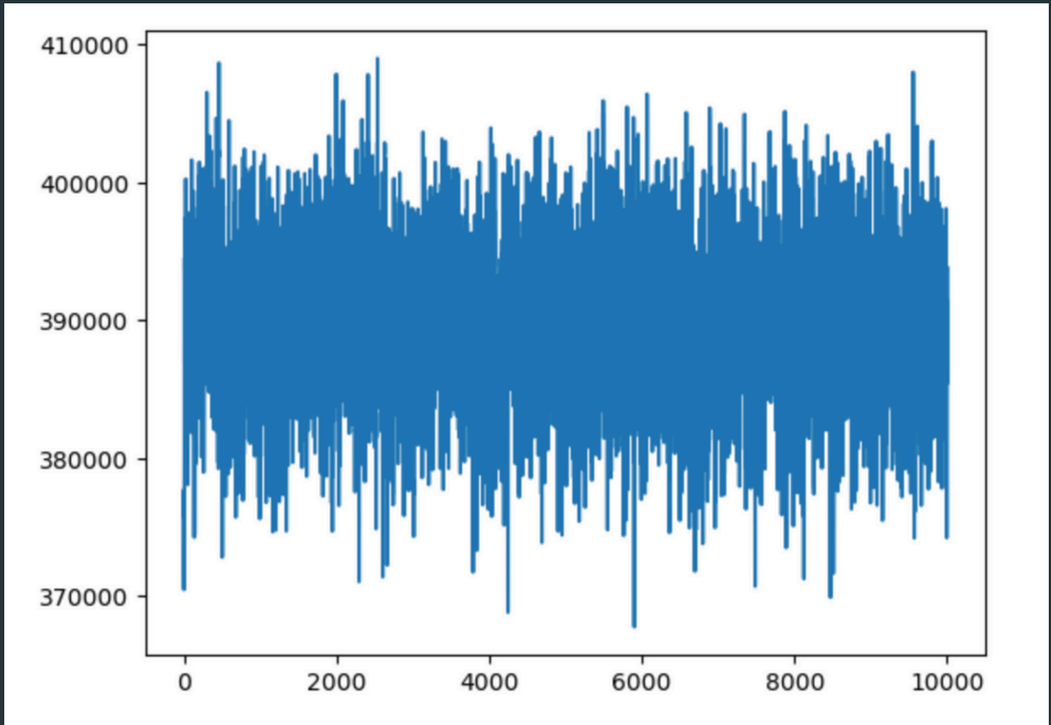
TSP Project

Will, Neelay, Ariana

Table of contents



First run performance on random sub tour

Intial Tests/Ideas

The first metaheuristic we used for TSP was the genetic algorithm. This included just a raw implementation of the ideas we saw in class.

We saw lackluster results from this implementation and moved onto new ideas quickly.

The next ideas we implemented were smart-crossover and messiahs.

For clustering, we used the provided kmeans algorithm. We also kept the k of 300, and the stopping condition at 3 iterations constant

Motivating & Introducing our Appraoches:

- Genetic Algorithms:
  - First algorithm we implemented in our approach to TSP
  - Straightforward implementation:
    - motivated to leverage this by adjusting its multiple possible population/mutation parameters
    - first changes: messiah hyper-parameters, elitism approach and smart crossover
- ACO:
  - Initially chosen as a method to connect centroids of results from GA clusters
  - Implemented parallelization of ants
- Greedy Algorithm:
  - Wanted to assess the limits of metaheuristic approaches:
    - can the simple solution beat GA/ACO?
  - Next best neighbor approach

Messiahs from SA

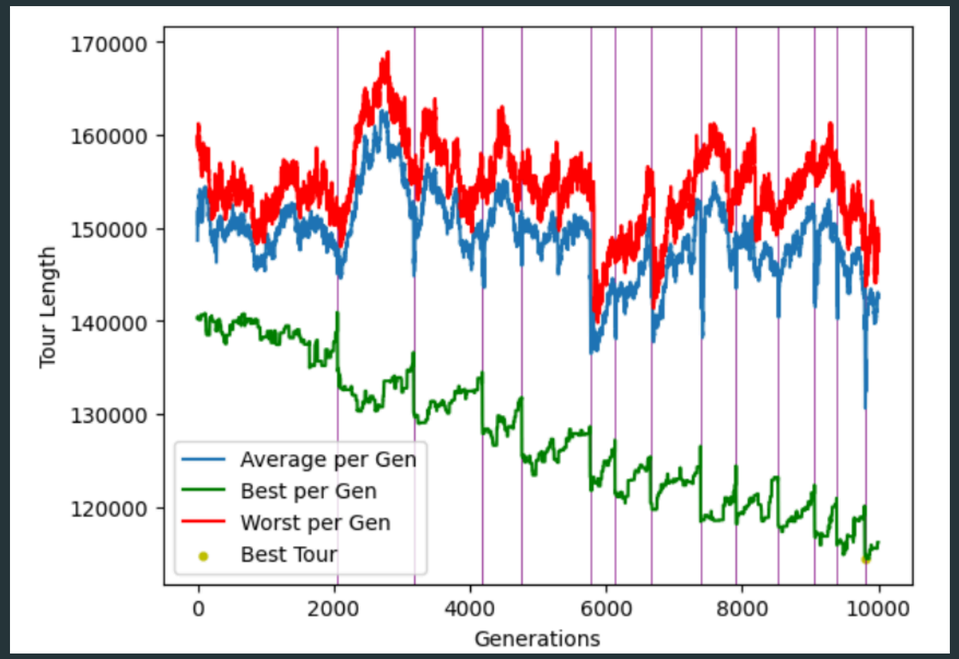
The introduction of messiahs and the two hyperparameters it adds allow us to shift our simple genetic algorithm to be a hybrid between genetics and simulated annealing, with the parameters dictating the strength of the annealing.

The messiah implementation consisted of taking the well-performing tours found in the SA code and using those as parents for genetic algorithm generations where the tours began getting worse.

Smart-Crossover

The smart-crossover was implemented as a way of generating new “children” tours that were valid. We implemented a fast self check that ensured each new tour had no duplicates. The crossover method provided in class did not account for gene duplication.

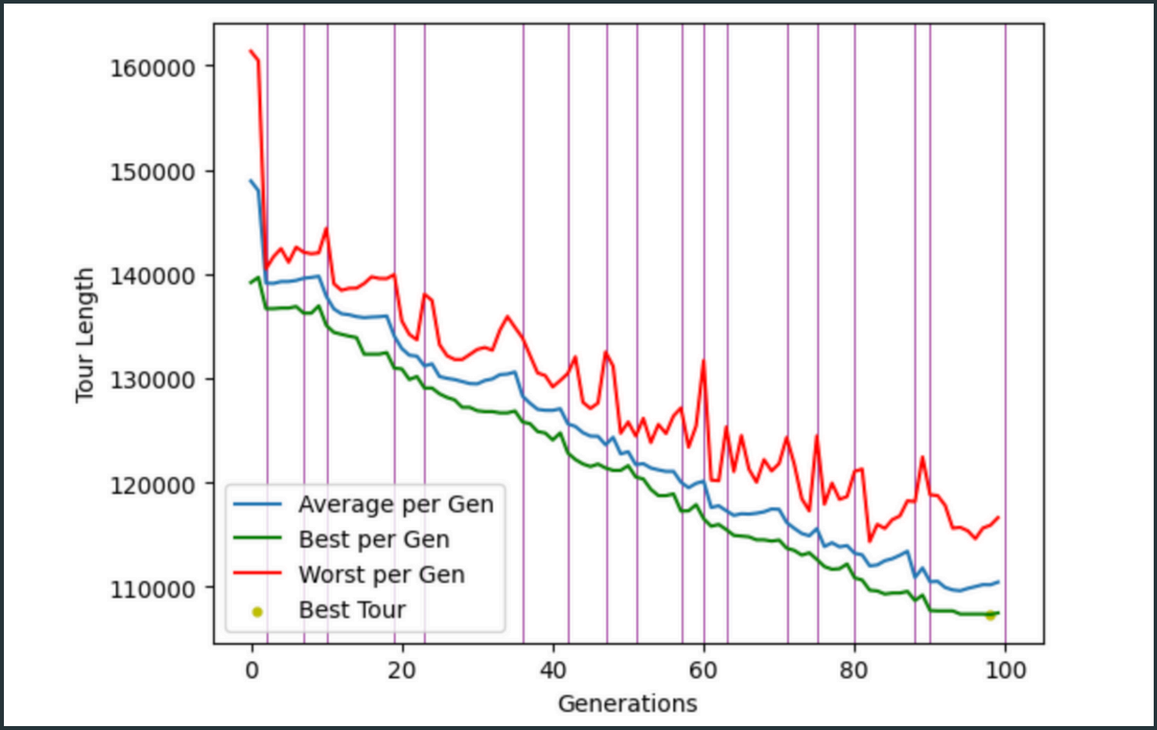
Messsiah and Smart-Crossover Performance (for single cluster of 300)



- Parameters
  - *gens=10000*
  - *population_size=100*
  - *darwin=0.3*
  - *mutate_chance_child=0.2*
  - *elite_clone_percent=0.1*
  - *mutate_chance_elite=0.05*
  - *messiah_margin=5000*
  - *messiah_percent=0.05*

Here you can see where each messiah was injected and the improvements that came from them

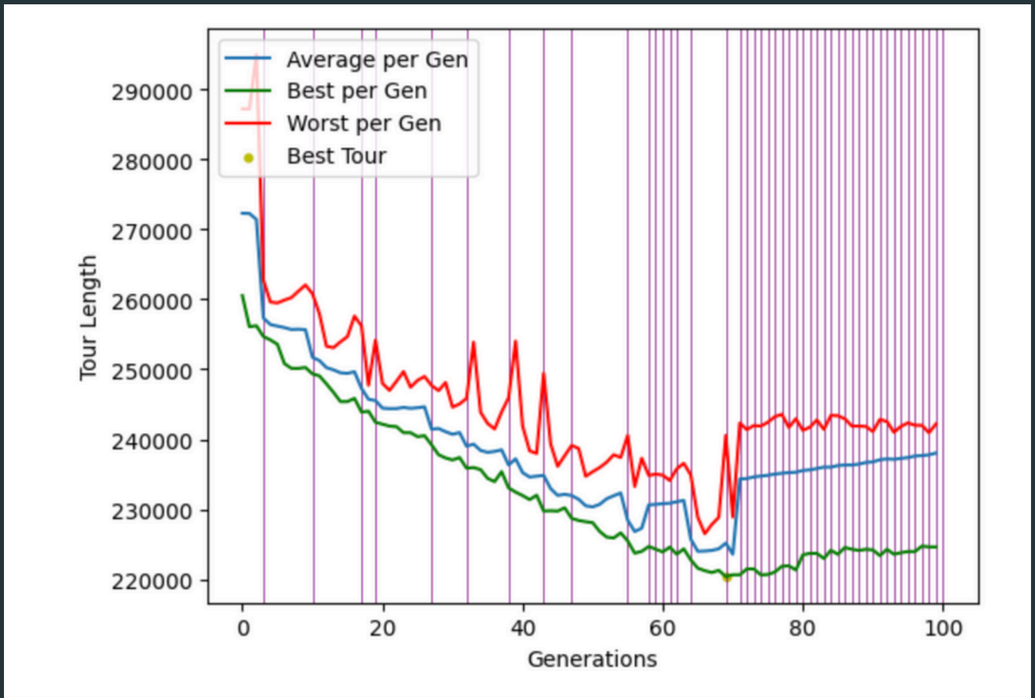
Running Genetic Algorithm on the all Clusters




```
:  #parameters for running on all clusters
   gens = 100
   darwin = .50
   mutate_chance_child = .35
   mutate_chance_elite = .75
   population_size = 100
   elite_clone_percent = .35/darwin
   messiah_margin = 100
   messiah_percent = .50
   messiah_num = int(messiah_percent * population_size)
```

Here you can see where each messiah was injected and the improvements that came from them.

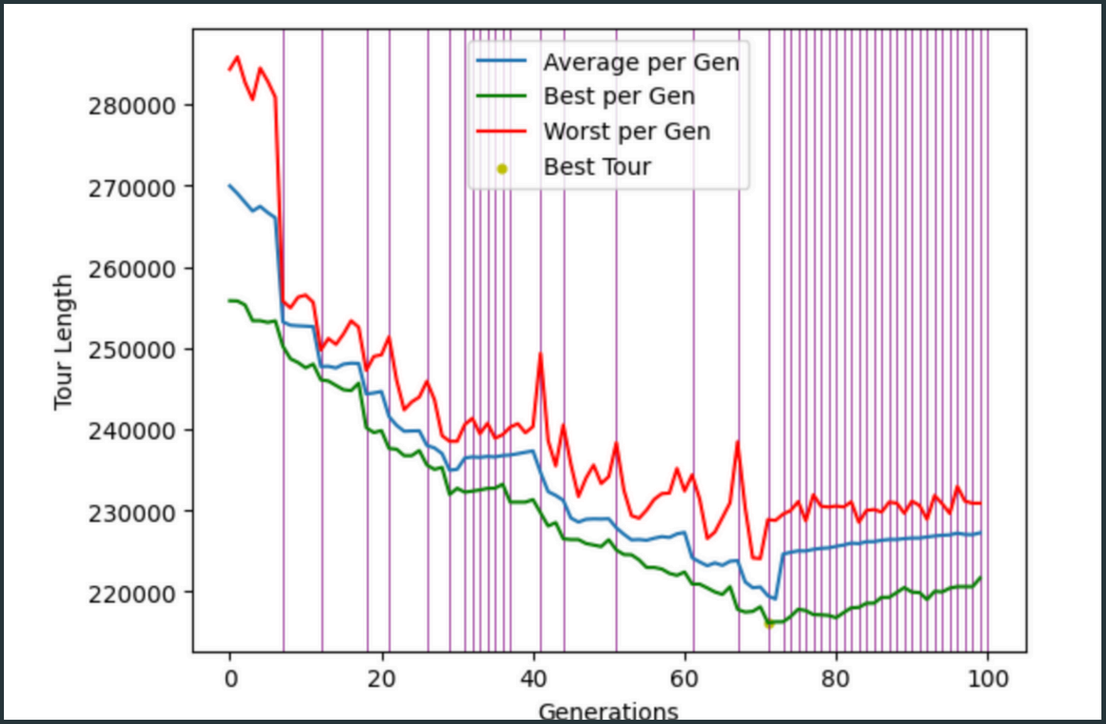
Messsiah and Smart-Crossover Performance




```
:  #parameters for running on all clusters
   gens = 100
   darwin = .50
   mutate_chance_child = .35
   mutate_chance_elite = .75
   population_size = 100
   elite_clone_percent = .35/darwin
   messiah_margin = 100
   messiah_percent = .50
   messiah_num = int(messiah_percent * population_size)
```

But here, you can see a case where we get stuck in a “local” optimum

Messsiah and Smart-Crossover Performance




```
:  #parameters for running on all clusters
   gens = 100
   darwin = .50
   mutate_chance_child = .35
   mutate_chance_elite = .75
   population_size = 100
   elite_clone_percent = .35/darwin
   messiah_margin = 100
   messiah_percent = .50
   messiah_num = int(messiah_percent * population_size)
```

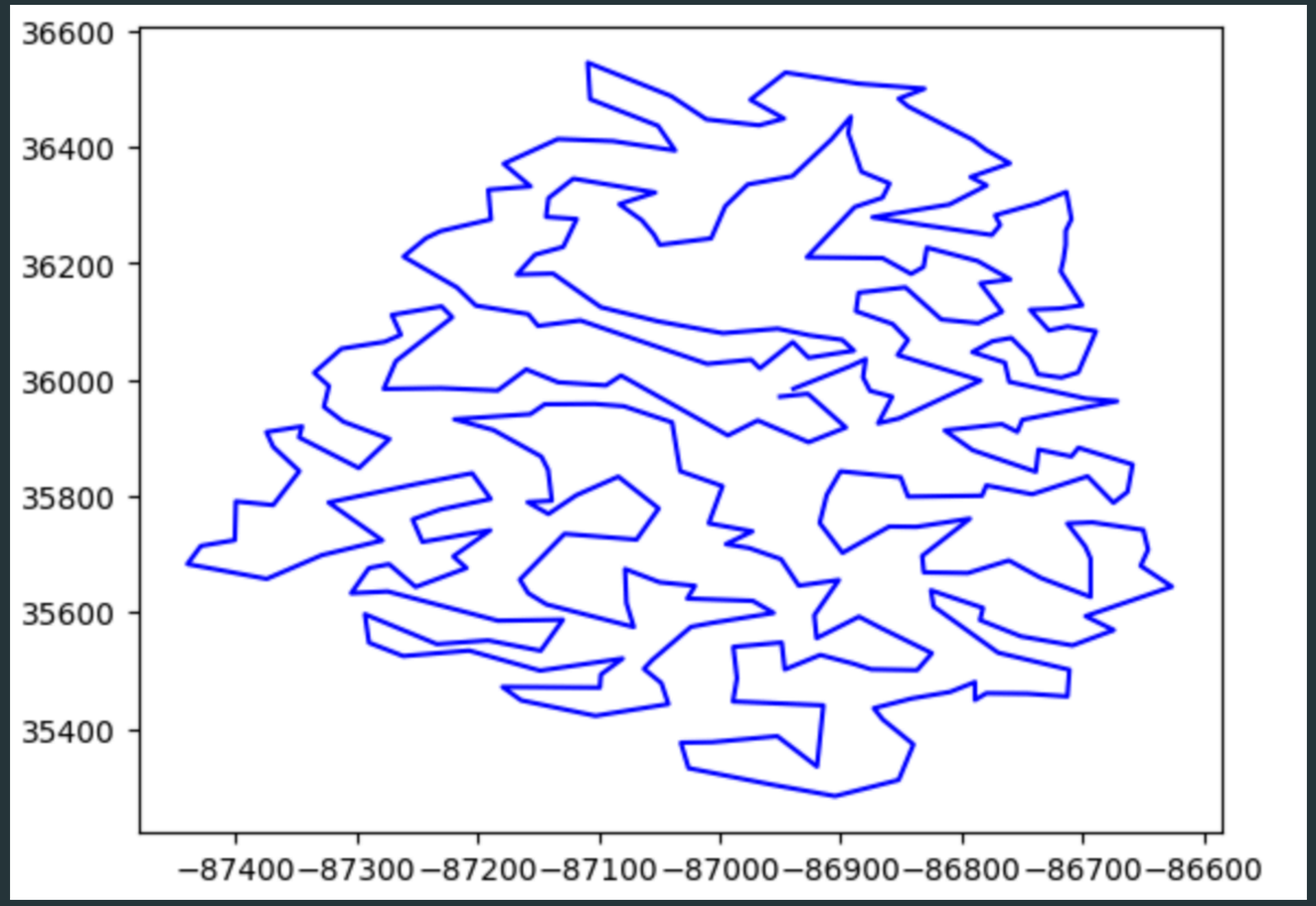
Or maybe not, enough generations and it can potentially go back in the right direction.

A particularly horrendous cluster

The Gospel of Matthew

(because of this uncrossing step, we see an 83% decrease in final tour length)

We uncross each sub tour



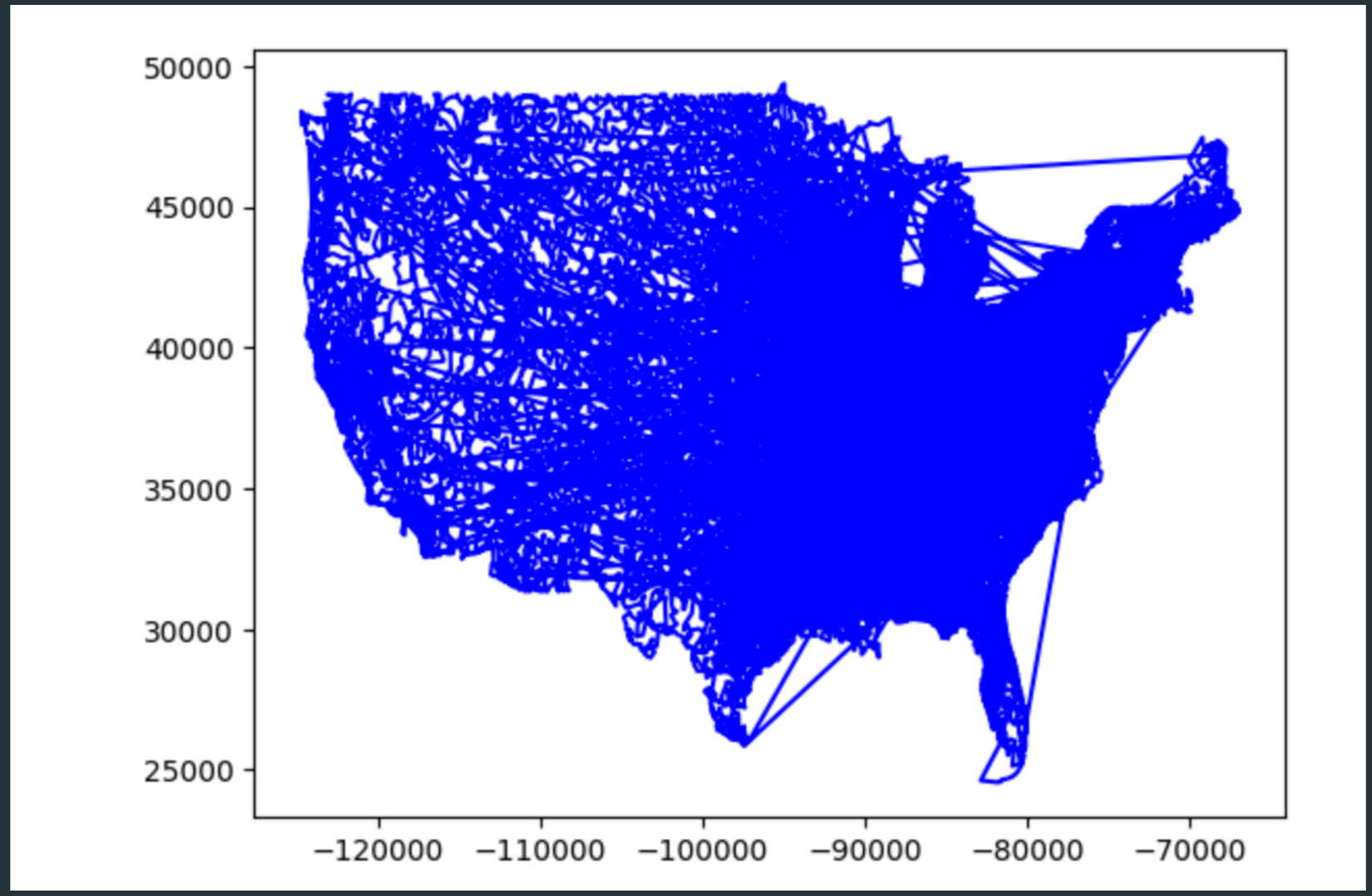
An uncrossed sub tour

We then simple concatenated (used existing concat points)each subtour together in the original cluster sequence (in order)

Tour Length of 11, 791, 646 in 1 hour and 45 minutes (38 min in parallel)

Without the Gospel of Matthew, we have a tour length of 74 million.

In Matthew We Trust.

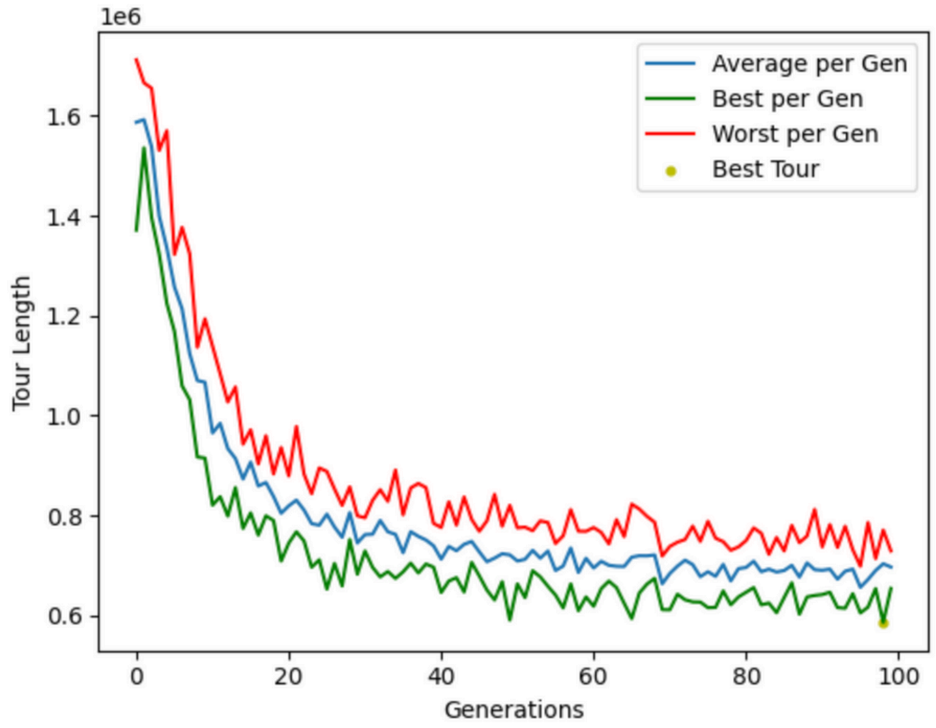


Optimizing Cluster Sequence for Concatenation

We implemented ACO too. Even with ants parallelized, runtime was still subpar compared to genetic, however parallelization did improve runtime drastically.

```
gens = 100 #number of generation of ants
ant_num = 10 #size of each generation of ants
#when calculating edge probability weights, pheromone is squared by alpha.
#Thus, alpha > 1 is for a more exploitative approach of swarm ai, putting greater importance on the pheromone
#An alpha of < 1 is for a more exploratative approach, more likely to explore new paths.
pheromone_alpha = 1
#when calculating edge probability weights, the inverse of the edge distance is squared by beta.
#Thus, a beta > 1 favors a greedy approach, increasing the importance of the heuristic a priori knowledge
#A beta < 1 then decreases the heuristic influence, allowing exploration of longer paths
distance_beta = 2
#when ants apply pheromone to an edge, p_inverse divided by the tour length is the calculation
#So, p_inverse represents the scale at which pheromone is applied
p_inverse = 10
p_decay = 0.8 #the percent at which pheromone decays every generation. A decay of 1 clears the pheromone map every generation.
```

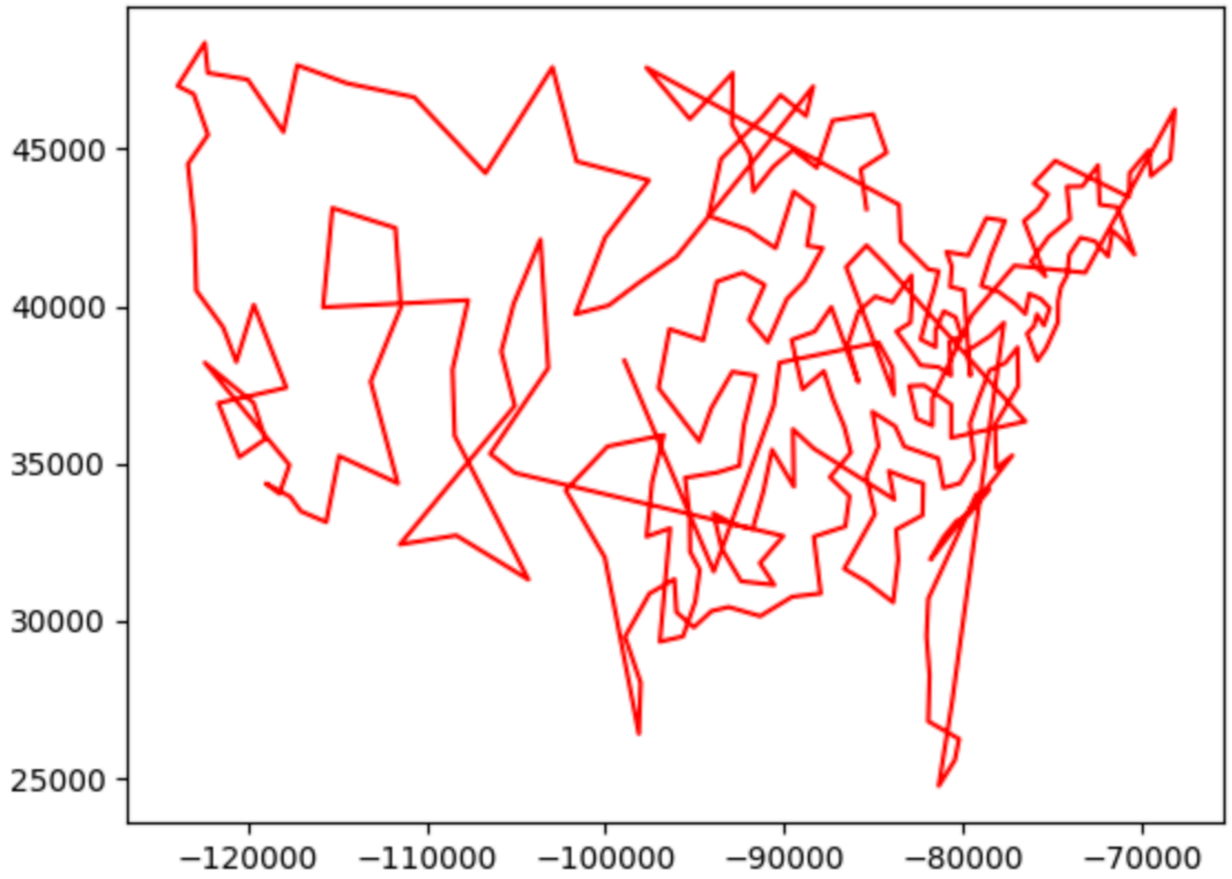
24.80716109275818 seconds



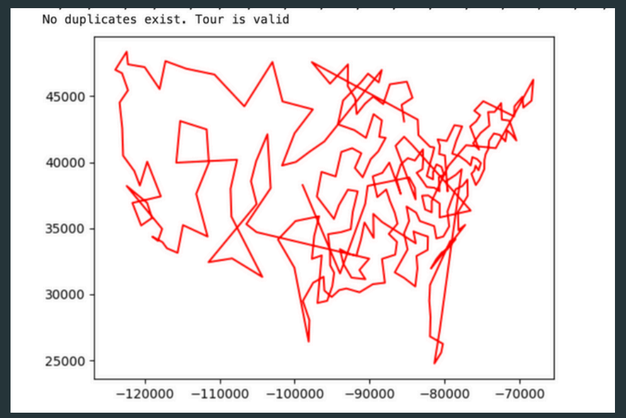
24 second runtime.

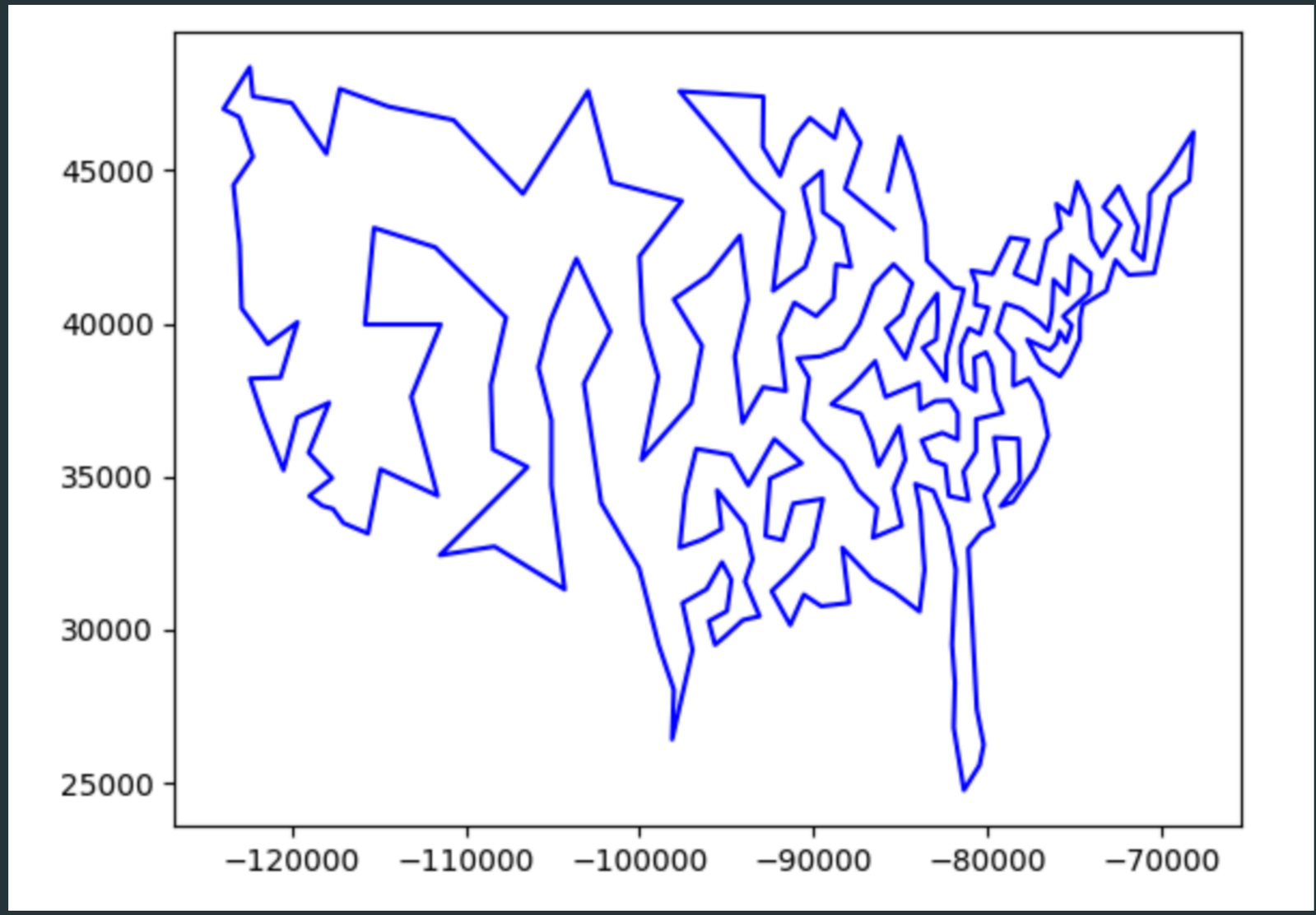
Can see relative convergence with a small number of cities/centroids

No duplicates exist. Tour is valid



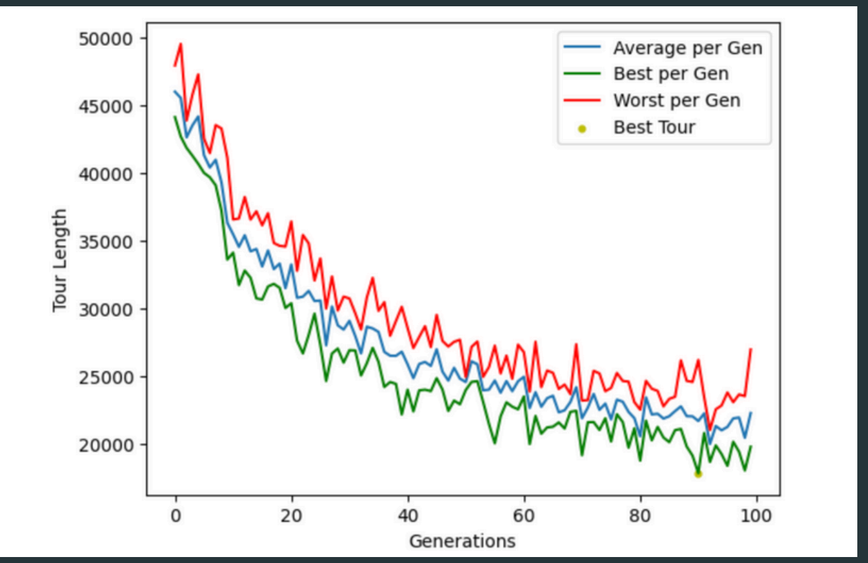
The Gospel of Matthew



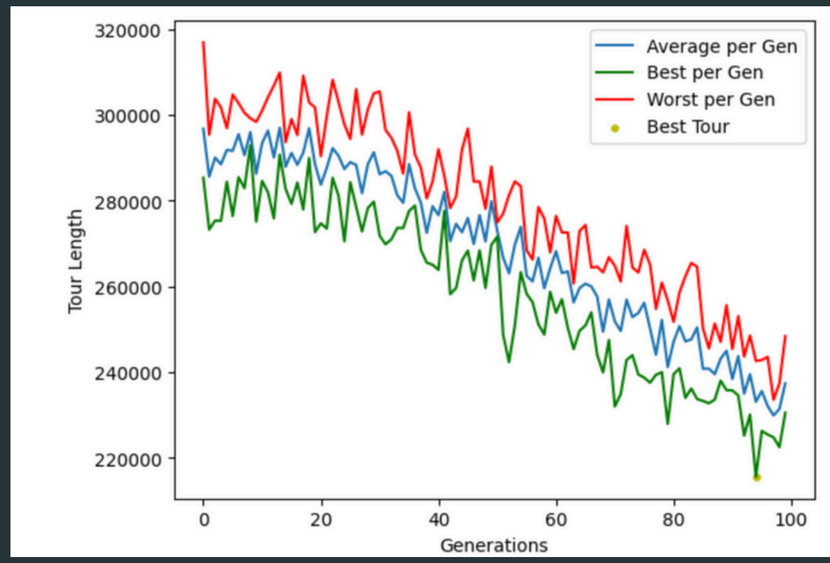


Running ACO on all clusters (eventually we parallelized cluster execution as well for better runtime, as well as with genetic on all clusters)

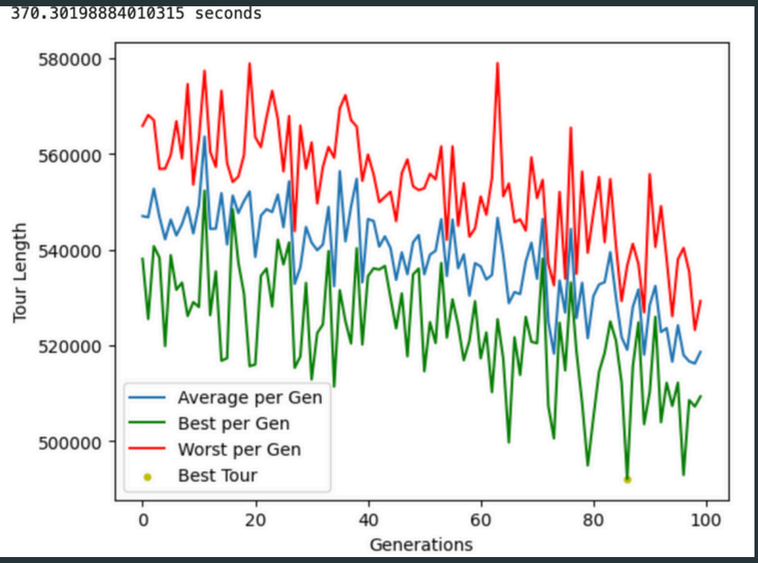
We observed that the greater the number of cities, the more difficult it is for the swarm to continually improve



196 cities



560 cities



900+ cities

This gives us motivation to increase number of clusters

Lazy Implementation of Longest Edge Removal for determining an optimal concatenation point

```
regions_ordered = []
for i in range(0, len(cluster_sequence)):
    regions_ordered.append(regions[cluster_sequence[i]])
```

```
raw_tours_ordered = []
for i in range(0, len(tours_ordered)):
    raw_tour = []
    for j in range(0, len(tours_ordered[i])):
        raw_tour.append(regions_ordered[i][j])
    raw_tours_ordered.append(raw_tour)
```

```
new_starts = []
for raw_tour in raw_tours_ordered:
    longest_dist = 0
    new_start = 0
    for i in range(0, len(raw_tour)-1):
        if dist(raw_tour[i], raw_tour[i+1]) > longest_dist:
            longest_dist = dist(raw_tour[i], raw_tour[i+1])
            new_start = i
    new_starts.append(new_start)
```

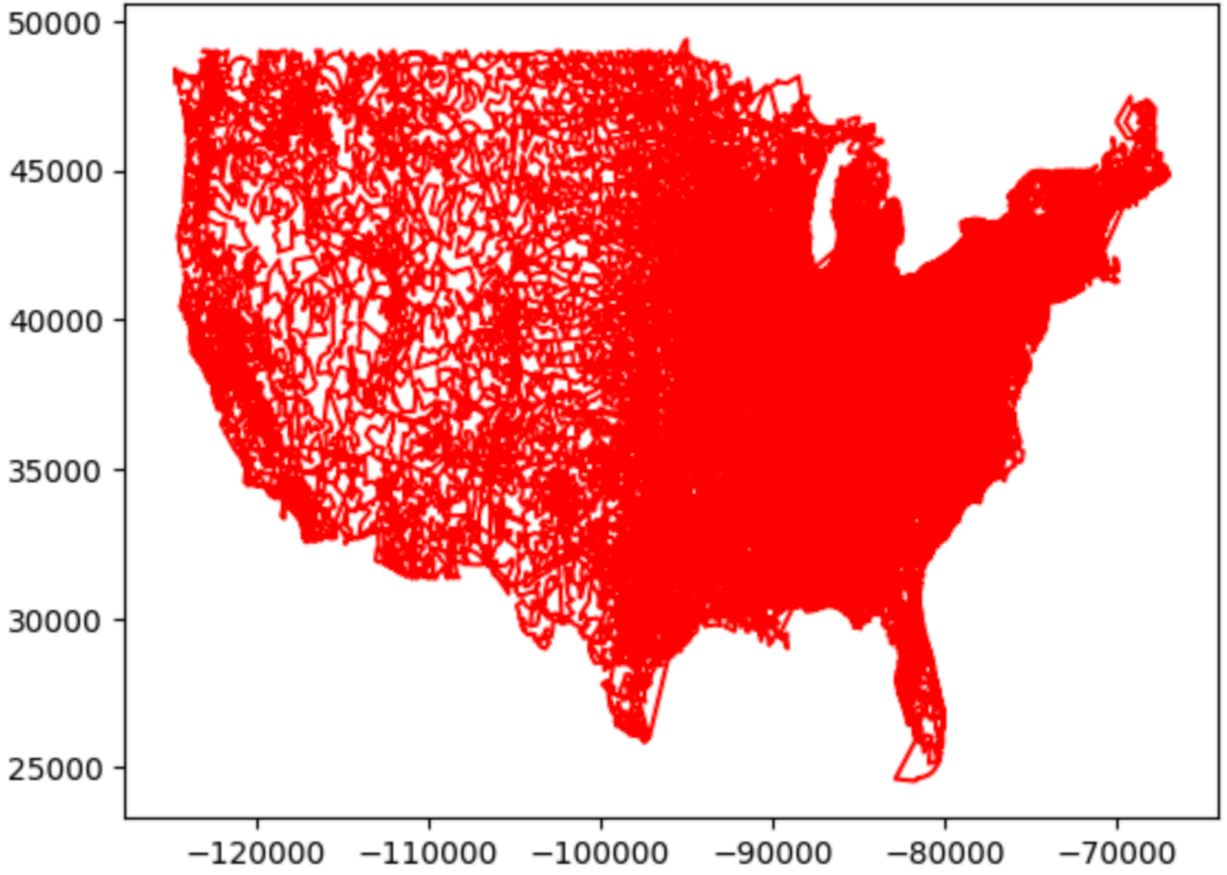
```
rotated_tours = []
for i in range(0, len(tours_ordered)):
    new_tour = tours_ordered[i][new_starts[i]:].tolist() + tours_ordered[i][:new_starts[i]].tolist()
    rotated_tours.append(np.array(new_tour))
```

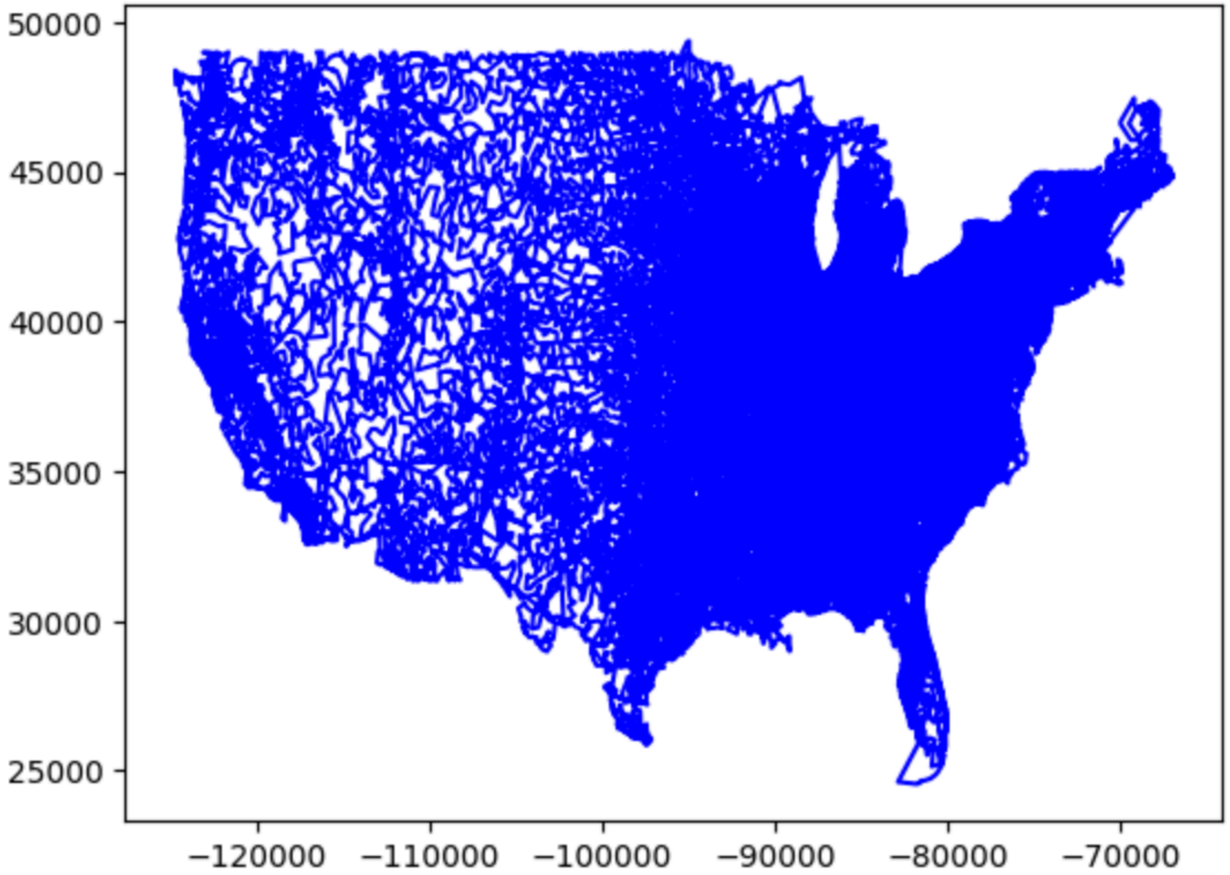
```
start = time.time()
final_tour = concatenate_tours(tours_ordered, CITIES, regions_ordered)
print(time.time()-start, "seconds")
```

We first used the cluster sequence ACO found to concatenate the sub tours from our genetic algorithm, resulting in a major decrease in tour length from 11 million : 7,694,815

Using the longest edge concatenation points, it decreased to : 7,673,970

Showing normal concatenation points vs longest edge concatenation points





We then run ACO on all the clusters (ants in parallel, clusters still sequentially)

And we uncross all of the resulting sub-tours.

We use the cluster sequence found by ACO between the cluster centroids as the order we visit each sub tour.

And we use the longest edge removal concatenation point to bring our final tour length to 7,672,426, taking 6 hours (around 2 hours when clusters parallelized)

Results

Tour length of uncrossed, optimal cluster sequence and optimal tours in each cluster: 7,672,426

Steps in our final algorthm:

-Cluster data set (I)

-Run ACO to find centroid path (II)

-Uncross the centroid tour (III)

-Genetic or ACO (ants in parellel) on each cluster (in parellel) (IV)

-Uncross each cluster (V)

-Longest edge replacement concatenation in cluster tour order (VI)

-Run time statistics:

-ACO on centroids: 24 seconds (II)

-Uncrossing ACO centroids: 1 second (III)

-Running genetic on clusters (not in parallel): 106 minutes (IV)

-Running genetic on clusters (in parallel): 38 minutes (IV)

-Running ACO on clusters (not in parallel): 5 hours 50 minutes (IV)

-Uncrossing cluster tour: 1 minute 50 seconds (V)

-Concatenating + longest edge replacement: 2 minutes 40 seconds (VI)

Greedy Implementation

We were intrigued by the idea of running a straight greedy algorithm, and chose to choose a solution to the TSP that invoked finding the next closest city, and moving there

```python
# Greedy algorithm to solve TSP
#@jit(nopython=True)
def greedy_tsp(cities):
    N = len(cities)
    visited = np.zeros(N, dtype=np.bool_)
    tour = np.empty(N + 1)
    tour[0] = 0  # Start from the first city
    visited[0] = True
    total_distance = 0.0
    current_city = cities[0]

    for k in range(1, N):
        next_city, distance = find_nearest_neighbor(current_city, cities, visite
        tour[k] = next_city
        visited[next_city] = True
        total_distance += distance
        current_city = cities[next_city]

    # Return to the starting city to complete the tour
    total_distance += euclidean_distance(current_city, cities[0])
    tour[N] = 0

    return tour, total_distance
```

Greedy Implementation Results & Next Steps

- Final tour length for the greedy algorithm: 7,692,556.969 tour length (~2 hour runtime)
- Next steps running on a strictly greedy appraoch:
  - Currently running this program invoking Matthew's edge uncrossing approach
- Conclusion:
  - This greedy algorithm quickly deliveres a good tour length, however there are still improvements to be made on this result.
  - ACO: 7.67M m tour length (6 hours) vs Straight Greedy: 7.69M m tour length (2 hours)
    - simplicity works!
- Next ideas:
  - The greedy algorithm is currently taking about 1/3 of the runtime of ACO (without invoking edge crossing):
    - Implement a pre-constructed pheremone map into the ACO program using the greedy approach result

Further Ideas

Letting clustering run longer/stopping condition

More clusters

Greedy Algorithm for cluster sequence

Longest Edge replacement modification

Running ACO on everything in parallel with more ants/more clusters

Start ACO in each cluster with a high degree of pheromone on greedy edges

Introducing changing temperature