CSE 490H 08au
Assignment 3

Rendering Map Data

Administrative Details

Assigned: Tue 11/4/08
Due: Tue 11/18/08 at 4:30 pm

Partners: Max of 2 people per group

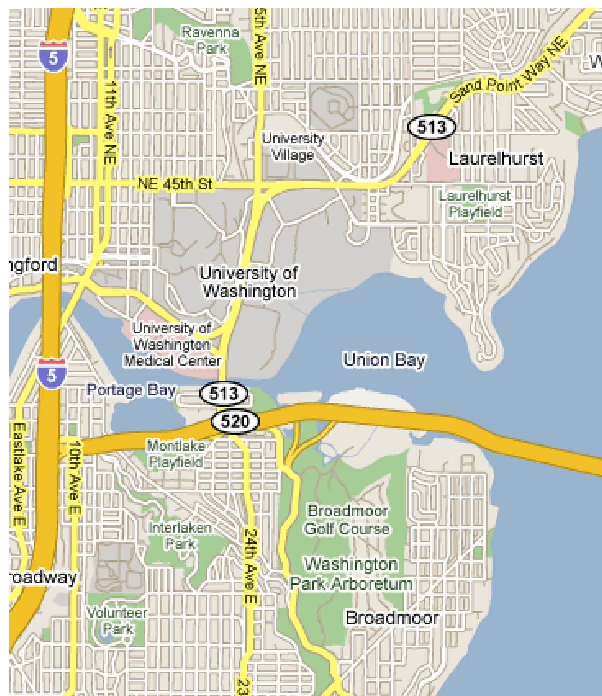
If you would like, you may pick a partner for this project. This will also be your partner for project 4 -- so choose wisely ;)

Submit: via turnin on attu, as usual

Starter Source Code: http://www.cs.washington.edu/education/courses/cse490h/08au/projects/geosource.zip

Introduction

A picture's worth a thousand words, so... the shortest description of this assignment is, do this:



You will be using geographic survey data and census data to render a map of the major transportation/geographic features of the United States. The United States Census (www.census.gov) produces a dataset describing all roads, highways, cities, civic works

(hospitals, parks, etc), bodies of water, etc. in the United States. This data set is referred to as the TIGER dataset (Topologically Integrated Geographic Encoding and Referencing system). We are using the "2006 TIGER/Line" dataset.

This project relies on processing the raw TIGER input data, combining it with other census data (e.g., population estimates for municipalities), and rendering the map of the US as a set of tiles. We will use Hadoop to perform this processing in parallel.

In the next assignment, you will take your computed data and host it on Amazon's EC2 and S3 platform, building a dynamic web application to allow access to your information. Effectively, assignment 3 is the Google Maps "backend" assignment, and assignment 4 will be building (a simplified version of) the Google Maps frontend without traffic direction routing.

We have provided starter code at http://www.cs.washington.edu/education/courses/cse490h/08au/projects/geosource.zip -- you should use this as the base for your project, conforming to its interfaces and extending it where indicated.

Tiles, Ranges, Zoom Levels, and Tile Sets

It is inefficient and unwieldy to attempt to render the entire map, at every zoom level you would like, as a single entity. So we break the map into tiles.

A **tile** is a square region of the map corresponding to the smallest unit we render at a time. The view that you display in a web browser is made of several tiles arranged in a 2d grid. A tile is defined by a **tile extent** which is a northern-most and a southern-most latitude, and a western-most and eastern-most longitude. We are making a major simplifying assumption in this assignment, that latitude and longitude are a rectangular coordinate system. This will result in some rendering inaccuracy, but for the latitudes in the USA, this is acceptable.

The tiles are arranged in a rectangular grid across a **mappable range**. The mappable range is the entire area we wish to render (e.g., the whole USA for your final production run, or maybe a few square miles for testing). A mappable range is a square (assuming rectilinear latitude/longitude) defined by the latitudes and longitudes of its edges. We break the mappable range into a 2D array of tiles.

Within this 2D array each tile has an ID. A **tile ID** is an (x, y) index into the 2D array that spans the mappable range.

So if we had 9 tiles across some mappable range, their tile IDs may be:

```
(0, 0)    (1, 0)    (2, 0)
(0, 1)    (1, 1)    (2, 1)
(0, 2)    (1, 2)    (2, 2)
```

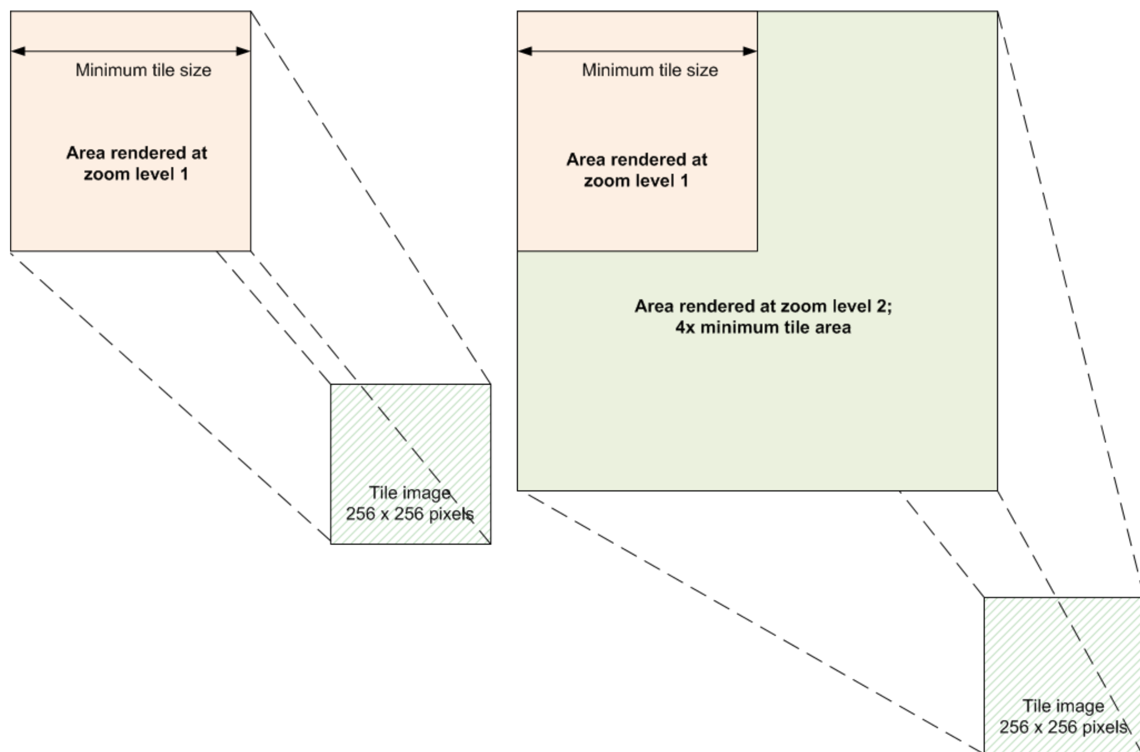
We will render tiles using an algorithm similar to the one described in Barry's lecture in class. The mappers will read all the features in our map data, and emit them to the reducers responsible for rendering them onto tiles. We will precalculate which tile(s) are responsible for which features, and the mappers will send the features to the reducer(s) for those tile(s). In the reducer, we will then render each output image.

We do not want to use a separate reducer for each tile; this is far too many reduce processes. Instead, each reducer will be responsible for rendering a **tile set**. A tile set is simply a set of tiles which will be rendered by a single reducer. These tiles do not necessarily need to be adjacent to one another (although, since a feature like a long

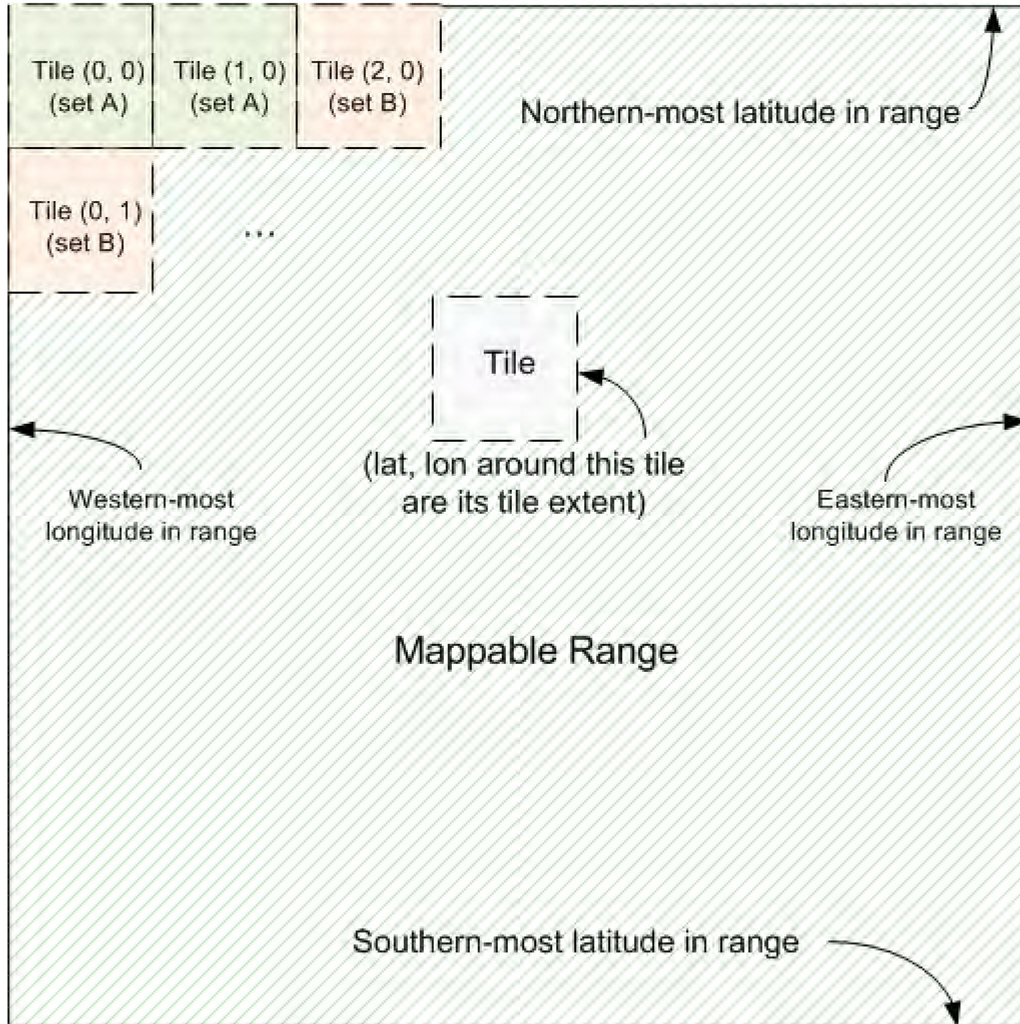
road segment may be rendered on multiple tiles, it would be nice if we could minimize the amount of data motion required between mappers and reducers).

The **minimum tile size** is a constant defined in the code which is the smallest width and height in degrees of lat/lon (we assume these are equal in nature) which corresponds to an individual tile. Do not change this constant, as changing this determines how fine the tile grid is, and has a dramatic effect on cluster utilization. Each tile is defined to be 256 x 256 pixels.

The **zoom level** is an integer >= 1, which is an exponential multiplier of the minimum tile size that determines how much land area is rendered onto a tile. At zoom level 1, the minimum tile size of land area is rendered onto a 256x256 pixel tile; this gives the closest zoom and the most detailed features. At zoom level 2, we take 2x the longitude in width, and 2x the latitude in height, and render that much land onto the same sized tile image. Zoom level 3 is 4x the width, 4x the height (16x the area), etc.



One of the components you must implement is a **tile set divider** which, given the number of reducers, the mappable range, and the intended zoom level to render, will break the mappable range into tiles, assign them tile IDs, and group the tile IDs into as many tile sets as there are reducers. This tile set divider is responsible for maintaining a 2-way mapping: it must be able to return a tile ID for any (lat, lon) in the mappable range, and given a tile ID, it must return the (lat, lon) of the northwest (top-left) corner of its tile extent. (Note that this is not neccessarily a direct inverse operation.)



This figure shows a mappable range broken up into several tiles (only some of the tiles are drawn in). The tiles have been broken up into two tile sets, "A" and "B." Tiles in the same tile set have the same background color. The mappable range is a square of (lat, lon) coordinates, as are each tile. The tile set divider determined the number of tiles to build based on the zoom level provided as an initialization parameter.

The same mappable range (e.g., "the United States") will be passed through the render process several times, one for each zoom level we intend to render.

Starter Source Code

We have provided source code for you to use to get started. These should strongly inform the design of your system. All of the source code is under the package **`edu.washington.cse490h.geo`**.

This contains the following subpackages:

- **`protocol`** - Defines data types which are marshaled from Mapper to Reducer and from Reducer to the next phase's Mapper.
- **`mapred`** - This is where all your mapper and reducer classes should go

Test Data Set

Use the datasets given below for local testing and debugging. Do not use the entire US dataset for debugging. Download the King County TIGER/Line data from:

http://www2.census.gov/geo/tiger/tiger2006se/WA/TGR53033.ZIP.

This is about 10 MB compressed. Do all your local testing on this data set. Only run on the cluster after everything works perfectly locally.
Download the BGN dataset for Washington state here:

http://geonames.usgs.gov/docs/stategaz/WA_Features_20080815.zip

Download the Population dataset for Washington state here:

http://www.cs.washington.edu/education/courses/cse490h/CurrentQtr/projects/wa_pop.txt

Data Extraction and Data Types

The TIGER data files come as a set of fields organized in records. Each record is a single line of text. The fields are **fixed width**, which means that there are no commas, tabs, or other delimiters marking the edges of fields. Instead, we know how many characters each field can take up, and all the text in those character designations is the field.

For example, the following record has fields "A", "B", and "C", and is 15 characters wide. If A was a 3-character field, B is a 6-character field, and C is a 6-character field, the fields would occupy positions:

```
0123456789ABCDE (position in hex)
AAABBBBBBCCCCCC (corresponding field)
```

Some example records in this format may look like:

```
1  Foo   542571
276Record1337
```

Note how records that do not occupy their full width are padded on one side or the other. (The specification determines whether fields are left- or right-side-padded.) Other fields that occupy their full width run up against the adjacent fields.

The first challenge you must face is parsing your input data set.
The TIGER data provides several types of records. As extensions, you may try to parse several of these record types and incorporate them into your output tiles. At minimum, you must parse the first two record types. These are:

1 - Complete chain basic data (line data)
2 - Complete chain shape coordinates (polygon data)

The classes protocol.TigerRecordType1 and protocol.TigerRecordType2 describe the fields of these two record types that we are interested in. We have provided you with the data fields; it is your job to write the readFields() and write() methods, and write the parser which turns a textual record into one of these objects.

Below is reproduced the pages from the TIGER data dictionary, which defines the records precisely:

Record Type 1 – Complete Chain Basic Data Record

Field	BV	Fmt	Type	Beg	End	Len	Description
RT	No	L	A	1	1	1	Record Type
VERSION	No	L	N	2	5	4	Version Number
TLID	No	R	N	6	15	10	TIGER/Line(R) ID, Permanent 1-Cell Number
SIDE1	Yes	R	N	16	16	1	Single-Side Source Code
SOURCE	Yes	L	A	17	17	1	Linear Segment Source Code
FEDIRP	Yes	L	A	18	19	2	Feature Direction, Prefix
FENAME	Yes	L	A	20	49	30	Feature Name
FETYPE	Yes	L	A	50	53	4	Feature Type
FEDIRS	Yes	L	A	54	55	2	Feature Direction, Suffix
CFCC	No	L	A	56	58	3	Census Feature Class Code
FRADDL	Yes	R	A	59	69	11	Start Address, Left
TOADDL	Yes	R	A	70	80	11	End Address, Left
FRADDR	Yes	R	A	81	91	11	Start Address, Right
TOADDR	Yes	R	A	92	102	11	End Address, Right
FRIADDL	Yes	L	A	103	103	1	Start Imputed Address Flag, Left
TOIADDL	Yes	L	A	104	104	1	End Imputed Address Flag, Left
FRIADDR	Yes	L	A	105	105	1	Start Imputed Address Flag, Right
TOIADDR	Yes	L	A	106	106	1	End Imputed Address Flag, Right
ZIPL	Yes	L	N	107	111	5	ZIP Code(R), Left
ZIPR	Yes	L	N	112	116	5	ZIP Code(R), Right
AIANHHFPL	Yes	L	N	117	121	5	FIPS 55 Code (American Indian/Alaska Native Area/Hawaiian Home Land), 2000 Left
AIANHHFPR	Yes	L	N	122	126	5	FIPS 55 Code (American Indian/Alaska Native Area/Hawaiian Home Land), 2000 Right
AIHHTLIL	Yes	L	A	127	127	1	American Indian/Hawaiian Home Land Trust Land Indicator, 2000 Left
AIHHTLIR	Yes	L	A	128	128	1	American Indian/Hawaiian Home Land Trust Land Indicator, 2000 Right
CENSUS1	Yes	L	A	129	129	1	Census Use 1
CENSUS2	Yes	L	A	130	130	1	Census Use 2
STATEL	Yes	L	N	131	132	2	FIPS State Code, 2000 Left (always filled both sides, except at U.S. boundaries)
STATER	Yes	L	N	133	134	2	FIPS State Code, 2000 Right (always filled both sides, except at U.S. boundaries)
COUNTYL	Yes	L	N	135	137	3	FIPS County Code, 2000 Left (always filled both sides, except at U.S. boundaries)
COUNTYR	Yes	L	N	138	140	3	FIPS County Code, 2000 Right (always filled both sides, except at U.S. boundaries)
COUSUBL	Yes	L	N	141	145	5	FIPS 55 Code (County Subdivision), 2000 Left
COUSUBR	Yes	L	N	146	150	5	FIPS 55 Code (County

							Subdivision), 2000 Right
SUBMCDL	Yes	L	N	151	155	5	FIPS 55 Code (Subbarrio), 2000 Left
SUBMCDR	Yes	L	N	156	160	5	FIPS 55 Code (Subbarrio), 2000 Right
PLACEL	Yes	L	N	161	165	5	FIPS 55 Code (Place/CDP), 2000 Left
PLACER	Yes	L	N	166	170	5	FIPS 55 Code (Place/CDP), 2000 Right
TRACTL	Yes	L	N	171	176	6	Census Tract, 2000 Left
TRACTR	Yes	L	N	177	182	6	Census Tract, 2000 Right
BLOCKL	Yes	L	N	183	186	4	Census Block Number, 2000 Left
BLOCKR	Yes	L	N	187	190	4	Census Block Number, 2000 Right
FRLONG	No	R	N	191	200	10	Start Longitude
FRLAT	No	R	N	201	209	9	Start Latitude
TOLONG	No	R	N	210	219	10	End Longitude
TOLAT	No	R	N	220	228	9	End Latitude

BV (Blank Value):
Yes = Blank value may occur here; No = Blank value should not occur here
Fmt:
L = Left-justified (numeric fields have leading zeros and may be interpreted as character data)
R = Right-justified (numeric fields do not have leading zeros and may be interpreted as integer data)
Type:
A = Alphanumeric, N = Numeric

Record Type 2 - Complete Chain Shape Coordinates

Field	BV	Fmt	Type	Beg	End	Len	Description
RT	No	L	A	1	1	1	Record Type
VERSION	No	L	N	2	5	4	Version Number
TLID	No	R	N	6	15	10	TIGER/Line(R) ID, Permanent 1-Cell Number
RTSQ	No	R	N	16	18	3	Record Sequence Number
LONG1	No	R	N	19	28	10	Point 1, Longitude
LAT1	No	R	N	29	37	9	Point 1, Latitude
LONG2	Yes	R	N	38	47	10	Point 2, Longitude
LAT2	Yes	R	N	48	56	9	Point 2, Latitude
LONG3	Yes	R	N	57	66	10	Point 3, Longitude
LAT3	Yes	R	N	67	75	9	Point 3, Latitude
LONG4	Yes	R	N	76	85	10	Point 4, Longitude
LAT4	Yes	R	N	86	94	9	Point 4, Latitude
LONG5	Yes	R	N	95	104	10	Point 5, Longitude
LAT5	Yes	R	N	105	113	9	Point 5, Latitude
LONG6	Yes	R	N	114	123	10	Point 6, Longitude
LAT6	Yes	R	N	124	132	9	Point 6, Latitude
LONG7	Yes	R	N	133	142	10	Point 7, Longitude

```
LAT7     Yes R   N    143 151  9    Point 7, Latitude
LONG8    Yes R   N    152 161  10   Point 8, Longitude
LAT8     Yes R   N    162 170  9    Point 8, Latitude
LONG9    Yes R   N    171 180  10   Point 9, Longitude
LAT9     Yes R   N    181 189  9    Point 9, Latitude
LONG10   Yes R   N    190 199  10   Point 10, Longitude
LAT10    Yes R   N    200 208  9    Point 10, Latitude


Note:
The TIGER/Line(R) files contain a maximum of ten shape coordinates on
one record. The number of shape records for a complete chain may be
zero, one, or more. Complete chains with zero shape points (a straight
line) do not have a Record Type 2. Coordinates have an implied six
decimal places. See the Positional Accuracy section in Chapter 5 for
more details.
```

All TIGER records have a 1 character Record Type (RT) field as their first field; this identifies the type of the record you are parsing.

Joining

Looking at the datatypes above, it is clear that the polygon files contain the points which make up the polygon, but does not say what the polygon actually is. The polygon record, however, contains a Tiger/Line ID. This is the same id as some record of type 1, which describes a line. This line is adjacent to one of the sides of the polygon. The line's record describes what type of object the polygon represents (e.g., body of water, school, park, etc). This is stored in its CFCC field (Census Feature Code Class). A MapReduce pass must join the polygon to its associated line record to fill out a more complete record body to determine the type of the polygon, and thus how to render it.

There is a second join between the BGN and Population datasets. BGN records contain the identifier for a location (city and state names), and its latitude and longitude coordinates. The population dataset has the name of the city/state entity, and its population -- but not its latitude and longitude. You should use MapReduce to execute a join so that you can complete the population entity data structure by filling in the latitude and longitude from the corresponding BGN record.

> The protocol.BgnRecord and protocol.PopRecord classes contain the names of all the imporatant fields; they also contain comments describing from where these fields are filled. You should read these comments before implementing the parsing for these records. We have already provided you with a complete parser for the population records; you must implement parsers for BGN and the TIGER datatypes.

If you are confused about how to execute a join, refer to Barry Brumitt's slides (posted on the course web site).

Rendering

The primary component of this assignment is the tile renderer. Tile rendering is done by subclasses of the **TileRenderer** abstract class. You must implement a concrete subtype of this class that renders important geographic features on tiles.

We have provided you with an example of how to implement this with the **FakeTileRenderer** class. The FakeTileRenderer will draw the same picture on every tile, regardless of its input features. The purpose of this class is to familiarize you with the graphics primitives necessary to perform the drawing. This renderer will render a line, a polygon, and a text object; it shows you how to change colors and fonts. When you are stuck on how to do something, refer to the example uses in this file, and of course, always refer to the Java API documentation online for specific methods to use and types of arguments.

The TileRenderer is used in a Reducer. The Reducer receives all shape object records for a given Tile Set. The Mapper should read in all shape object records, determine which tile or tiles they cover, and emit them to the reducers for the Tile Sets containing the tile IDs in question.

When features are read in by the Reducer, they are added to the collection seen by the TileRenderer with the addRecord() method. After buffering all its features, the Reducer invokes the TileRenderer repeatedly to render every Tile in its Tile Set. The setTile() method is used to set the tile extent which should be rendered. Feature records which are visible (partially or fully) within this tile should then be rendered on to the output buffer when renderTile() is called. You must provide the body of renderTile() to draw things nicely.

How elaborate you want to get with your rendering is up to you. We appreciate interesting refinements to your rendering algorithm, which make your tiles "prettier" in any way. You should document any extensions or tweaks you make here.

At minimum, your renderer must do the following:

- Render all roads and highways at low zoom levels
- Render major highways at farther-out zoom levels
- Make a reasonable effort to abstract away densely-developed areas at high zoom levels (e.g., render "some roads," make roads fade into the background, etc, so that Seattle appears neither a solid mass of black when zoomed out, but nor is it just the intersection of 520 and I-5)
- Do a "best effort" to draw polygon objects (see below) at all zoom levels
- Place labels on top of the tiles so they are legible

Label Placement

Label placement is a difficult problem (optimum label placement is, in fact, NP-Complete). You must draw labels on your maps. You must invent an algorithm to place labels "reasonably." How you do this is up to you. Document this thoroughly in your writeup, and defend why it is a reasonable algorithm.

Your algorithm should follow these guidelines:

- At each zoom level, "appropriate" labels should be shown (surface roads at low zoom levels, cities at farther-out zoom levels, etc)
- Labels should be legible (they should not be hard to read because of lines or polygons underneath of them; they should not overlap one another, etc
- When labels might overlap or otherwise conflict, employ a conflict resolution strategy

    - Bigger cities should take priority over smaller cities
- Labels should not overly clutter your map
    - At wider zoom levels, do not draw a label for every surface road; "some roads" is a good idea (how do you choose which roads?)
    - Don't draw an excessive number of the same label (i.e., don't relabel I-5 every 25 pixels)
    - Labels should be chosen so they are relatively uniformly distributed across the tile (don't have 30 labels all in the corner)
- Font sizes chosen for labels should reflect the stature of the object they represent (cities should have more prominent labels than roads; larger cities should have more prominent labels than smaller cities)

Polygons

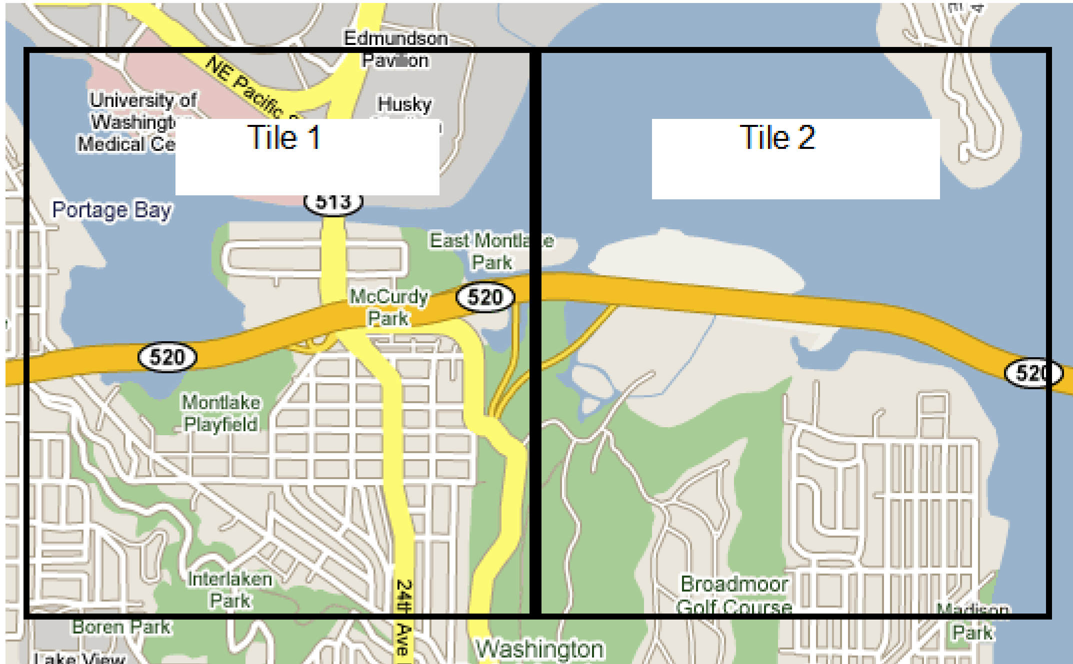
Polygon drawing is tricky and we don't have a good solution. The points which correspond to each polygon are not presented in the type-2 record in order; you cannot necessarily draw them clockwise (for example) and expect the correct shape to come out. More problematically, they allow concave shapes, so standard "lasso" techniques to resolve the polygons do not necessarily work. Do something reasonable here, but it is okay if polygons are rendered poorly.

Tile Set Divider Algorithms

The Mapper and Reducer for tile rendering rely on a TileSetDivider to inform them which tile IDs are grouped together in which reducers.

We have provided you with a SimpleDivider class which implements the TileSetDivider interface. This will divide the mappable range into a set of tile extents, assign them tile IDs, and assign the tile IDs to as many tile sets as there are reducers. The tile sets are created in a very naive way.

Imagine a line segment (e.g., road) that spans two adjacent tiles. This road feature data must be sent to the reducers for the tile sets for both tiles.



Consider SR-520 in the above diagram. Assuming a slightly simplified version where this is a single straight line segment, it can extend to 2 or more tiles. The Mapper that reads in the record for SR-520 must emit it to each Reducer for the tilesets corresponding to Tile 1 and Tile 2 (and possibly others, if the length of this line segment is very long). Your TileSetDivider can tell you the tile IDs (and thus the tile sets) for the endpoints of the line segment; from there you can determine all tile IDs and thus all tile sets that need a copy of the record. Don't forget, if a long line segment is oriented "diagonally," it may need to be emitted to a rectangle of tile IDs.

The SimpleDivider is very inefficient -- tile sets do a poor job of taking into account the locality of tiles to one another. We would like you to implement a better divider, which you should call the **HilbertDivider.** The HilbertDivider will draw Hilbert curves over the 2D array of tile IDs. A Hilbert curve is a **space-filling curve**: It will draw a "snake" that runs through all points in the space (the 2D array of tile IDs), and never overlaps with itself.

Linear regions of the Hilbert curve are used as the set of tile IDs comprising a given tile set. There is a mathematical theorem that shows that the Hilbert curve is the optimal partitioning of tile IDs into tile sets, to use minimum bandwidth for grouping tiles together that are likely to share features. Which is to say, given that we cannot predict whether there are more north-south or east-west roads that are shared between adjacent tiles, we should do "something reasonable" to make sure that tiles that are likely to share features wind up in the same tile set. The Hilbert curve is an algorithm for determining the best such partitioning

The Wikipedia article http://en.wikipedia.org/wiki/Hilbert_curve does a good job of explaining the Hilbert curve. See also http://en.wikipedia.org/wiki/Space-filling_curve

Your implementation should number the tiles in the range along the curve, then split the range into tile sets by dividing the curve into a number of segments equal to the number of the rendering tasks. The starter code includes a JUnit test case for the Hilbert divider you will write so that you can test it to see if your implementation is working correctly

Geocode Index Generation

As mentioned in the introduction, the next assignment will be a crude version of the online Google Maps interface. The goal of assignment four is to display to a user a pannable map on a web page. This web page will contain a text input where a user can type an address; the map should then jump to this address. To do this, we will need to generate a **geocode index**. The geocode index is a data structure that can quickly return the latitude and longitude for any address. From this latitude and longitude, your TileSetDivider can then look up the particular tile IDs (and thus the filenames of tiles) which correspond to the tiles the user should be shown.

We want to support primarily queries in the format of "Street address, ZIP code". (although you are welcome to implement geocoding for other types of queries as well - see the extensions section below).

Your geocode should be output in such a way that you can efficiently look up a latitude and longitude given an address. One way to structure your geocode index, for example, is to have the keys be zip codes and street names and the values be address ranges with appropriate street coordinates. Your output would then be sorted first numerically by zip code, and then alphabetically by street name. You could then seek into the index (think binary search) to quickly find the latitude and longitude for a desired address. (You may wish to consider java's RandomAccessFile class for this)

Extensions

This project has virtually limitless room for extensions. None of them are required. Here are just a few we have thought of (ranked roughly from easiest to hardest):

- Exploit other data present in the datasets. Currently we are asking you to use records type 1 and 2 from the TIGER/Line data. There are about fifteen others with a range of other useful data (census, economic census, further address resolution data, etc. Find a way to use this data in this project
- Build the geocode index to support other types of queries, some ideas for query types are:
    - City, State
    - County, State
    - State
    - Street address + city + state
- Exploit other data sets. Several examples are the National Hydrography Dataset (NHD), satellite tiles, etc
- Currently we are making a simplifying assumption that the world is a flat rectilinear coordinate system. (ie, we are not compensating for the spherical coordinates.) Correct this so the tiles represent actual squares in terms of land distance and not in terms of degree distance.
- Label placement: this is deceptively hard. There are at least three types of label placement. Point labeling(cities, features, etc), line labeling(roads, rivers, etc), and shape labeling(lakes, seas, etc). With each you must figure out which labels to place at which zoom level and how to place them optimally. With line labeling you must figure out how to place the label along the linear feature (eg the road label curves with the road). With shape labeling you must figure out how to efficiently calculate centers of polygons where to place labels. This is all complicated of course by having to figure out relative importance of labels, dealing with labels that have to cross tile edges, etc. These are all very hard problems, there are several papers that you can find about these topics if you look for them

- Driving directions. Barry mentioned a few papers that might help you here, but you are on your own….

Writeup

Answer the following questions in your writeup:

1. Put both your and your partner's names here.
2. In your own words, describe your complete pipeline. Name the most important classes, and describe how they function. Indicate design choices you had to make, what options you considered, and how you resolved them.
3. Describe your label placement algorithm. Why do you think this is a good algorithm to solve this problem?
4. Describe, in your own words, how the HilbertDivider you implemented works, and why it is important. That is, explain how it is better than the SimpleDivider that was included in the starter code.
5. Describe how the geocode index works to facilitate address queries efficiently.
6. Describe any extensions you implemented. What additional data sets or record types did you employ?
7. Describe your test plan. Include any test source code in your source code submission.
8. How did you and your partner break down the labor in this project?

How To Get Started

The above sections have described the complete pipeline for the map data preprocessor. Here are a set of concrete steps to orient you through building the various pipeline stages. This is not the only order in which you can do things. You will note that while there are some dependencies (some steps here must be completed before others), other steps are independent of one another. Make efficient use of both partners.

- Build your datatypes and record parser for TIGER data. The classes in the protocol package contain all the fields you need. You need to write string parsing code which will read in the fixed-length record types 1 and 2 from the TIGER data set. You should be able to read in the files, discard irrelevant (or corrupt) data, pull the requisite fields from those two record types, and emit them back as output with a toString() method that lets you know that you've parsed them correctly. For you to be able to get from mappers to reducers, you must implement write() and readFields(), of course, too.
- Build the joiner that can take in records of type 1 (line) and type 2 (polygon). This must populate the fields of the type 2 record with some of the type-1 fields; the fields are populated from whichever line record shares the Tiger Line ID with the type 2 record.
- Get the existing render pipeline to run start to finish, parsing the TIGER records, joining them, and emitting the joined records from a mapper to a reducer that uses the FakeRenderer to render King County as a set of fake tiles with the dummy image on every tile.
- Copy this renderer to a new class and modify this so that it actually draws the roads instead. Draw all line data as simple black 1-pixel-wide lines.
- Add polygons, labels, etc
- Discriminate between line types, drawing highways and roads differently (color, line width, label font, etc)
- Implement thresholding in the render step mapper so that only the relevant features at a given zoom level are emitted to the reducer at that zoom level.
- Implement the HilbertDivider to divide tile sets more efficiently

- Parse the BGN data records into a data type. Make sure that toString() output from this looks sane.
- Join population records with BGN data so that we have a mapping from cities to population.
- Refine your label placement and rendering algorithm (take advantage of population data where possible)
- Make all your tiles look pretty on your computer for King County.
- Write the address geocode index generator
- Run the geocode index pipeline on King County, locally.
- Think about any extensions you might want to implement (fancier tile generation, polar latitude/longitude coordinates, other TIGER record types, other data sets, etc). Obviously this optional step is to be done only when you have finished everything else.
- Run the entire pipeline on the cluster for a larger area than just king county at an intermediate zoom level (ie 6 or so)

Final Misc Information and Hints

- The full TIGER dataset is in /shared/tiger
- The TIGER dataset for King County is in /shared/tiger-king -- do testing on this
- The BGN dataset is in /shared/allstates
- The population dataset is in /shared/population
- The BGN and population datasets are small; there are no test-only versions of these
- If you are excluding certain features at a particular zoom level, do your filtering in the mapper, so that you don't waste bandwidth (and time).
- Be aware of the way the number of mappers and reducers at a particular stage affects performance.
- Zoom level has an exponential effect on the number of tiles to render (work to do). For testing, make sure you clamp down your mappable range to a very small area to test how you render close-zoom tiles, or else you'll be doing a lot of work (you should pick a range of say 4--12 tiles and render that when tweaking your renderer; don't render several hundred tiles each time until you are sure your renderer works well). Start your experimentation at around zoom level 6.
- When re-running the mapreduce pipeline, you do not need to re-run every pass every time. For example, it is fine to generate the filtered and joined data once (when you have gotten the filtering and joining right) and run only the render step when developing/testing the renderer multiple times. Note that when you run the full pipeline and some intermediate steps have already been computed (eg TIGER data has already been filtered), that step in the pipeline will fail with an exception message about the output already existing - this is OK, this simply means that this step will be skipped and the existing output will be used instead.
- Use the reporter and the job tracker page to your advantage when running on the large dataset. This can help you understand where the bottlenecks in your code are.
- If your renderer runs out of memory try creating more rendering tasks (more reducers) or try rendering a smaller subset of the US. Also be careful how many features you emit at higher zoom levels - you do not need every road at high zoom levels and trying to add every road to a renderer will fill up your heap space in a hurry.

Image Extractor Tool

The output of the render step will be stored on the DFS in SequenceFiles. We have provided a class that will extract individual tiles to PNG images on the local disk (the machine from which the job is being run) with appropriate file names.

Map Visualization Web Page (for testing)

Also included is a small html file (viewer.html) that can display a 3x3 grid of tiles, can scroll in all 4 directions, and can zoom in and out. This html page relies on the tile images being located in the "tiles/<z>" directory relative to it. (For example to display tiles at zoom level 6 make sure that the tiles are located in "tiles/6/" relative to the viewer.html file. This viewer also relies on tiles being named in format specified by the image extractor tool, as well as on the static images in the 'html_img' directory.

What To Turn In

Turn in all your code for your complete pipeline. This must be capable of start-to-finish parsing the data sets, joining the appropriate records, and rendering the tiles. It must also generate the indices that allow you to look up the latitude and longitude coordinates for addresses, cities, and states.

Add a text file containing your writeup.

Add the tile image files at zoom levels 4 to 8 for the area centered at the University of Washington (include a reasonable area, but not too large - 10 or so tiles per zoom level should be enough). When you zip up your submission, put these in "tiles/4/", "tiles/5/", ..., "tiles/8/"

You should run your full rendering pipeline on the cluster and leave your image files and geocode indices in your /user/*username* directory. You will need this data for assignment 4; you should not need to rerender it at that time.

Appendix: Data Sets

TIGER -- The 2006 complete US TIGER/LINE survey data.

2006 TIGER/Line data home:
http://www.census.gov/geo/www/tiger/tiger2006se/tgr2006se.html
Complete TIGER/Line Technical Reference Document: http://www.census.gov/geo/www/tiger/tiger2006se/TGR06SE.pdf
    In particular, see section 6 (Data Dictionary), which defines all the record types.
Tiger CFCC's (Census Feature Class Codes)
http://proximityone.com/tgrcfcc.htm

BGN -- This ties place names to locations.

USGS BGN Home:
http://geonames.usgs.gov/
BGN File Format spec:
http://geonames.usgs.gov/domestic/gaz_fileformat.htm
BGN Feature Classes:
http://geonames.usgs.gov/domestic/feature_class.htm

Census Population Data - This provides city and town populadtion information.

We use a pre-processed format of this data that has the state codes added and headers and footers cleaned, but you can see the raw data here:
http://www.census.gov/popest/cities/SUB-EST2007-4.html

The preceding material is from the University of Washington Computer Science & Engineering senior undergraduate course:

CSE 490H
Scalable Systems: Design, Implementation and Use of Large Scale Clusters
Autumn 2008

For further information (including all lecture material) see:

http://www.cs.washington.edu/education/courses/490h/08au/

Except as otherwise noted, all content is licensed under the **Creative Commons Attribution 2.5 License**.