Compressing Chronological Geographic Data

Introduction:

One November day, I visited geacron.com. I'm a fan of maps and history, so I was intrigued by the concept, only to be disappointed by the implementation. Upon clicking around, I was annoyed by how long it took to navigate from year to year, as borders reloaded, despite only changing slightly, if at all. I could not abide the redundancy of transmitting the vertex data of Russia's borders over and over, simply to account for tiny variations in the borders of some country in the Balkans. As a student of data structures and algorithms, I vowed to improve, to do better than those who had come before, and to develop an application which would provide an example of how this could be accomplished more efficiently.

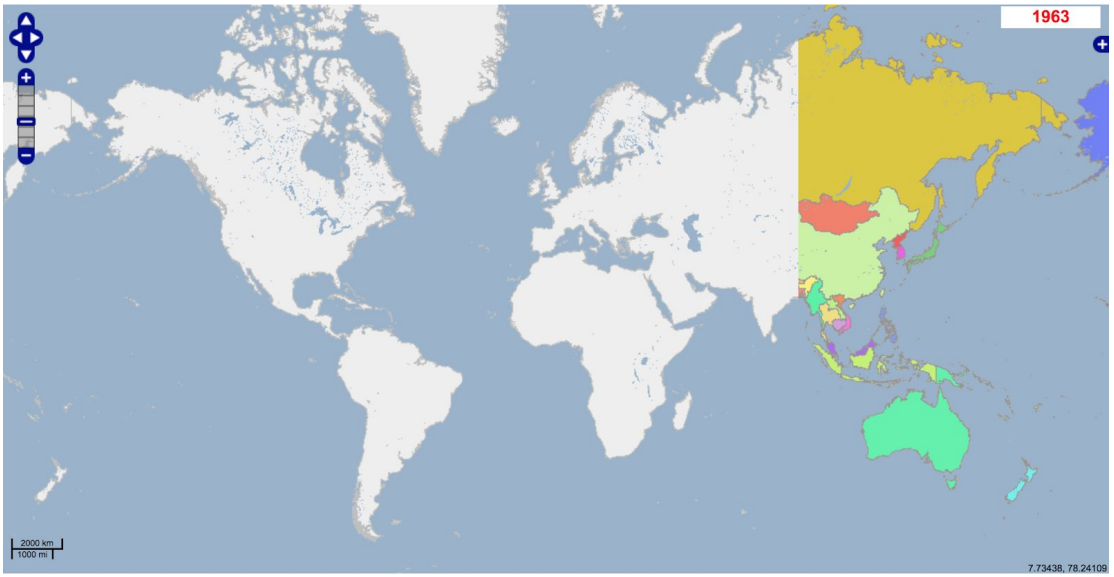

Figure 1. Geochron, as it struggles to load, presumably due to the vast amount of redundant data being transmitted - an undesirable and expensive (computationally and fiscally) user experience. Despite the low precision at this zoom level, there remains significant lag.

Analysis of Potential Solutions:

There are several possible improvements which revolve around the same concept: recycling vertex data which is still relevant. For both my analysis and my implementation, I will use vertex data describing the political borders of the United States at ten year intervals from 1790 to 1880. Let us consider one key case, in which the client has already downloaded and displayed data from 1880 (Fig. 2) and now wishes to do the same for 1790 (Fig. 3).

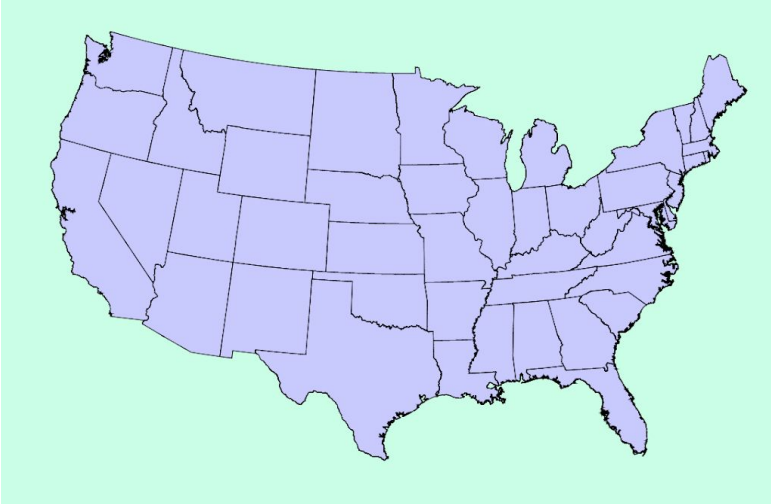

Figure 2. Political boundaries within the United States in 1880.
Map created from vertex data in GeoJSON format.
Original file size: 1,137,954 vertices, 47.2 MB.

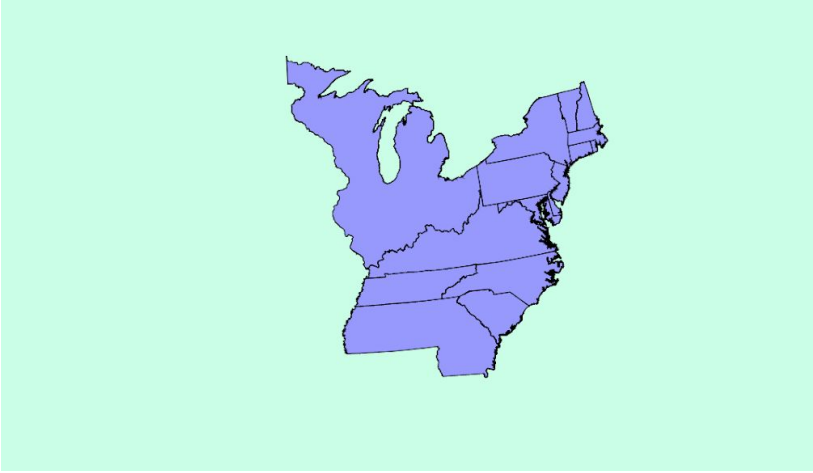

Figure 3. Political boundaries within the United States in 1790.
Map reconstructed from vertex data in GeoJSON format.
Original file size: 482,186 vertices, 19.9 MB.

It is readily apparent that there is significant redundancy if the server were to transmit both data sets in full, as most of the borders which existed in 1790 continued to exist in some form by 1880. To better illustrate this, I represented the data from each year as a set of points, such that the difference of the two sets could be easily determined (Fig. 4).

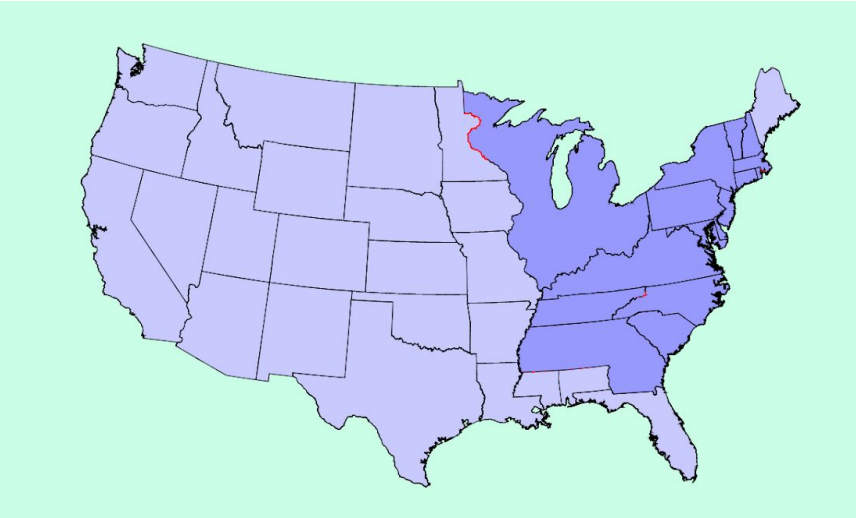

Figure 4. Difference of the set of vertices in 1790 from the set of vertices in 1880. Vertices which exist in the 1790 data (blue) but not the 1880 data (pale blue) are shown in red. There are 7,826 such vertices.

Impressively, only 7,826 out of 482,186 vertices in the 1790 data are not elements of the 1880 data. This implies that, if we were to transmit the data from 1790 in full, 98.4% (19.6MB) would be redundant. This offers a tantalizing potential for significant compression, which could increase responsivity while significantly decreasing the financial cost of sending the data.

While it is trivial to determine the set of points which are unique to a given set and the intersection of two sets, efficiently developing a way to efficiently represent and reconstruct this data is more difficult. I will undertake this in the next section.

<u>One Solution:</u>

My particular solution uses the following process for server-side compression and client-side reconstruction:

Parameters:

- Old data: one year's worth of vertex data which the client has already downloaded, will be used for compression and decompression.
- New data: one year's worth of vertex data which the client wishes to download, will be compressed server-side and decompressed client-side.

Method:

1. Construct a linked set of vertices using a hash table for both the old data and new data, such that each vertex in the set has a previous (consecutive) vertex and a next (also consecutive vertex).
2. Traverse the coordinate data of each new state, such that each coordinate is classified as a hook, line, or junk (all I could think of was a fishing analogy, for some reason).
    a. **Junk**: A coordinate in the new year is said to be junk if it does not exist in the set of vertices for the old year.

    ```
    Example:
    A (old): [[0,0], [1,1]]
    B (new): [[0,1], [1,0]]
    ```

    Both vertices in B are junk.

    b. **Hook**: A coordinate in the new year is said to be a hook if it exists in the set of vertices for the old year and the previous vertex in the new set is not equivalent to the previous vertex in the old set.

```
Example (assume A and B are ordered):
A (old): [[0,0], [1,1]]
B (new): [[0,1], [0,0]]
```

$B[1]$ is classified as a hook because $B[1] \in A$ (at index 0) but $B[0] \neq A[-1]$ (the previous vertex of $A[0]$).

c. **Line**: A coordinate in the new year is said to be a line (or exist on a line) if it exists in the set of vertices for the old year and the previous vertex in the new set is equivalent to the previous vertex in the old set.

```
Example (assume A and B are ordered):
A (old): [[0,0], [1,1]]
B (new): [[0,0], [1,1]]
```

$B[1]$ is classified as a line because $B[1] \in A$ (at index 1) and $B[0]A[0]$ (the previous vertex of $A[1]$). Note that every line element must be preceded by either another line element or a hook element. Thus, every line element must be associated with a hook.

3. Define a resulting list of compressed consecutive vertices such that each element is either a vertex (implying that that vertex is junk and does not exist in the old year) or an object which contains the hook's coordinate data and the length of the line following it, as well as the state in which it exists in the old data set.
4. (Client side) For each element in the compressed list, if that element is a vertex, add that vertex to a linked list. If that element is an hook-line object, locate the hook in its source state, then add the hook vertex to the linked list, as well as every consequent vertex within the length of the line (i.e. the next *n* vertices).

This process results in a linked list of consecutive vertices for each state, equivalent to the data we could have transmitted directly.

Implementation and Analysis:

Linked Vertex Hash Set (Appendix A):

Because mutable objects are not hashable in Python, it was necessary to implement a linked vertex hash set from scratch. The advantage of this unusual data structure is that it can be sequentially traversed, but a search operation can be accomplished in constant time. This means that more complex operations such as set unions, differences, and intersections can be accomplished in linear time. This can be explained intuitively by considering that each of these operations requires that a set be traversed a constant number of times, with each element in the list requiring a search. If each search is O(1), the overall time complexity for a set will be O(n). In a list, these operations would be $O(n^2)$ time complexity, as each search would run in O(n) time.

Decomposition Algorithm (Appendix B):

The decomposition contains two major steps, both of which run in linear time. First, the linked vertex hash set must be constructed for each data set. For a data data set with n vertices, this will require n additions, each of which runs in constant time, yielding overall linear time. The second major step is compressing the new data based on the old data. This requires traversing the new data, once, including a constant number of searches and comparisons for each element. Thus, this step will also run in linear time.

Reconstruction Algorithm (Appendix C):

Reconstruction time complexity will depend on the number of hooks in the compressed data, as well as the average number of vertices in a state. For each vertex in the compressed data, a constant-time append operation is required. However, for every hook-line object, one linear time search must be completed for the source state. Thus, the time complexity of decompression will be proportional to the number of hooks times the average number of vertices in a state. This could conceivably be improved by constructing another linked vertex hash set.

Results:

Some data grooming was necessary to bring this project to a manageable scale. First, each state's vertex data was trimmed, such that only contiguous land is included in the data. This offers room for the algorithm to be further developed, such that it can identify and handle states with multiple, unconnected territories.

Compression of the 1790 data using the 1880 data produced a 4.1MB JSON file which was successfully reconstructed on the client side and ultimately identical to the original file. This represents a compression rate of 79.4%, which does not match our theoretical maximum for lossless compression, but is still an excellent improvement.

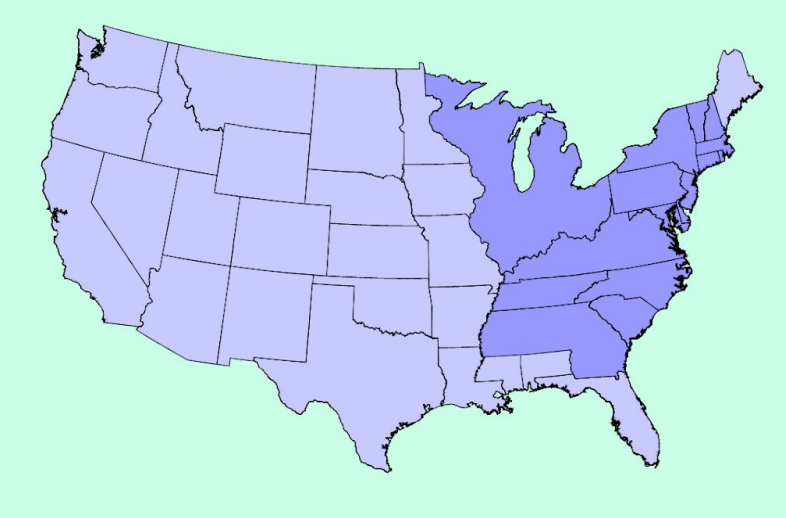

Figure 5. 1790 borders reconstructed from compressed data and 1880 data. The file size was reduced by ~79.4%.

The compression rate for 1840 data was even more impressive, with a smaller compressed file size (3.2MB) despite a larger data set (34.4MB), representing a compression rate of about 90.7%.

I hypothesize that the difference in theoretical and observed compression rates comes from a flaw in my algorithm - it only probes in one direction. If the Virginia data defines the VA-NC border from west to east and North Carolina defines the same border from east to west, these borders cannot currently reference each other using a hook-line object.

Guidelines for Practical Application:

While the compression algorithm runs in linear time, the large number of vertices in some datasets can produce long run-times. In a practical context, it is probably best that these compressed files be precomputed and stored, rather than produced as needed.

The problem of efficient compression becomes significantly more complex when the client has downloaded more than one year's worth of data. The simplest efficient solution to this is to simply use the set intersection method to determine which old set shares the most vertices with the new set. A more sophisticated solution would create a joint set of vertices which could be referenced - this is beyond the scope of my implementation.

The United States is somewhat unique in that its borders tend to grow over time, without changing existing borders significantly. In this case, I believe the optimal solution for network load and time-complexity is to precompute all compressed files in terms of 1880, which should be downloaded as the page loads. This may not apply to other countries, but a similar approach may be appropriate for a world map.

An ideal implementation would load vertex data as needed, based on the zoom level - e.g. a zoomed in view of Africa would require only African vertex data with relatively high precision. A zoomed out view of the world would require all the vertex data for that year, but with less precision. Either way, space is conserved.

Full Implementation
https://github.com/henryhill1999/GeoCompress

Appendix A:
Linked Vertex Hash Set (Server Side)

```
import copy

class LinkedVertex:
    def __init__(self, crd, prev = None, nxt = None,src = None):
        self.src = src
        self.crd = crd
        self.prev = prev
        self.nxt = nxt

#checks equality between two coordinate pairs
def verticesEQ(v1,v2):
    return v1[0]==v2[0] and v1[1] == v2[1]


class LinkedVertexHash:
    def __init__(self,buckets = 2500000):
        self.els = [None]*buckets
        self.buckets = buckets
        self.size = 0

    def items(self):
        for b in self.els:
            if b:
                for el in b:
                    yield el

    def contains(self,el):
        h = self.vHash(el)
        if self.els[h]:
            for v in self.els[h]:
                if verticesEQ(el,v.crd):
                    return v
        return False

    def add(self,el):
        if self.contains(el.crd):
            return
```

```
        self.size+=1
        h = self.vHash(el.crd)

        if self.els[h] == None:
            self.els[h] = [el]
        else:
            self.els[h] += [el]

    def addPoly(self,items,src=None):
        crds = items

        prev = None
        for v in crds:
            newVertex = LinkedVertex(crd = v, prev = prev, nxt = None,src =
src)
            self.add(newVertex)

            if prev == None:
                first = newVertex
            else:
                prev.nxt = newVertex

            prev = newVertex

        newVertex.nxt = first
        first.prev = newVertex

    def remove(self,el):
        h = self.vHash(el.crd)
        if self.els[h]:
            for i,v in enumerate(self.els[h]):
                if verticesEQ(el,v.crd):
                    self.els[h].pop(i)
                    self.size-=1
                    return
        raise f'self set does not contain {el}'

    def vHash(self,v):
        p = 53
        return int((v[0]*(p)+v[1]*(p**2)))%self.buckets
```

```
    def asList(self):
        res = []

        for b in self.els:
            if b:
                res+=b

        return res

    @staticmethod
    def union(s1,s2):
        res = LinkedVertexHash()

        for i in s1.items():
            res.add(i)

        for i in s2.items():
            res.add(i)

        return res

    @staticmethod
    def intersection(s1,s2):
        res = LinkedVertexHash()

        for i in s1.items():
            if s2.contains(i.crd):
                res.add(i)

        return res

    @staticmethod
    def difference(s1,s2):
        res = copy.deepcopy(s1)
        its = LinkedVertexHash.intersection(s1,s2)

        for i in its.items():
            res.remove(i)

        return res
```

Appendix B:

Decomposition/compression algorithm (Server-side)

```
from itertools import islice

def decompose(new, prev):

    prevSet = setFromJSON(prev)
    newSet = setFromJSON(new)

    newStates = new['features']
    compressedStates = []
    for s in newStates:
        crds = s['geometry']['coordinates']
        vs = []

        en = iter(enumerate(crds))
        count = 0
        for i,c in en:
            hook = prevSet.contains(c)
            if hook:
                record = [hook]
                j = 0
                line = hook.nxt
                while (verticesEQ(crds[(i+j+1)%len(crds)], line.crd) and
                        not verticesEQ(line.crd, hook.crd) and
                        i+j < len(crds)):

                    j+=1
                    record += [line]
                    line = line.nxt
                count+=j
                vs += [{'hook':hook.crd, 'length':j, 'src':hook.src}]

                next(islice(en,j ,j), None)

            else:
                count+=1
                vs += [c]
        compressedStates.append(vs)
    return compressedStates
```

Appendix C:

Reconstruction/decompression algorithm (client side)

```
function reconstruct(vs,prev){
  res = []
  vs.forEach(v => {
    if (Array.isArray(v)){
      res.push(v)
    }else {
      hook = findVertex(prev.features[v.src].geometry.coordinates,v.hook)
      for (let l = 0; l < v.length+1; l++){
        crds = prev.features[v.src].geometry.coordinates
        res.push(crds[(hook+l)%crds.length])
      }
    }
  });
  return res
}

function findVertex(ls,v){
  for (let i = 0; i<ls.length; i++){
    if(ls[i][0]==v[0] && ls[i][1]==v[1]){
      return i
    }
  }
  console.log('not found')
}
```