Dynamic Programming: Bitonic Paths

C343 — Assignment 6

Due: February 24th, 2015 at 11pm

1 Preliminaries

- Please form teams of 2-3 for this assignment.
- If you are using `silo`, make sure you are using the right version of Python. Type:

  ```
  > module load python/2.7
  ```

  to make sure you have the right version.
- Download the following three files from Oncourse:
  - `bitonic.py`. This is the file you will be working with; you are given a few helpful definitions to get you started.
  - `gmapsrappa.py` and `DrawBitonic.py`. These two files together define a function `createMap` which you can use to overlay your paths on Google maps. It is a great way to visualize (and debug) your solutions. You do need to touch these files at all.

2 Points

The first step is to create a Python class representing points on Earth. Each point will be specified by its latitude and longitude in degrees. You can use `http://www.latlong.net` to find the coordinates of your favorite locations. For example, we can define:

```
indianapolis = GeoPoint('Indianapolis',39.768403, -86.158068)
bloomington  = GeoPoint('Bloomington' ,39.165325, -86.526386)
chicago      = GeoPoint('Chicago'     ,41.878114, -87.629798)
louisville   = GeoPoint('Louisville'  ,38.252665, -85.758456)
cincinnati   = GeoPoint('Cincinnati'  ,39.103118, -84.512020)
stlouis      = GeoPoint('St. Louis'   ,38.627003, -90.199404)
columbus     = GeoPoint('Columbus'    ,39.961176, -82.998794)
```

We will only need one method on these points that calculates the distance from one point to another. There are several formulae to calculate distances on Earth. In my calculation, I produced the following distances from Bloomington:

```
>>> round(bloomington.distance_to(indianapolis))
46.0
>>> round(bloomington.distance_to(bloomington))
0.0
>>> round(bloomington.distance_to(chicago))
```

```
196.0
>>> round(bloomington.distance_to(louisville))
75.0
>>> round(bloomington.distance_to(cincinnati))
108.0
>>> round(bloomington.distance_to(stlouis))
201.0
>>> round(bloomington.distance_to(columbus))
196.0
```

It is acceptable if you get slightly different results. Note that these distances are as "the crow flies"; driving distances are obviously longer.

The file `bitonic.py` also includes a useful utility for generating a few random points near a location:

```
def generate_points (p,n) :
    """Generate 'n' points near point 'p' (approximately +/- 5 degrees)."""
    res = []
    for i in range(n) :
        dlat = (random() - 0.5) * 10
        dlon = (random() - 0.5) * 10
        res.append(GeoPoint('p'+str(i),p.lat+dlat,p.lon+dlon))
    return res
```

The given function will generate `n` points near the given point `p`. The notion of "near" used above is to be within ± 5 degrees in both latitude and longitude. Feel free to modify this at your convenience.

3 Paths

Our next step is to create a generic class for representing *paths*. A path is simply a sequence of points. You will need to implement the following methods that are needed for implementing the shortest paths calculation:

- `extend`: adds a new point to the current path
- `reverse`: reverses the current path
- `clone`: makes a fresh copy of the sequence of the points and returns it

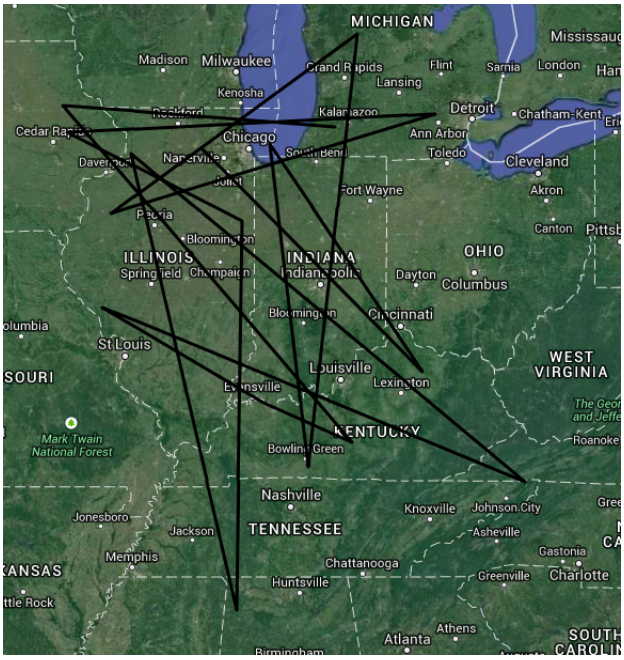
The file `bitonic.py` includes a fairly comprehensive set of test cases for these methods.

Once you finish implementing the class `Path`, you can start visualizing them on Google maps. I have provided two utilities for doing that. The first `draw_cities_path` is given below:

```
def draw_cities_path (fileName) :
    cities_path = Path(indianapolis)
    cities_path.extend(bloomington)
    cities_path.extend(chicago)
    cities_path.extend(louisville)
    cities_path.extend(cincinnati)
    cities_path.extend(stlouis)
    cities_path.extend(columbus)
    createMap(indianapolis,cities_path,fileName)
```

Calling this function with a filename (use an `html` extension) will create a file you can display using your favorite browser. Here is what you should see:

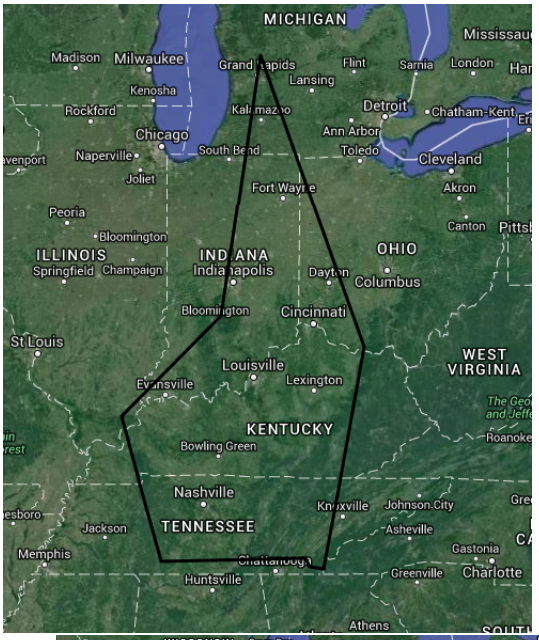
If you re-order the code in the method above, you should see a different path. I also provide another function `generate_path` that can produce a path connecting `n` random points. Here is an example of 17 points near Bloomington:

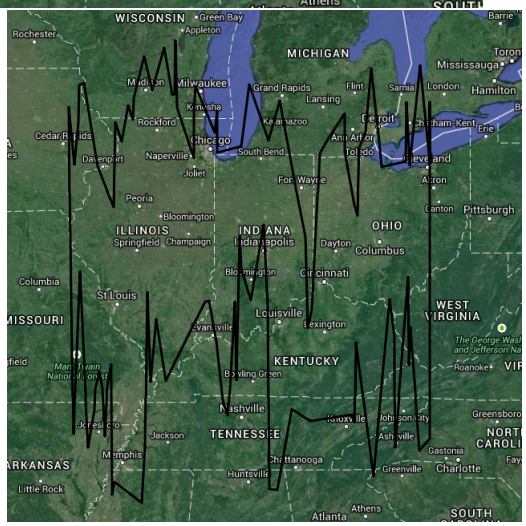


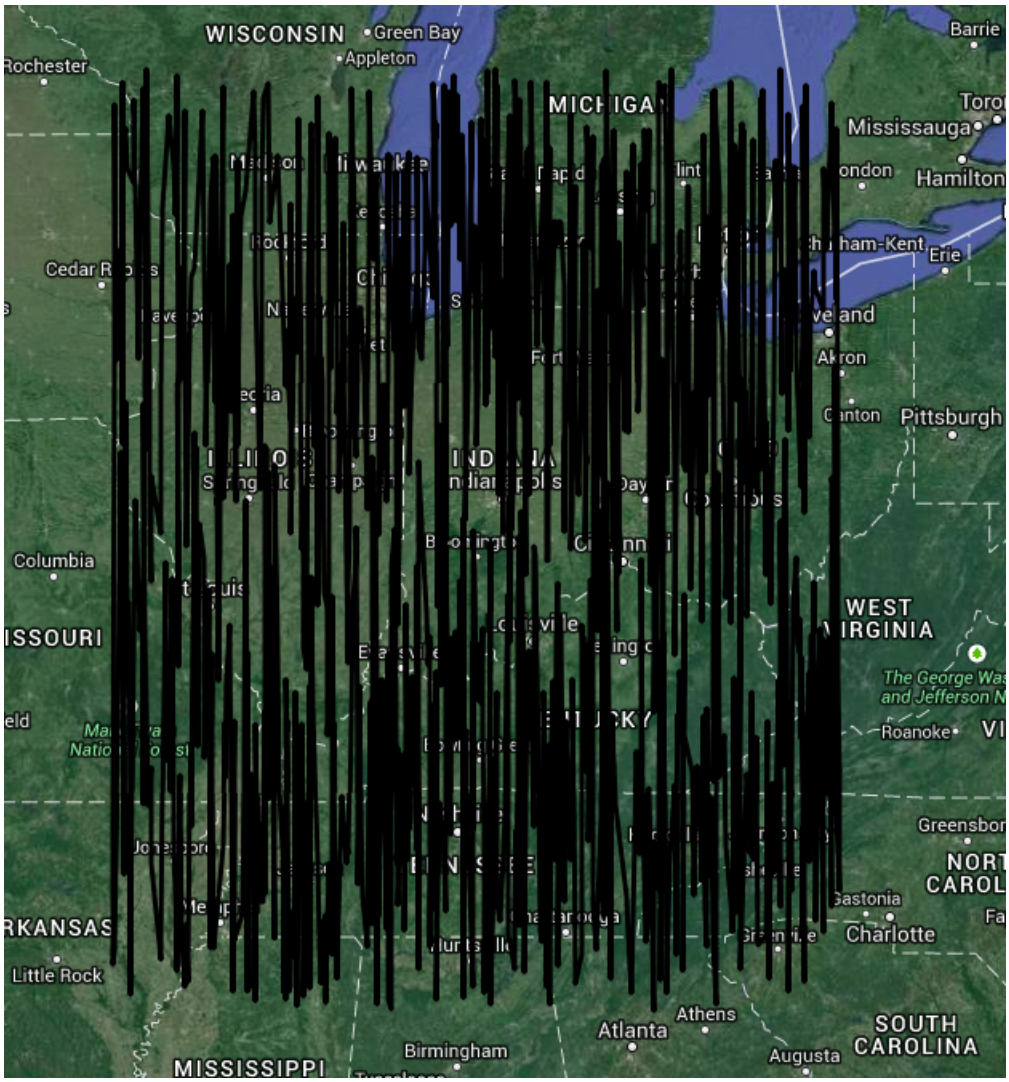
4 Shortest Paths

The main part of the assignment is to take a collection of points and compute the shortest *bitonic tour* that goes through all the points. A bitonic tour starts at one end — say the western end, goes exclusively east until the easternmost point, and then goes back exclusively westwards to the original point. For the 7 cities defined above, the shortest bitonic tour is displayed below:

To get a feeling for how this idea scales, here are few more bitonic tours of random collections of points of sizes 5, 8, 20, 99, and 999 respectively:







The remainder of this note gives several important hints to help you calculate the shortest bitonic tour for a given set of points $p_0, p_1, \ldots, p_{n-1}$:

- The general strategy you must follow is to develop a naïve recursive solution that does not attempt to avoid recomputing subproblems over and over, and then use a hash table to created a memoized efficient version.
- The first step to sort the points from west to east. After sorting, the point at index 0 should be the westernmost point and the point at index `n-1` should be the easternmost point.
- You will need a helper method in the class `Bitonic`. I called it `path`. This helper method takes two indices `i` and `j` with the guarantee that `i < j`. This guarantee will be an invariant of our construction: we will never make a call to the method `path` unless we guarantee that `i < j`. In other words, the point at index `j` is always guaranteed to be east of the point at index `i`. The method `path` will construct the shortest path that starts at the point with index `i`, then goes westwards until it reaches the point at index 0, and then goes eastwards until it reaches the point at index `j`.
- Given the helper method, the definition of the method `tour` is almost trivial. First one uses the helper method to compute the path starting at index `n-2`, going to index 0, and ending at index `n-1`. The

tour simply closes this path by connecting its end point with the starting point.

- The most difficult part of this assignment is to write the method `path`. Other than the base case (which you need to figure out), there are two interesting situations:
  - `i < j-1`. Visually this means that the path we want to construct starts "way" west of where it ends or symmetrically that the end point is "way" east of the starting point. The smallest allowed value for `j` is `1` and hence the point at index `j` which is the last point on the path must lie on the eastern part of the path. Now let us consider the point at index `j-1`. Is this point on the shortest path? We argue that it must. Consider the eastern portion of the shortest path starting at 0 and going east towards `j`. Let the next-to-last point on that path to have index `k < j-1`. This means that the shortest path goes from 0 to index `k` and then to index `j` skipping the point at index `j-1`. That skipped point at index `j-1` must however be part of the entire tour. Once the eastern path skipped the point, it is impossible to turn around to include it. It is also impossible to include the point on the western end of the path since index `i` is already west of index `j-1` and cannot also turn around to include the point at index `j-1`. We conclude that the point at index `j-1` must be the next-to-last point on the desired path. Since `i < j-1` we can simply make a recursive call to construct the shortest path starting at `i` and ending at `j-1` and append the point at `j` to it to construct the desired path.
  - `i == j-1`. In this case, we want to construct a path that starts immediately west of where it ends. Reasoning as above, we want to pinpoint the location of the next-to-last point on the eastern portion of the path. Previously we used the fact that `i` was way to the west of `j-1` to argue that the next-to-last point must be the one at index `j-1`. That argument does not work when `i` is immediately to the west of `j-1` so we must consider that the next-to-last point on the eastern portion of the path is at some index `k < j-1` or equivalently `k < i`. The question is which index `k` will minimize the total length of the path. There is no a priori reason to select any particular index `k` so we calculate the distances for every possible `k` and choose the one that produces the minimum overall path length. Note that the only way to invoke our recursive call is to pass arguments to `path` such that the first argument is strictly less than the second. In other words, the recursive call will construct a path starting from `k` and ending at `i` which is the `reverse` of what we want. Note that the length of a path does not change when it is reversed!
- The method `clone` in the class `Path` plays an important role in the Python implementation and depending on the details of your implementation will be critical to getting the code to interact correct with the hashtable. The issue is that the memoization introduced by the use of a hashtable, by definition, shares results instead of recomputing them. The results in question here are paths of course. This is wonderful for performance but is a disaster if one computation updated its path (for example by reversing it) as that reversing will interfere will all shared computations.

5 Deliverables

Each group should submit one instance of `bitonic.py`.