.355 feet/pixel.

Can use d7_x84_y28, d7_x85_y28, d7_x86_y28, d7_x84_y29, d7_x85_y29, d7_x86_y29, d7_x84_y30, d7_x85_y30, and d7_x86_y30.

Tiled together, these 9 images cover the entire space we need to cover.

- Actually cover a bit more space, but that's fine!



Example:

Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

This means:

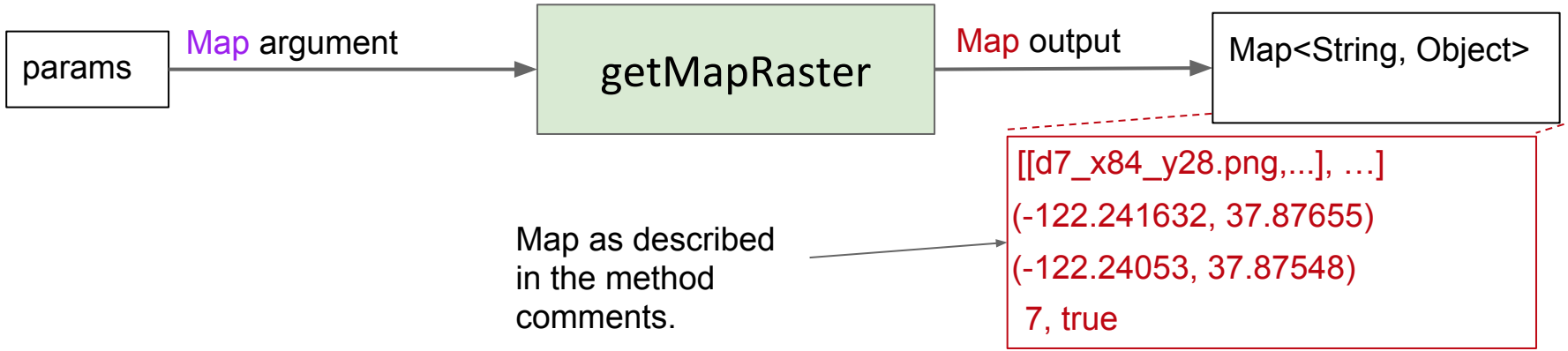
- The user is requesting to see everything with longitude (a.k.a. x coordinates) -122.24053 and -122.241632, and latitude (a.k.a. y coordinates) 37.8765 and 37.87548. This is our **query box**.
- The user's browser window display is 892 pixels wide by 875 pixels tall.

How do we return this picture, which consists of a 3x3 grid?

High Level Picture

Suppose `params = {ullon=-122.241632, lrlon=-122.24053, w=892.0, h=875.0, ullat=37.87655, lrlat=37.87548}`

Returned value might be: `{raster_ul_lon=-122.24212646484375, depth=7, raster_lr_lon=-122.24006652832031, raster_lr_lat=37.87538940251607, raster_ul_lat=37.87701580361881, query_success=true, render_grid=[[d7_x84_y28.png, d7_x85_y28.png, d7_x86_y28.png], [d7_x84_y29.png...]}`



FileDisplayDemo

Don’t miss the FileDisplayDemo:

- https://sp18.datastructur.es/materials/proj/proj3/FileDisplayDemo.html

Can use this tool to deepen understand the layout of the files and check your understanding of various numerical aspects of the project.

General Programming Tips

Some General Tips

Break your goals down into smaller tasks, e.g.

- Given a query, find all the filenames that go with that query. Possible subgoals:
  - Figure out the correct depth for the query.
  - Figure out how to compute the bounding box for a given filename.
  - Figure out how many tiles you'll need.

I use the term "Hello Worlding" to refer to little programs that you write just to be able to get some momentum going.

Solve your smaller tasks by **"hello worlding"** first:

- For example: Before trying to write the correct result to the query, consider simply hard coding some arbitrary answer.
- Then move on to trying to display the correct images.

Some General Tips

When trying to test your code, don’t use map.html at first.

- Use the provided JUnit test files (not actually released yet at the time of this video).
- Make sure that your code works with the provided test1234.html, testTwelveImages.html, and test.html before using map.html.
  - These three html files have hard-coded queries that cannot be interacted with (unlike map.html)

Some General Tips

Take things a little at a time.

- Writing tons of code all at once is going to lead to misery and only misery.
- If you wrote too much stuff and feel overwhelmed, comment out whatever is unnecessary.

Rastering (Programming Tips)

Getting Started Programming on Rastering

Make sure you can get access to the inputs provided by your web browser. My recommendation:

- Edit `getMapRaster` and print out the `params` on execution.

```
public Map<String, Object> getMapRaster(Map<String, Double> params) {
   System.out.println(params);
   ...
}
```

Then run MapServer, try opening up test.html in your browser, and you should see something like below (though the order and coloring might be different).

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_71.jdk/Contents/Home/bin/java ...
[Thread-0] INFO org.eclipse.jetty.util.log - Logging initialized @1481ms
[Thread-0] INFO spark.webserver.JettySparkServer - == Spark has ignited ...
[Thread-0] INFO spark.webserver.JettySparkServer - >> Listening on 0.0.0.0:4567
[Thread-0] INFO org.eclipse.jetty.server.Server - jetty-9.3.2.v20150730
[Thread-0] INFO org.eclipse.jetty.server.ServerConnector - Started ServerConnector@72886fbc{HTTP/1.1,[http/1.1]}{0.0.0.0:4567}
[Thread-0] INFO org.eclipse.jetty.server.Server - Started @1752ms
{lrlon=-122.24053369025242, ullon=-122.24163047377972, w=892.0, h=875.0, ullat=37.87655856892288, lrlat=37.87548268822065}
```

HelloWorlding

In line with the practice of “hello worlding”, you might consider:

- Hard coding a return value into `getMapRaster` that ignores the query rectangle and just returns some arbitrary region of the map that you choose.
  - This is good for ensuring that you know how to interact with your map, and that you know how to return a working answer with `getMapRaster`.

Identifying the Right Tiles

There are many possible algorithms for identifying the right tiles.

- One approach: Iterating through every single filename and checking to see if the corresponding file is “good”.
  - Downsides:
    - Linear in the number of files (though this has no practical implications since there are only a few tens of thousands of files).
    - Nontrivial to figure out how to arrange the good files.
    - Resulting code is messier than it could be.

Try and come up with something better!

Graph Building

Working with Data

Real data sets are messy. You won't fully understand them and this is normal.

- Your job is just to get the roads and intersections into the system.

In the OSM Format (http://wiki.openstreetmap.org/wiki/OSM_XML), files have the following information (in this order):

- an XML suffix introducing the UTF-8 character encoding for the file (we don't care about this)
- an osm element, containing the version of the API (and thus the features used) and the generator that distilled this file (e.g. an editor tool) (we also don't care about this)
- a block of nodes (vertices in our graph)
- a block of ways (edges in our graph)
- a block of relations (we don't care about these)

A Peek at the Data

Structure of our data:

- an XML suffix introducing the UTF-8 character encoding for the file (we don't care about this)
- an osm element, containing the version of the API (and thus the features used) and the generator that distilled this file (e.g. an editor tool) (we also don't care about this)
- a block of nodes (vertices in our graph)
- a block of ways (edges in our graph)
- a block of relations (we don't care about these)

```
<?xml version='1.0' encoding='UTF-8'?>
<osm version="0.6" generator="osmconvert 0.8.4">
  <bounds minlat="37.821" minlon="-122.3269999" maxlat="37.893"
maxlon="-122.1889997"/>
  <node id="30362155" lat="37.8213443" lon="-122.2974753" version="1"/>
  <node id="30362156" lat="37.8217935" lon="-122.2982612" version="1"/>
  <node id="30362160" lat="37.8240326" lon="-122.3032181" version="1"/>
```

A Peek at the Data

Structure of our data:

- an XML suffix introducing the UTF-8 character encoding for the file (we don't care about this)
- **an osm element, containing the version of the API (and thus the features used) and the generator that distilled this file (e.g. an editor tool) (we also don't care about this)**
- a block of nodes (vertices in our graph)
- a block of ways (edges in our graph)
- a block of relations (we don't care about these)

```
<?xml version='1.0' encoding='UTF-8'?>
<osm version="0.6" generator="osmconvert 0.8.4">
  <bounds minlat="37.821" minlon="-122.3269999" maxlat="37.893"
maxlon="-122.1889997"/>
  <node id="30362155" lat="37.8213443" lon="-122.2974753" version="1"/>
  <node id="30362156" lat="37.8217935" lon="-122.2982612" version="1"/>
  <node id="30362160" lat="37.8240326" lon="-122.3032181" version="1"/>
```

A Peek at the Data

Structure of our data:

- an XML suffix introducing the UTF-8 character encoding for the file (we don't care about this)
- an osm element, containing the version of the API (and thus the features used) and the generator that distilled this file (e.g. an editor tool) (we also don't care about this)
- a block of nodes (vertices in our graph)
- a block of ways (edges in our graph)
- a block of relations (we don't care about these)

```
<?xml version='1.0' encoding='UTF-8'?>
<osm version="0.6" generator="osmconvert 0.8.4">
  <bounds minlat="37.821" minlon="-122.3269999" maxlat="37.893"
maxlon="-122.1889997"/>
  <node id="30362155" lat="37.8213443" lon="-122.2974753" version="1"/>
  <node id="30362156" lat="37.8217935" lon="-122.2982612" version="1"/>
  <node id="30362160" lat="37.8240326" lon="-122.3032181" version="1"/>
```

A Peek at the Data

Structure of our data:

- an XML suffix introducing the UTF-8 character encoding for the file (we don't care about this)
- an osm element, containing the version of the API (and thus the features used) and the generator that distilled this file (e.g. an editor tool) (we also don't care about this)
- a block of nodes (vertices in our graph)
- a block of ways (edges in our graph)
- a block of relations (we don't care about these)

```
<?xml version='1.0' encoding='UTF-8'?>
<osm version="0.6" generator="osmconvert 0.8.4">
  <bounds minlat="37.821" minlon="-122.3269999" maxlat="37.893"
maxlon="-122.1889997"/>
  <node id="30362155" lat="37.8213443" lon="-122.2974753" version="1"/>
  <node id="30362156" lat="37.8217935" lon="-122.2982612" version="1"/>
  <node id="30362160" lat="37.8240326" lon="-122.3032181" version="1"/>
```

A Peek at the Data

Structure of our data:

- an XML suffix introducing the UTF-8 character enco
- an osm element, containing the version of the API (generator that distilled this file (e.g. an editor tool)
- a block of nodes (vertices in our graph)
- a block of ways (edges in our graph)
- a block of relations (don't care)

Each Way includes a path of nodes, and optionally a bunch of different tags.

- Extra meta information (like bicycle).
- Special note: Ignore all one-way tags since many of them are wrong.

See Way documentation for more!

```
...
<way id="21456366" version="1">
  <nd ref="1993196224"/>
  <nd ref="260910353"/>
  <nd ref="3178364013"/>
  <nd ref="3178364019"/>
  <nd ref="260910347"/>
  <nd ref="286080654"/>
  <nd ref="53073462"/>
  <nd ref="286633456"/>
  <nd ref="2769094584"/>
  <nd ref="53111679"/>
  <nd ref="53080628"/>
  <tag k="foot" v="yes"/>
  <tag k="name" v="Bancroft Way"/>
  <tag k="oneway" v="yes"/>
  <tag k="bicycle" v="yes"/>
  <tag k="highway" v="secondary"/>
  <tag k="maxspeed" v="25 mph"/>
  <tag k="sidewalk" v="both"/>
  <tag k="tiger:cfcc" v="A41"/>
  <tag k="tiger:county" v="Alameda, CA"/>
  <tag k="tiger:zip_left" v="94704"/>
  <tag k="tiger:name_base" v="Bancroft"/>
  <tag k="tiger:name_type" v="Way"/>
  <tag k="tiger:zip_right" v="94704"/>
</way>
```

Regarding Ways

Some notes on the skeleton code:

- MaxSpeed is only useful if you want to play around with it. Not required for assignment.
- The name of Nodes and Ways are only useful if you are doing Autocomplete. You can ignore these if you don't plan to do the gold points.
  - Feel free to only store what is needed for routing, and you can add names if you decide to do gold points later.

The OSM file also contains lots and lots of other information. You're welcome to read it in and store it if you'd like, but you only need to keep whatever is necessary for the GraphDB and Router (and Autocomplete) methods to work properly.

Way Validity

A Way is considered valid only if it is a “highway” AND it is one of the ALLOWED_HIGHWAY_TYPES given in GraphBuildingHandler. If a given way is NOT one of these types, it should not be added to the graph.

- Note the word highway is used weirdly in OSM as it also includes things like minor roads, foot paths, etc.

Note: Some ways aren’t “highways”. For example, some of them delineate the perimeter of buildings.

- Example, way id="23226440" is the perimeter of Berkeley Bowl.

Graph Building Coding Tips

Routing Skeleton Code

The startElement method is called every time the SAXParser encounters an element.

Acts sort of like an iterator, e.g. your startElement method will get called once for each line of the code to the right.

endElement is called at the end of any element, e.g. anything that looks like </way> or </node>.

- A tag with a slash / denotes an end.

```
...
<way id="21456366" version="1">
  <nd ref="1993196224"/>
  <nd ref="260910353"/>
  <nd ref="3178364013"/>
  <nd ref="3178364019"/>
  <nd ref="260910347"/>
  <nd ref="286080654"/>
  <nd ref="53073462"/>
  <nd ref="286633456"/>
  <nd ref="2769094584"/>
  <nd ref="53111679"/>
  <nd ref="53080628"/>
  <tag k="foot" v="yes"/>
  <tag k="name" v="Bancroft Way"/>
  <tag k="oneway" v="yes"/>
  <tag k="bicycle" v="yes"/>
  <tag k="highway" v="secondary"/>
  <tag k="maxspeed" v="25 mph"/>
  <tag k="sidewalk" v="both"/>
  <tag k="tiger:cfcc" v="A41"/>
  <tag k="tiger:county" v="Alameda, CA"/>
  <tag k="tiger:zip_left" v="94704"/>
  <tag k="tiger:name_base" v="Bancroft"/>
  <tag k="tiger:name_type" v="Way"/>
  <tag k="tiger:zip_right" v="94704"/>
</way>
```

Routing Skeleton Code

Uncomment the provided print statements to get a better sense of how things work. Consider using the debugger and setting a breakpoint to avoid just dumping a bajillion lines of text.

- Consider using GraphDBLauncher (instead of MapServer) to explore.

```
if (qName.equals("node")) {
   /* We encountered a new <node...> tag. */
   activeState = "node";
   System.out.println("Node id: " + attributes.getValue("id"));
   System.out.println("Node lon: " + attributes.getValue("lon"));
   System.out.println("Node lat: " + attributes.getValue("lat"));

   /* TODO Use the above information to save a "node" to somewhere. */
   /* Hint: A graph-like structure would be nice. */

}
```

Way Validity

When you first encounter a way, you don't know if it is a "highway" or something else.

- Have to keep track of all the nds since they might be intersections as part of a road.
- Only at the purple line do you realize it's not a highway.

Note: You don't need to write a bunch of statements for everything that might go wrong, see next slide.

```
<way id="23226440" version="1">
      <nd ref="251271013"/>
      <nd ref="251271014"/>
      <nd ref="251271015"/>
      <nd ref="251271016"/>
      <nd ref="251271017"/>
      <nd ref="251271018"/>
      <nd ref="251271021"/>
      <nd ref="251271022"/>
      <nd ref="251271026"/>
      <nd ref="251271027"/>
      <nd ref="3187950917"/>
      <nd ref="3187950918"/>
      <nd ref="251271013"/>
      <tag k="url"
v="http://www.berkeleybowl.com"/>
      <tag k="name" v="Berkeley Bowl"/>
      <tag k="shop" v="supermarket"/>
      <tag k="building" v="yes"/>
      <tag k="addr:city" v="Berkeley"/>
      <tag k="wheelchair" v="yes"/>
      <tag k="addr:street" v="Oregon
Street"/>
      <tag k="addr:housenumber" v="2020"/>
</way>
```

Way Validity

The way on the right corresponds to a section of University Avenue.

We know that we should have edges between nodes 2131737582 - 258758044 - 52987320 - 74261534 - 258758047

- Valid way since it's a highway and it's in the ALLOWED_HIGHWAY_TYPES

Recommended approach in skeleton:

- Set a valid boolean (a.k.a. "flag") when you see a valid highway tag.

```
<way id="23873792" version="1">
  <nd ref="2131737582"/>
  <nd ref="258758044"/>
  <nd ref="52987320"/>
  <nd ref="74261534"/>
  <nd ref="258758047"/>
  <tag k="hgv" v="designated"/>
  <tag k="foot" v="no"/>
  <tag k="name" v="University Avenue"/>
  <tag k="lanes" v="2"/>
  <tag k="layer" v="1"/>
  <tag k="bridge" v="yes"/>
  <tag k="oneway" v="yes"/>
  <tag k="bicycle" v="no"/>
  <tag k="highway" v="primary"/>
  <tag k="surface" v="paved"/>
  <tag k="cycleway" v="no"/>
  <tag k="maxspeed" v="35 mph"/>
  <tag k="sidewalk" v="none"/>
  <tag k="tiger:cfcc" v="A41"/>
  <tag k="source:name" v="local knowledge"/>
  <tag k="utility_wires" v="underground"/>
</way>
```

Exploratory (Red-Shirt) Coding

There's a nice writeup (that I linked earlier in the semester) on "red-shirt" coding: http://ryantablada.com/post/red-green-refactor---a-tdd-fairytale

The basic idea:

- Feel free to write little bits of code to build a mental model of what you're trying to accomplish and how everything works.

However, don't try too hard to guess-and-check your way to success.

- If you find yourself frustrated, rather than make random permutations to your code in the hopes of it working, take a step back (or sleep on it) and come back with a more disciplined approach.

GraphBuildingTest, GraphDBLauncher, and MapServer

Three provided ways to interact with your GraphDB class:

- MapServer (the complete large system that we're building)
- GraphDBLauncher (smaller launcher just for GraphDB)
- GraphBuildingTest (provides very basic checks)

As a basic sanity check for your graph construction, use GraphBuildingTest.

- Might also help you understand what you're supposed to do.
- Provides feedback for some of the most common bugs from Spring 2016/2017.

~~Note on Spec Modification: Distance Method~~

~~An earlier version of the skeleton code and the spec required that your distance method return the distance between two vertices measured in units of longitude.~~

- ~~These units were wrong. The comment should have read "Returns the Euclidean distance between vertices v and w, where Euclidean distance is defined as sqrt((lonV - lonV)^2 + (latV - latV)^2)."~~

This was a leftover slide from Spring 2017. Ignore.

Your Graph Representation

It is your choice how to implement the Graph. What is important is that after we use your constructor that our six required methods behave properly.

- vertices, adjacent, distance, closest, lat, lon
- Linear time closest is fine.

To make these work you'll need some core graph methods like:

- addNode
- addEdge
- removeNode

Princeton Graph API is good for inspiration, but it does not have addNode or removeNode.

Routing Coding Tips

Implementing A*

What follows are four different ideas about how to implement A*, collected from looking at student code.

- Identifying which approach you used might be helpful for you in debugging, so you can compare your implementation to the given descriptions.

Note: Approach 2 seems to be the easiest to get right.

Warning: Premature optimization is the root of all evil.

Known Student Approaches

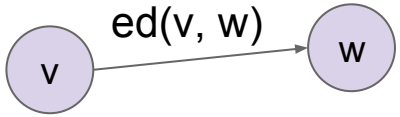
#	What starts in fringe	**New** vertices can be **added** to **fringe**?	Vertex **priorities** can be **updated** in fringe?
1 (same as lecture) (video link)	Everything	No	Yes
2 (same as HW4) (video link)	Source only	Yes	No
3 (video link)	Source only	Yes	Yes
4 (video link)	In approach 4, students attempt to adapt PuzzleSolver, see notes at end.		

For approaches 2 and 3, you may optionally use a “marked” data structure to reduce the number of times that any given vertex is enqueued.

Relaxation

Part of any approach involves relaxation (a.k.a. “better than checking”), e.g.

- If w is a neighbor of v:
  - If d(s, v) + ed(v, w) is less than d(s, w) in **best**:
    - Update **best** and fringe



d(s, v): best known distance from s to v
ed(v, w): great circle distance from v to w
h(w): great circle distance from w to goal

best is some data structure(s) that should keep track of:

- Best known distance from source to every vertex w, i.e. d(s, w)
- “Parent” of every vertex (similar to edgeTo in lecture, i.e. edgeTo[w]).

Probably the easiest idea: Two maps, one from vertex number to best known distance, and the other from vertex number to best vertex number.

- But students tried other things, like storing the information inside GraphDB nodes. This can work, too.

Most Common Tricky Bugs

Some of the most common hard-to-spot bugs:

- Setting d(s, w) = d(s, v) + ed(v, w) **+ h(w)** in **best**.
  - Distance from source to w should not include h(w) in **best**!
- Not including heuristic distance to goal h(w) in calculation of priority in fringe.
  - This actually works fine, but it's Dijsktra's and thus slow.
- Changing priorities of objects already inside a java.util.PriorityQueue.
  - This can result in the wrong node being returned by poll.
- Using a marked set inappropriately (e.g. marking when a vertex when it is added to fringe). See Approach #2X slides.
- Letting shortestPaths mutate your graph. When shortestPaths completes, your GraphDB object should be exactly the same as when it started (i.e. don't leave a bunch of previous pointers and distances set).

A* Implementation Philosophy 1: (video link)

Approach 1: Enqueue Everything First, with Priority Adjustment (**EEFPA**)

- Data structures: fringe (priority queue), **best** (?)
- Start by adding the all vertices to the fringe, such that all vertices have infinite priorities, except the source with priority 0. Initialize **best**.
- Dequeue the closest vertex from the fringe, which we'll call v.
  - If v is the goal, we're done.
  - For each edge v ⟶ w, relax that edge, where relax means:
    - If d(s, v) + ed(v, w) is less than d(s, w) in **best**:
      - update **best** so that d(s, w) = d(s, v) + ed(v, w)
      - If present, **update w's priority** on the fringe to be d(s, v) + ed(v, w) + h(w)

d(s, v): best known distance from s to v
ed(v, w): euclidean distance from v to w
h(w): euclidean distance from w to goal

This is what we did in lecture, but it requires the ability to update priorities which can be tricky.

Very easy to do this either too slowly (linear time) or incorrectly. More details follow in two slides.

Priority Updating

To “update” the priority you can’t just change an instance variable somewhere and expect the item to be in the right place (see below). Updating priority requires you to either:

- Use a priority queue with a special “change priority” operation like in lab 10’s bonus exercise. Java’s PriorityQueue class doesn’t have this operation!
- Delete the item out of the priority queue and re-insert it.
  - Warning: This operation is linear time with Java’s PriorityQueue class.
  - May be too slow to pass autograder.

Why can’t you just change the priority?

- Imagine you change the root’s priority by setting node.distance = 2837432. Without swimming or sinking it might no longer be in the right place.

Avoiding Priority Updating

To avoid the annoyance of updating priorities of objects already in the fringe, you can also allow multiple copies of the same vertex on the fringe.

- If you don’t start off every vertex with infinite priority, you may as well as also only start with the source on the fringe.

This approach is detailed in Pieter Abbeel’s video, linked in the spec, but it is written out in the next two slides for your convenience.

A* Implementation Philosophy 2 (video link)

Approach 2: Enqueue Only the Source, Allow Multiple Copies (**EOSAMC**)

- Data structures: fringe (priority queue), **best** (?)
- Start by adding the source to fringe. Initialize **best**.
- Dequeue the closest vertex from the fringe, which we'll call v.
  - If v is the goal, we're done.
  - For each edge v ⟶ w, relax that edge, where relax means:
    - If d(s, v) + ed(v, w) is less than d(s, w) in **best**:
      - update **best** so that d(s, w) = d(s, v) + ed(v, w)
      - add w to the fringe with a priority equal to d(s, v) + ed(v, w) + h(w).

d(s, v): best known distance from s to v
ed(v, w): euclidean distance from v to w
h(w): euclidean distance from w to goal

This may result in multiple copies of w on the fringe. That's OK, though depending on your code you might need to add a marked data structure to pass timing tests (see next slide).

This is the version in the Pieter Abbeel video linked in the spec and is also used by the reference solution.

A* Implementation Philosophies

You shouldn't need a marked set to pass the timing tests.

Approach 2X: Enqueue Only the Source, Allow Multiple Copies, with Marked Set (**EOSAMCMS**)

- Data structures: fringe (priority queue), marked (?), best (?)
- Start by adding the source to fringe. Initialize best.
- Dequeue the closest vertex from the fringe, which we'll call v.
  - If v is marked, ~~return~~ skip to the next node, i.e. `continue`.
  - If v is the goal, we're done.
  - Add v to the marked set.
  - For each edge v → w, relax that edge, where relax means:
    - If d(s, v) + ed(v, w) is less than d(s, w) in best:
      - update best so that d(s, w) = d(s, v) + ed(v, w)
      - add w to the fringe with a priority equal to d(s, v) + ed(v, w) + h(w).

This may result in multiple copies of w on the fringe, each with different priorities. That's OK.

Warning: If you mark a vertex when it is added to the fringe, your code will fail.

Must be when you dequeue!

Priority Updates Without Enqueuing the Entire Graph

If you aesthetically dislike the idea of allowing multiple copies (i.e. you want to do priority updates) and ALSO dislike enqueuing the entire graph with infinite priority, there's a philosophy for you, too.

In this approach, you enqueue only the source at first AND use updates.

A* Implementation Philosophy 3 (video link)

Approach 3: Enqueue Only the Source, with Priority Adjustment (**EOSPA**)

- Data structures: fringe (priority queue), **best** (?)
- Start by adding the source to fringe. Initialize **best**.
- Dequeue the best vertex from the fringe, which we'll call v.
  - If v is the goal, we're done.
  - For each edge v → w, relax that edge, where relax means:
    - If d(s, v) + ed(v, w) is less than d(s, w) in **best**:
      - update **best** so that d(s, w) = d(s, v) + ed(v, w)
      - If w is on the fringe, update its priority to be d(s, v) + ed(v, w) + h(w), or if it is not there, add w to the fringe with priority d(s, v) + ed(v, w) + h(w)

d(s, v): best known distance from s to v
ed(v, w): euclidean distance from v to w
h(w): euclidean distance from w to goal

This may result in w being added back to the fringe after it is removed. That's OK since all relaxations of the "repeat" are guaranteed to have no effect. If you don't like that idea, see the next slide.

A* Implementation Philosophy 3

Approach 3X: Enqueue Only the Source, with Priority Adjustment, with Marked Set (**EOSPAMS**)

- Data structures: fringe (priority queue), marked (?), **best** (?)
- Start by adding the source to fringe. Initialize **best**.
- Dequeue the closest vertex from the fringe, which we'll call v.
  - If v marked, return. Mark v. If v is the goal, we're done.
  - For each edge v → w, relax that edge, where relax means:
    - If d(s, v) + ed(v, w) is less than d(s, w) in **best**:
      - update **best** so that d(s, w) = d(s, v) + ed(v, w)
      - If w is on the fringe, update its priority to be d(s, v) + ed(v, w) + h(w), if w is unmarked and not in PQ, add w to the fringe with priority d(s, v) + d(v, w) + h(w)

A* Implementation Philosophy 4 (video link)

Approach 4: Didn't we do this in HW4?

- Just override neighbors and estimatedDistanceToGoal and use HW3 solver.
- Potential issues:
  - Unlike HW4, edges don't all have equal weights.
  - Some vertices may get enqueued tons of times. The "critical optimization" in HW4 may not be good enough.
    - If it's too slow, see philosophy 2X.
  - Overhead for building SearchNode objects is potentially too high.
    - This might not actually be a big deal (we haven't tested).

It is probably possible to get this approach to work, but it is untested.
Ultimately your approach will end up being similar to approach #2.

General Tips on A*

If you use the GraphDB itself to store the results of your A*, be careful. If shortestPaths changes anything in your GraphDB, make sure to reset it when you finish your call to shortestPaths!

- This is a seemingly rather common bug.
- **If you're getting JSON errors that only appear when you have shortestPaths, this is probably why.**
  - When shortestPaths finishes execution, your GraphDB object should be back to the way it was before the method started running.

(Other possible cause of JSON errors in autograder: Storing too much stuff in your GraphDB)

General Tips on A*

If your routing code is yielding the wrong answer try getting rid of non-essential features that only serve to speed up your code.

- Approach 2 and 3: Comment out marked sets (or other similar data structures that serve only to speed up your code).
- Approach 1 and 3: Switch from log N priority updating to linear time priority updating (where you just delete the old copy of w).
- Turn your A* into Dijkstra by giving vertices on the PQ a priority equal to d(s, v) + ed(v, w). That is, ignore the h(w).

Or start over. I rewrote an implementation of shortestPaths from scratch in 20 minutes while putting together these slides. Trying to bend subtly broken code like A* into working shape can be much harder than just rewriting from scratch. My method was 68 lines long including helper functions and helper class.