Whatever the Weather
Basil Siddiqui and Kevin Schmid

With our project, we aim to utilize visualization techniques to promote explanation of weather data, and related data about the environment (for example, population size). We aim to design an intuitive visualization that will provoke users to explore trends in climate data, and investigate correlations, look at change over time, and be able to answer their own questions about weather trends. One of most immediate applications of this, we feel, is to examine claims made about global warming in greater details. Websites like NASA's offer, to some extent, visualizations of data for variables like "global surface temperature." But these visualizations have limited interactivity: while there is, for example, a world map with temperature changes, it doesn't support zooming, details-on-demand, or any of the useful features that make visualizations dynamic and engaging. As a result, the website gives a straightforward visualization that tells a one-sided story. We think that there's a lot more to the global warming story, and we hope that we can produce a visualization that is effective in letting users piece together the story themselves. Primarily, we will limit our region of interest to the United States and work to enable the following tasks:

- What portions (cities, states, regions) of the United States have experienced the largest changes (both increases and decreases!) in temperature (and any other factors that the API offers, such as dew point, humidity, air pressure, wind chill index, etc.)?
- How has the population of a city/state/region changed over time? Did the temperature change accelerate when the population growth in a region accelerated?
- Be able to compare this data (and more, on-demand) in a detailed way for two or more cities, on a line graph. Or more generally, a rectangular region of the United States that the user can select.

We will obtain the data for our visualization from the following sources:

- **The wunderground.com Weather API** offers historical weather data for cities. The data is available in both JSON and XML formats. Currently, we have registered for an API key, and we are able to obtain the weather data from this API. We did notice that there is an API limit of 500 calls per day; we will probably have to fetch some of this data in advance (instead of fetching it as the user requests it, which may not even be possible without doing something like this). This shouldn't be a problem, since one API call can give us the historical data for an entire state's worth of city temperature data for a particular day. So, we can choose some established days in the year (a summer date, winter date, etc.) and then accomplish this in about several hundred API calls using a Python script. Other than the API key, no authentication is required, so we don't expect this to be that complicated.
- **Various census data sets** to extract population information for cities. There appears to be population data available for the 1980-2010 censuses, right around the global warming hot spot[1]. Datasets appear to be unorganized and available in inconsistent formats. Hopefully, with Google Refine and Google Fusion Tables, it won't be difficult to grab

[1] No pun intended.

corresponding population data for each of the cities.

We would like our main visualization to convey data in a color-colored, interactive map. Let's stop there first: independent of the data we've collected, we need to have the framework set up for this map. Ideally, the map will be zoomable, have labels, and be reasonably accurate in terms of scale and depiction of landmarks. We have considered using the Google Maps API, which can embed a full screen map on a page. The map can be represented as a JavaScript object, which has various methods that allow you to customize the appearance and behavior of the map, specifically with event handlers. However, there are some downsides to this: namely, the customization that can be done is limited by a high-level API, which could restrict the possible visualization techniques we could employ. (That said, the API's markers, events, and polygons aren't bad!) To make this more in-line with a project for this class, we will probably use D3 and its lightweight abstractions over DOM manipulations instead to create a map, which won't be difficult after completing HW6, according to feedback from our project TF.

Once we have the map, we need to decide on how to initially present the data to the user, and then further decide what kind of interactive components to add. An earlier homework assignment included readings that advised against the use of a rainbow color map, so we'll stay away from that, but we will probably want to use some sort of color scale to indicate temperature, and color the map in accurately based on this scale. When you load the visualization, the information displayed could pertain to the most recent decade (2010 temperature data), but there could be a slider on the bottom of the map (similar to the NASA visualization referenced earlier) that allows you to select a specific time period. Moving this slider back and forth will presumably change the colors shown on the map and enable the viewer to get a good sense of the change over time for a certain place (or just the entire US in general) on the map. The behavior would be the same when one zooms in on a particular spot: you would just see the color change for that particular region.

One possible interactive component of this map would be the ability to select a specific city. When you select a specific city, a modal window would appear, displaying specific information for that city (such as the population data, climate data, etc.), as well as any other visualizations we may want to display on a per-city basis. This could be a simple line graph, showing the trends in the data mentioned earlier. We feel that this series of visualizations would promote a very natural and intuitive way to explore the data: yes, there's a lot of data available, but not all of it is shown at once, it's not too "busy," and it doesn't stray too far away from familiar visualizations. At the same time, we intend to add "features" to our visualization to make it more engaging, but keep it not overwhelming for viewers. One of these features is a tool that allows the viewer to select a rectangular region from the map, and then view specific details about the cities contained in this region, including the averages of all the statistics from the API, as well as the trends for the averages over time. At the same time, the user may be interested in customizing exactly what kind of data is displayed. Currently, that functionality is limited; the map merely provides a slider for selecting a particular target year. We can extend this to allow for removal of outliers and to limit the data shown for cities with a value for a certain data type in between a certain range, etc. Form elements (especially with a library like JQuery UI, or Bootstrap) would make this easy to implement on the interface side, and we could just add a layer of abstraction in our JavaScript functions that makes the rendering functions accept range

information.

Table of Contents

Whatever the Weather

search for a location... | GO! | mm/dd/yy to mm/dd/yy

US City	Weather °F	Date
San Francisco, CA	66	5/11/57
Sacramento, CA	75	5/11/57
Dana Point, CA	70	5/11/57
Berkeley, CA	64	5/11/57
Davis, CA	72	5/11/57
San Mateo, CA	72	5/11/57
Santa Monica, CA	74	5/11/57
Roseville, CA	80	5/11/57
Marin, CA	83	5/11/57
...	...	...

COMPARE MODAL

Compare

by cities..

1 Enter a city... 2 Enter a city

by region..

Year Range

1910 to 2000

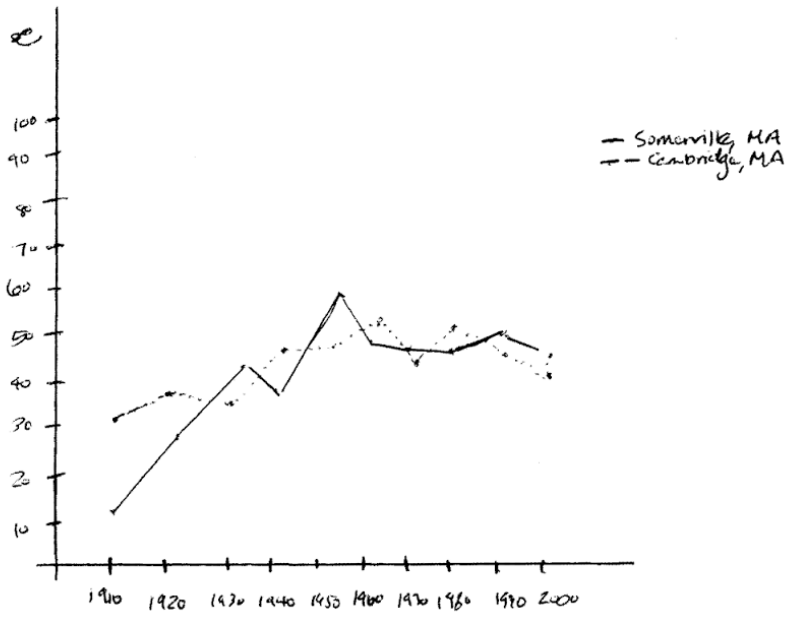
Adjust

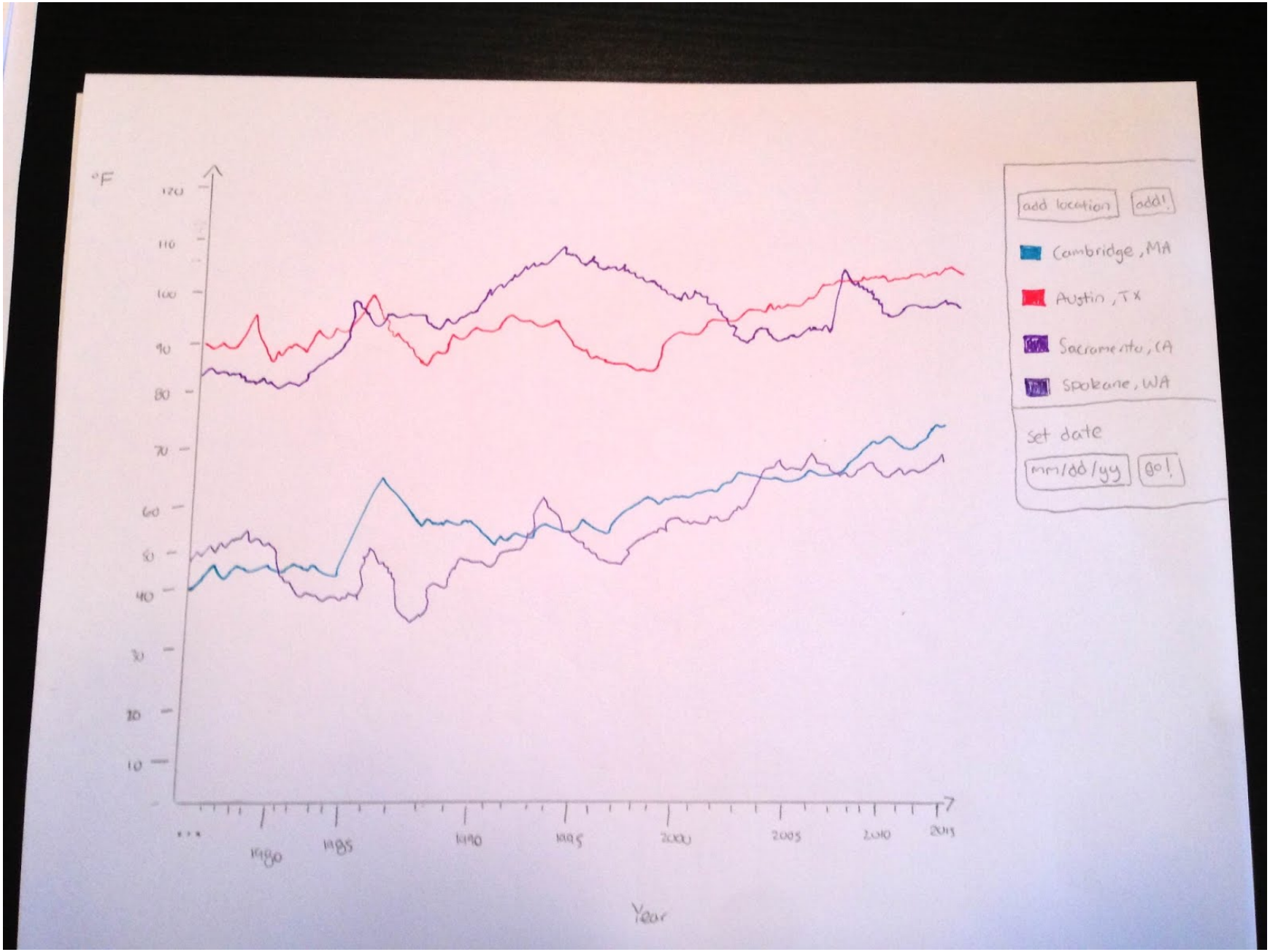
Views

- (•) Temperature
- ( ) Humidity
- ( ) Population
- ( ) Wind Chill

Season

- (•) Spring
- ( ) Summer
- ( ) Fall
- ( ) Winter

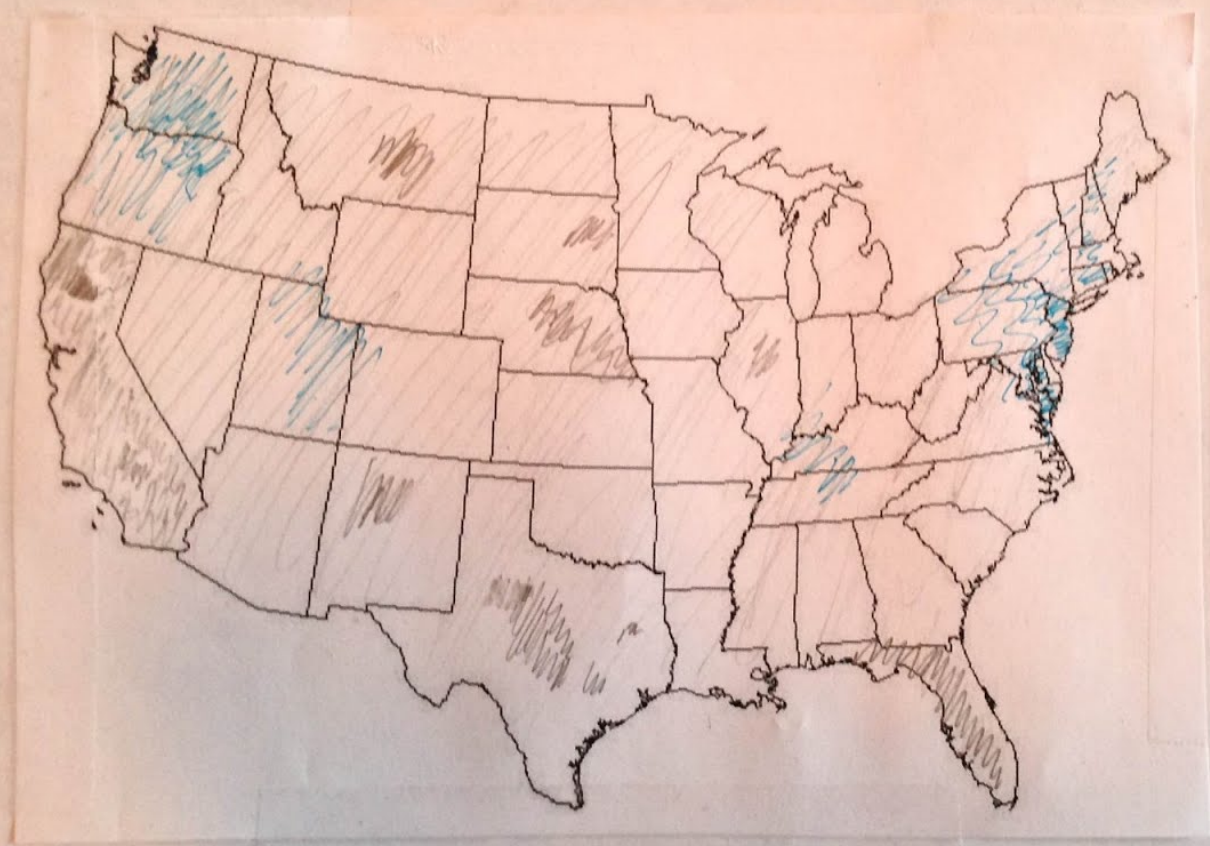



°F
120
110
100
90
80
70
60
50
40
30
20
10
1980
1985
1990
1995
2000
2005
2010
2015
Year
add location
add!
Cambridge, MA
Austin, TX
Sacramento, CA
Spokane, WA
set date
mm/dd/yy
go!

Whatever the Weather

search a specific area or event . . .

GO!



Control Panel

animate visualization

1957

slide through the years

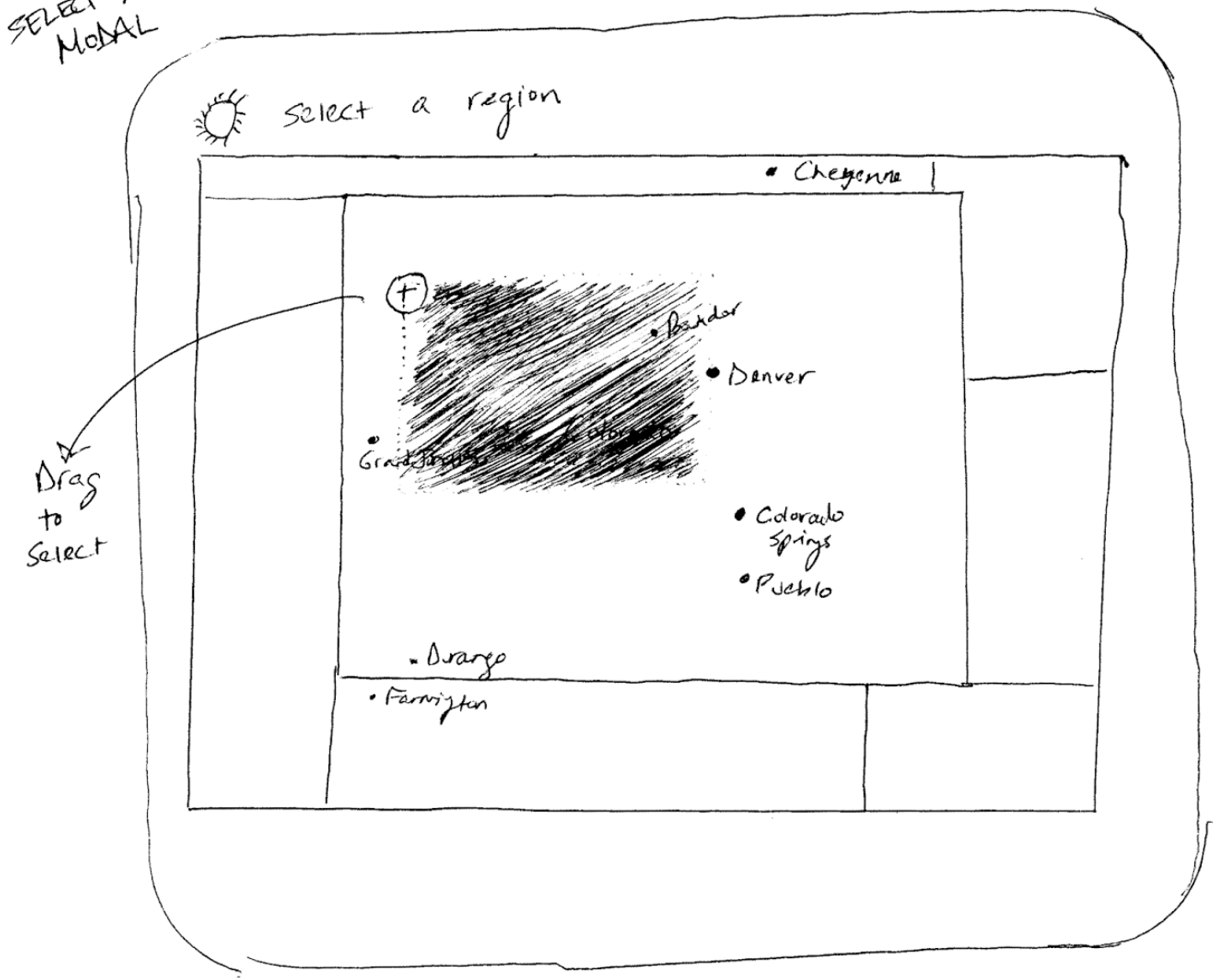
mm/dd/yy to mm/dd/yy

display specific dates

weather index

10 30 50 70 90 110

SELECT A REGION MODAL



Tasks:

1. Develop a JSON structure for representing weather data. It should be easy to filter out weather data for a particular year, and then by city, so something like this

   ```
   {"2000": {"San Francisco": {/* data */}}}
   ```

   would be reasonable.

2. Get some data.
   a. Ideally one or two cities at first, with a bunch of dates for each, to take full advantage of the JSON structure
3. Implement a "template" CSS/HTML page for the visualization.
   a. Whoever's doing this may want to use Bootstrap for the modal views.
   b. Need not be interactive or include maps - just the basic structure.
4. Implement basic components of the visualization
   a. Start with DataMaps, and the weather for one or two cities.
   b. Once DataMaps is fully functional, try Google Maps. (This could almost be done independently of 4a)
5. Add the key deliverable interactive features:
   a. Brushing
   b. Linking
   c. Cross filter
6. (In the mean time) Get more data
   a. It should be in the format decided by the person who does #1.
7. (Always) Document things in the process book

A JavaScript Interface for Retrieving Map Data

```
function filterByYear(year)
{
      // filterByYear allows for the retrieval of all weather data for
         a particular year.

      // It returns an array, where each element corresponds to a
         specific city.  Each element is structured in the following
         way:

         {
            "name": "San Francisco",
            "state": "CA",
```

```
            "data": {
                  // temperature data here
            }
          }

      // If the year is invalid, it throws an exception.
      // If the year is valid, but no data is located, it returns
          the empty array.

}

function filterByCity(city, state)
{
      // filterByCity allows for the retrieval of all historical
         weather data for a given city.

      // It returns an array, where each element is structured in the
         following way:

         {
            "year": 2000,
            "data": {
                  // temperature data here
            }
         }

      // If no city is given and the function is called, an
         exception is thrown.

      // If a city is specified but not found, an empty array is
         returned.

      // Same behavior with the state argument.
}
```

Scraping

Kevin

Today, I started to scrape data for usage in our visualization. Early into the process, I realized something: the Wunderground API we decided to use had a rate limit that was really going to interfere with our scraping. We were limited to making 500 API calls per day. On a single API

call, the most information we could get was the weather for a single city on a particular day. The API did not offer the averages, either. That's a lot of API calls that would be necessary!

I contacted Basil, and we began exploring other options. We considered sites like:

- http://weather-warehouse.com/WeatherHistoryListing/monthlyWeatherDataStart.html
- http://www.nws.noaa.gov/pa/climate_data.php
- The list of sites at http://droughtmonitor.unl.edu/source.html

But none of them were very organized and cost-effective.

However, then we had an idea! We knew one place that offered this weather data - namely, Weather Underground. So maybe there was another way to access this data. Sure enough, there was! For a sample page, check out http://www.wunderground.com/history/airport/KBOS/2013/3/27/DailyHistory.html?req_city=Cambridge&req_state=MA&req_statename=Massachusetts.

So we have now modified our scraping data source - instead of using the WUnderground API, we will use the WUnderground website, and use Python's Pattern library to do the scraper.

Regardless of how we carried out the scraping, I wanted to determine a reliable and consistent JSON representation of the scraped data, so that I could write the helper JavaScript functions that operate on the dataset independently (and test them on smaller datasets, etc.)

Here's the format that I came up with:

```
{
      "2010": [
            {
                  "city": "Cambridge",
                  "state": "MA",
                  "data": {
                        "temperature": "23.4"
                        /* other key-value pairs can go here */
                  }
            },
            {
                  "city": "North Babylon",
                  "state": "NY",
                  "data": ...
            }
      ]
}
```

There are a couple of things I would like to note about this specific representation of the weather data we scraped. First, that it is, at the top level, a JavaScript object whose keys are the years. (The corresponding values are arrays.) This was done deliberately, to make it easy to get all of the weather data for a given year. I specifically chose this organization (rather than organizing by city, then year) because I think that a common usage of our visualization is going to be to view the changes in weather data over time for a given map view.

As a negative, though, now, it makes gathering all of the data for a particular city somewhat inefficient, in that we have to loop through each of the years, and then perform linear search for the city. If this proves to be too slow in practice, I am thinking of some possible workarounds:

- Store a pre-processed copy of the data that was scraped, organized by city.
- During the initialization of the visualization, create a new version of the dataset that is organized by city.

Also, there is something of an asymmetry in the way we note the year (as a key of the object), and the way we note the city/state (a value). This was intentional, to avoid ambiguity in using the city name alone (i.e., it's possible for two states to have identically named cities), and also to avoid having the city and state combined. (We wanted to be able to access both separately.)

Scraping went pretty well overall. I started with the HW1 example to refresh myself on the usage of the Pattern library, and then wrote a scraper that could get one field from one specific website, just to make sure I was accessing the right DOM element on the page. Then, I factored that out into a function, and added more looping logic, so that it would proceed through a set of web pages!

To make the scraper more configurable, I have created a csv file (currently called `cities`) that holds a list of cities to scrape.

JavaScript Functionality for Accessing Data

Kevin

I worked on implementing the interface proposed earlier for retrieving map data. Given the way we structured the JSON data, this was pretty straightforward, and most of the implementations were done by accessing array elements in a loop, and comparing strings in the search part.

We also implemented a function called `getCities`, which returns an array of all of the cities in the dataset. This was useful later on for some tasks in the JavaScript code (specifically, implementing the `typeahead` functionality on a search box), so I added it to the interface.

```
function getCities() {
```

```
      // returns list of cities
}
```

Also, I required that clients of this API call `initWithData`, passing in the JSON data, before using the functions.

Google Maps API

Kevin

Now that I had some preliminary data, it was time to build some of the initial features of the visualization. I emailed our TF, Azalea, to ask about whether or not we could use the Google Maps API for one component of the visualization. This turned out to be a very helpful conversation about the various options for the map visuals (thanks!)

For now, we decided to start with Google Maps API for the homepage's visualization, and then use D3 to generate more complex visualizations for the individual city and "selection of cities" views.

The Google Maps API is very well documented. I started by looking at the simplest map example available in the documentation, to learn how to instantiate the map. Then, I worked with a simple example of a Bootstrap CSS layout (available in Bootstrap's documentation) and merged the two to create an initial page:

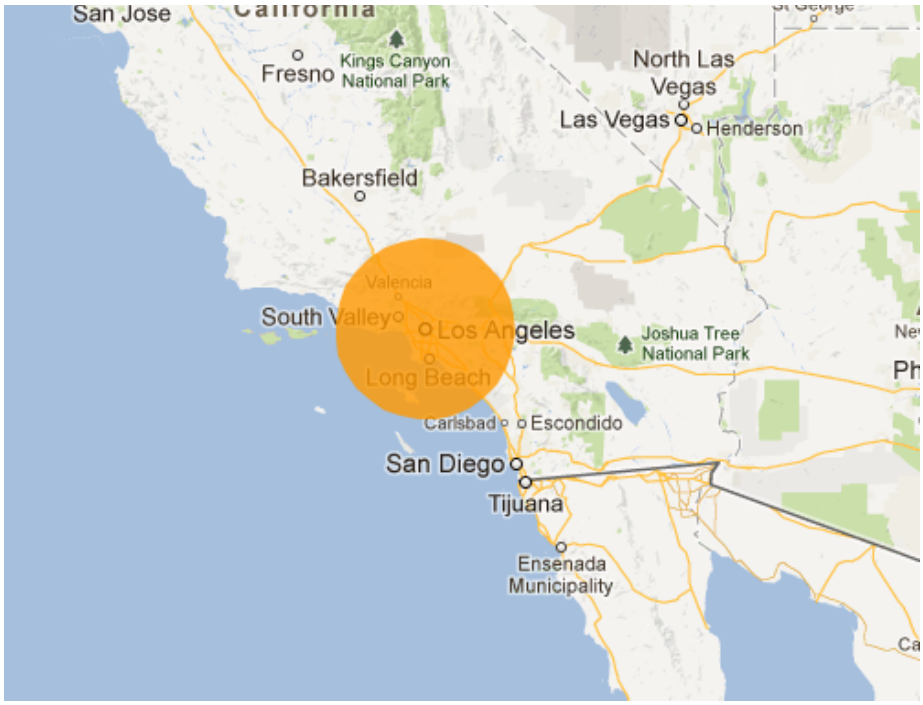
Looking at the documentation for Google Maps API's `MapOptions`, I learned how to configure the initial zoom level, center, type of map, and available of controls. Here are some choices we made with regard to this:

- The `center` is set to coordinates in the middle of the United States, since that's where all our data is from.
- `mapTypeId` is set to `google.maps.MapTypeId.ROADMAP`. We chose not to use `google.maps.MapTypeId.SATELLITE`, because we felt that with `.ROADMAP`, the range of colors that the map imagery would be limited, and we could make more informed color choices. Also, the `ROADMAP` type seems brighter and looks better.
- We decided to disable the `panControl`, the `streetViewControl`, the `overviewMapControl`, and the `mapTypeControl`. We felt that all of these were features that provide nice functionality, but are unnecessary and take the focus off of the visualization itself.

Now, it was time to add some data to the visualization. In doing this, I decided to explicitly avoid using the default markers that come with the Google Maps API. These don't really offer a great way to visually encode the notion of a "data point" and a lot of them at once can appear overwhelming. Even with custom icons. Instead, we noticed that Datamaps (one of the map

visualization technologies we were considering using and the one that HW 6 used) generates nice, clean, well-encoded visualizations using “bubbles.” So our objective was to replicate this concept in Google Maps. For that, the `Circles` class (and the corresponding `CircleOptions` class) came in handy.

Duplicating this concept was pretty straightforward - we wrote a `chooseColor` function that chooses a color for the circle based on a data value, and customized the circles so that they are somewhat transparent.



It’s unclear whether we would want to change this particular encoding (maybe add a border, or decrease the radius!) but we think that this is a good start.

Adding Controls to the Layout

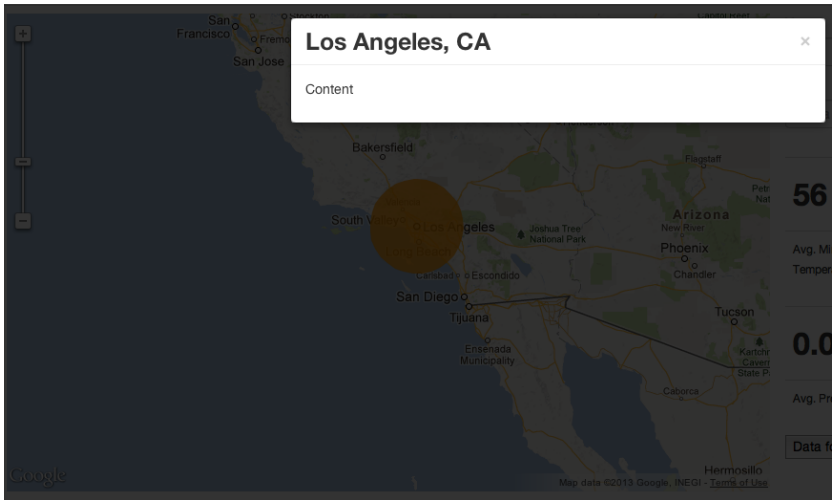
Kevin

Out of the box, the Google Maps API gave us an interactive, dynamic map that supports panning. We were also able to successfully add circles to the map to represent our particular data points. However, it’s hard to classify the visualization itself as “interactive” at this point - there’s no easy way to view data in more detail and filter. That’s what my next goal was in working with the visualization.

Our original vision of the visualization was such that, when a user clicked on one of the circles, a modal with more details about the particular city selected would appear. Inside this modal would

be a comprehensive set of visualization for that particular city's historical data.

Implementing this feature was pretty straightforward, thanks to `google.maps.event.addListener` and Bootstrap's modal functionality.



Above is the modal view: when you click on the orange click, the modal appears, shading everything and bringing the window into focus. In our final visualization, we intend to make the modal much larger and include actual visualizations!

Another interactive layer to our visualization was that our users would be able to select the year for which they wanted to view the weather data. In our original sketches, we represented this feature with a slider on the bottom on the map. For now, though, we're choosing to implement this feature with a text box, that can allow the user to just type in the new year value. This is because we'd like to get the visualization in a working state before we begin to use a fancy slider or something like that; the textbox is easy to setup and pretty well documented.

So we added a textbox. When the textbox is changed, an event listener gets called, which updates the map to have the appropriate data for the given year (using `filterByYear`). If no data is available, no data is shown.

Similarly, we wanted users to be able to find a particular city on the map. Using the `getCities` function described earlier, we are able to obtain a list of the dataset's cities, which we used to fill a Bootstrap typeahead box with the list of the cities, so that they appear when the user types them. The typeahead documentation permits us to specify some callbacks (one in particular is called `updater` - it gets called when the user selects something), so we used the `updater` callback to adjust the zoom of the map.

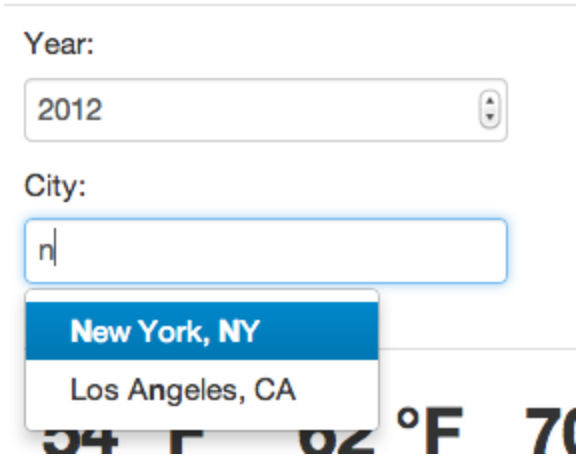
This raises the question of "how does the user zoom back out?" since we disabled the zoom controls from the map. Here is what we have decided to do as an alternative:

- When the zoom level is changed, and it is above a certain threshold, a button will appear (an event listener is responsible for doing this) that allows the user to restore the original

zoom mode, which is the United States.

- Outside of that threshold, no button will be shown.
- The user is still allowed to zoom in on the map by double clicking. We think that this is okay, since the button will appear in this case, too.

Notice the "Full US" button that appears when zoomed in! Currently, we've positioned it below the map.

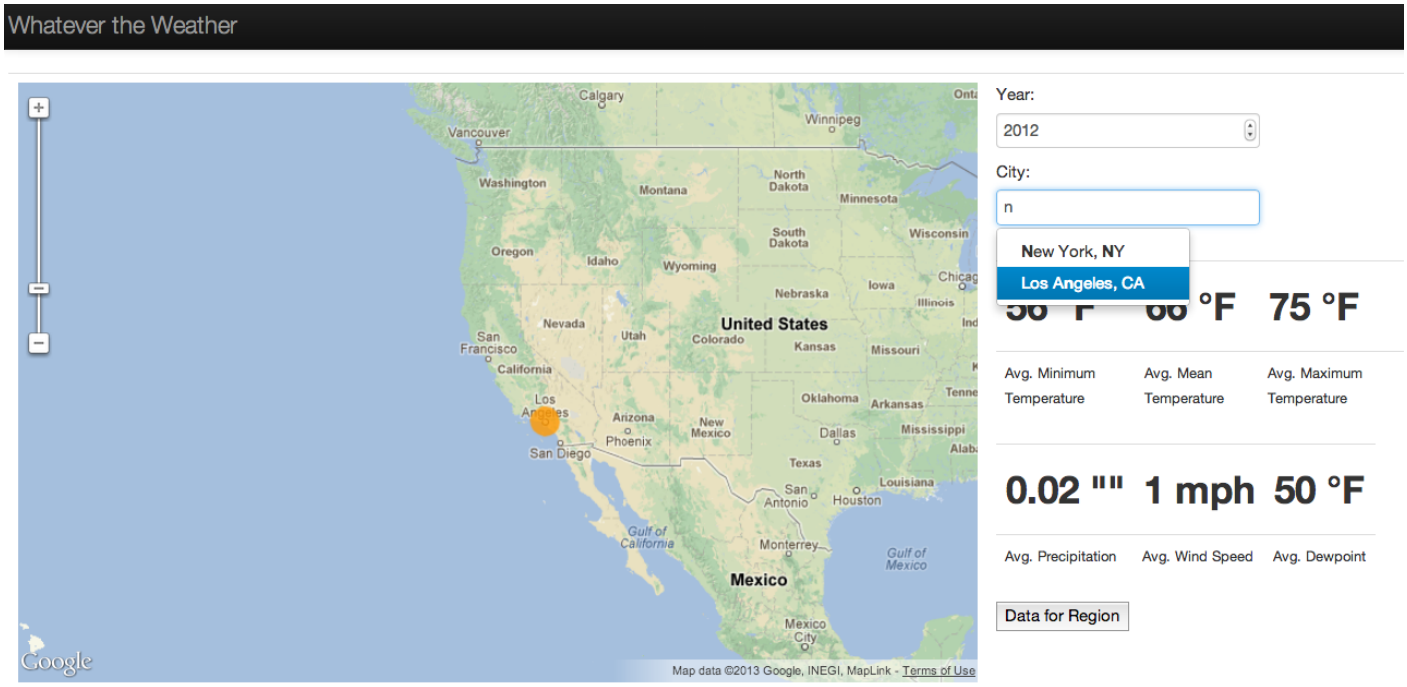


The map control pane, positioned to the right of the map itself.

Before we began to work with `d3` to generate per-city (or per-city-selection visualizations), we wanted to try out one more thing without `d3`. We were interested in showing some statistics about the user's current selection (average temperature of cities in the selection, say) on the sidebar with the control pane, that would be updated when the user panned around the map. We thought this would be neat, non-intrusive, and just a "cool" component of the visualization. We also felt that it would be a good starting point for implementing the per-city-selection visualizations, since those rely on being able to determine what is in the current map view.

Tools used to implement this:

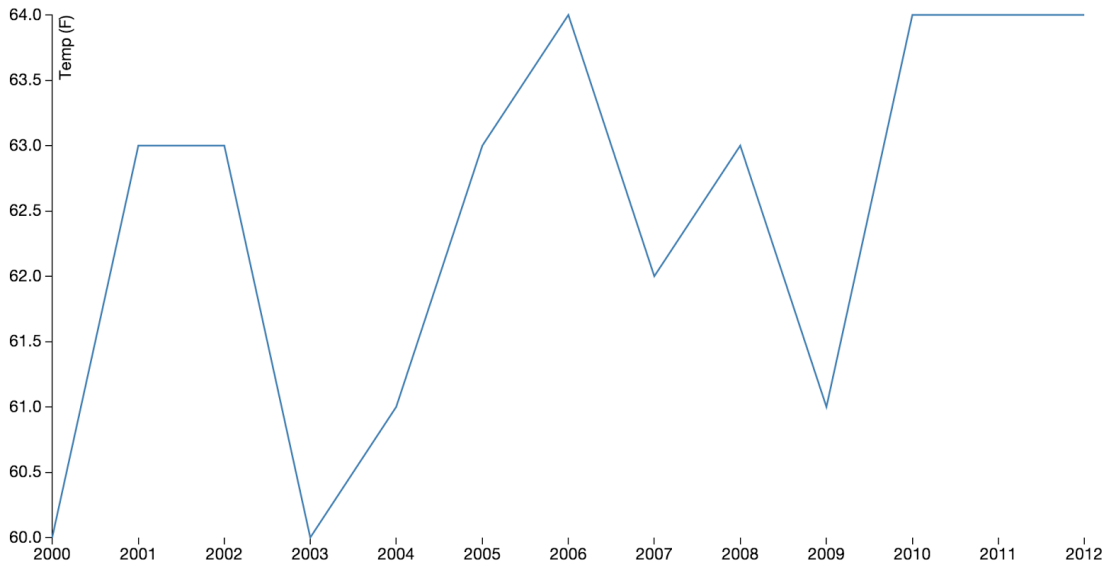
- Our `filterByYear` function, to grab the selection of cities for the current year.
- The `getBounds` method on the Google Maps Map object.
- Event listeners.
- DOM updating (to push the newly computed statistics to the DOM).



The full map view. You can see a video of our map view (with panning in action) at http://www.youtube.com/watch?v=Cyrk7pdGNBk

Implementing a Line Graph with D3

The most intuitive visual encoding to display weather trends was a line chart. I decided to take a simple line chart example and modify it for our purposes. During the testing period I hard coded values and created a separate view.



I discussed possible implementation methods with Kevin and we decided to integrate the graph in a modal for each city. I refactored the code for the line graph and created our graphs.js file, a file for all our graphing functions.

```
function lineChart(city, state, datapoint){
      // draws D3 line chart for the given city and datapoint
}
```

Adding NVD3

While I was learning about D3 and working through some tutorials, I found a great graphing library built on D3 called NVD3. According to the NVD3 website's description:

> "This project is an attempt to build re-usable charts and chart components for d3.js without taking away the power that d3.js gives you. This is a very young collection of components, with the goal of keeping these components very customizeable, staying away from your standard cookie cutter solutions."

After looking through some of examples on the NVD3 website, I decided to add the library to our project for use with future encodings. All of the examples were documented well, had great interactivity, and were fully customizable.

Getting Data Ready for NVD3

To use the built-in graphs provided by NVD3, we had to reformat our JSON in NVD3's preferred structure:

```
[{
  "key" : "Cambridge",
  "values" : [
  ["2000", 32],
  ["2001", 37],
  ["2002", 29]
  ]
},
{
  "key" : "New York",
  "values" : [
  ["2000", 35],
  ["2001", 40],
  ["2002", 39]
  ]
}]
```

I wrote a simple function in graph.js for all our NVD3 graphing functions to use to get the proper JSON format:

```
function compareCitiesData(city1, state1, city2, state2, datapoint){
     // returns JSON of reformatted relevant data for two cities
}
```

Implementing a Stacked Area Chart / Compare Line Chart

Once I got the data in the proper format, I began testing the various graphs provided in the NVD3 library:

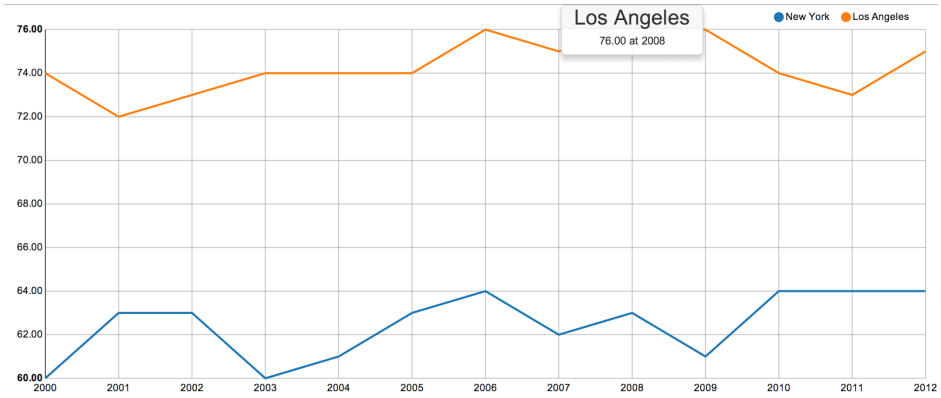
We met to discuss the graphs we would like to include in our comparison tool and ultimately decided to go with a Stacked Area Chart and Multiple Line Graph.

From there I wrote two functions in graph.js to simplify the creation of these graphs:

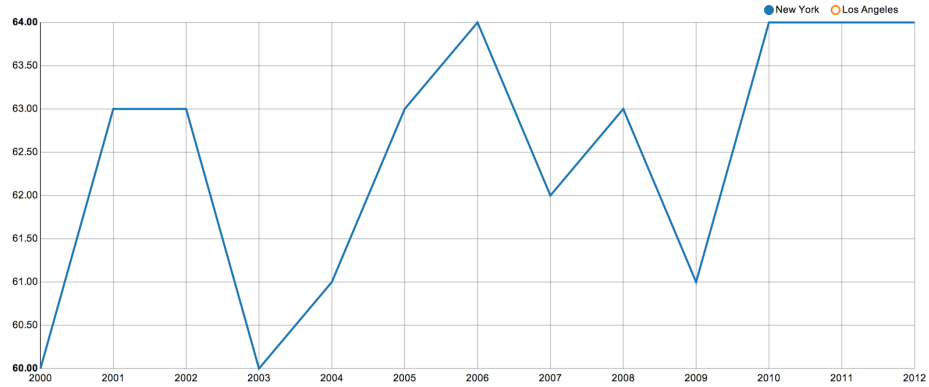
1) compareLineChart

```
function compareLineChart(city1, state1, city2, state2, datapoint){
      // draws a D3 a line chart of the two cities with interactivity
}
```

When you mouse over a given data point, a pop-up is displayed with information about the datapoint:



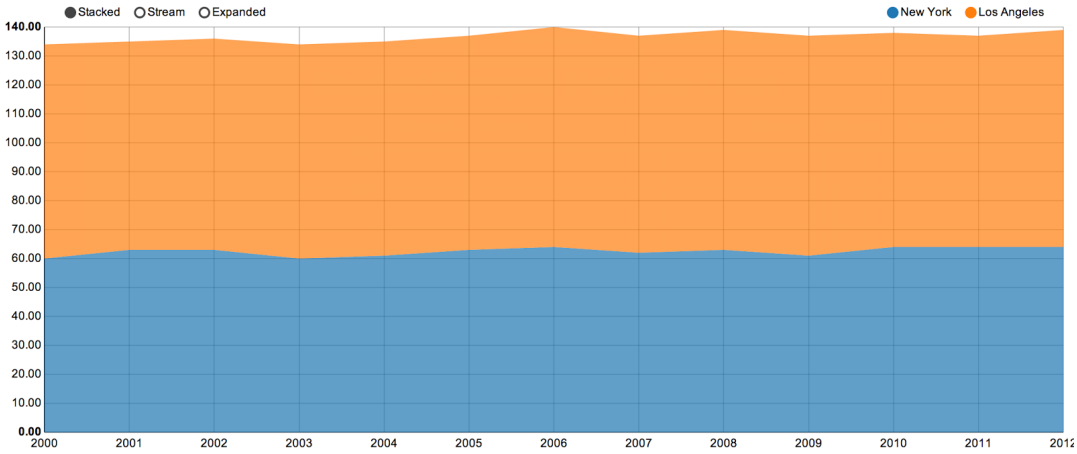
You can also choose to toggle which cities are displayed:



2) compareStackedAreaChart

```
function compareStackedAreaChart(city1, state1, city2, state2, datapoint){
      // draws a D3 stacked area chart of the two cities with interactivity
}
```

The Stacked Area Chart has the same mouseover and toggling features as the Multiple Line Chart:



Reflections on Choice of Visual Encodings

To summarize, our visualization utilizes a variety of visual encodings to represent data:

- In the overview map view, we use circles to show the average temperature of a city in a given year. Given the nature of the the data, which is quantitative in form, we felt that this would be an appropriate encoding. The color scale used is very coarse grained - this is

also intentional, to allow the viewer to quickly discern which territories and cities are “cold” and which are “warm” without being overwhelmed by a myriad of colors. It also promotes, we feel, exploration on behalf of the user: if they can easily determine the temperature range of a particular city (or even region), they can further explore these insights with the more fine grained tools and the line graphs.

- In the detailed view, we use a single series line graph to show the trends in weather data (average temperature, average maximum temperature