Path Regression on Limited Sensor Data Using Simulated Annealing

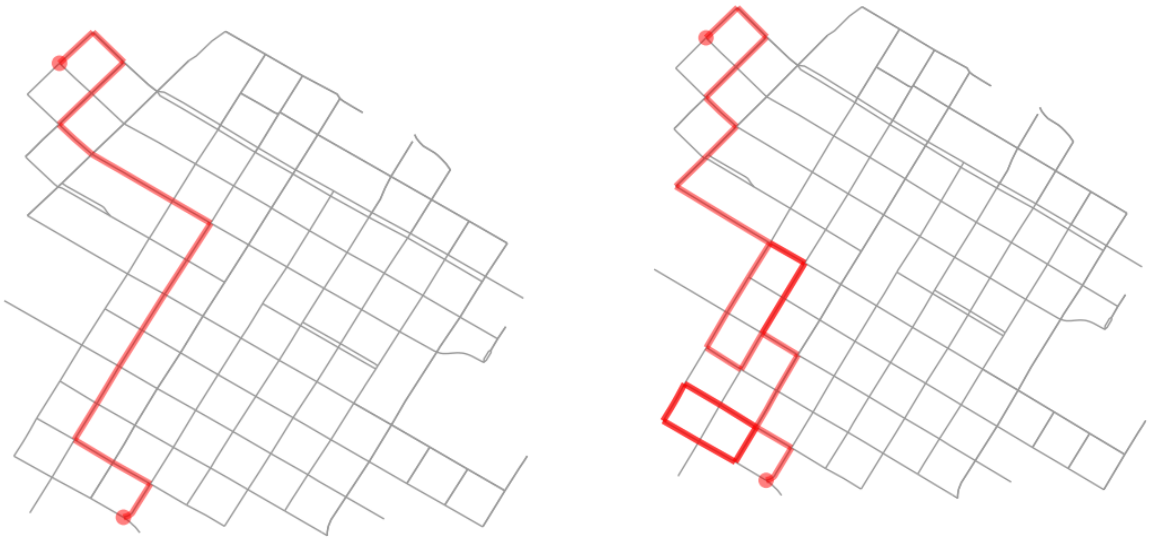
Graham Droge

Abstract: There are many algorithms that are in used today that fall under the title of a meta heuristic algorithm. One such algorithm is simulated annealing, which is built on the process of annealing in metallurgy to solve optimization problems on a large search space. The following paper discusses the use of an algorithm that inherits characteristics from simulated annealing to a problem of matching a noisy driven car path to a road map of given intersection nodes. Given a driven path with a limited set of sensor data as input, we extract useful features for the corresponding node matching algorithm. Results are then analyzed and a discussion of possible enhancements to the overall process is given.

1. Introduction

The general problem of filtering out noise in data to better represent the true distribution is one that has produced many possible solutions. One such solution is the Kalman filter, which uses a series of acquired data points to get a better representation of the model for the physical quantities involved. While these methods prove to be effective in situations where a target variable which we are interested in modeling is available to us, what are our possible solutions when the target variable is not available to us? This is the question with which the following paper is built on. The specific problem can be stated as followed: Assume we are given a data capture (start point and end point is known) that corresponds to a cars driven route over a specific time where the data is composed of a limited amount sensor data excluding GPS and has a significant amount of noise in it, can we re-engineer the true path from the data on a street network. The sensor data that we have allows us to build a representation for the path that was taken which means we can use this to create an image of the path, extract features such as turns from the path, or compute distances from the path. Figure 1 below shows an example of a path on a small search space in Minneapolis.

I use the *osmx* python library to acquire a street network that is comprised of a graph of nodes that correspond to intersections. This graph of nodes will be the search space for which the algorithm wishes to find the best match through the nodes the corresponds to the noisy data path. Given that the search space of the street network is discrete and can be very large for a dense city we need an algorithm that isn't based on gradient descent but rather uses a probabilistic model. This is why simulated annealing is the algorithm with which my algorithm is based off of. Simulated annealing is a probabilistic approach to approximating the global optimum of a function and provides good robustness from getting trapped in local minimums. The algorithm is based on the annealing in metallurgy, where metals are heated to a very high temperature and cooled slowly to produce certain chemical effects. There are 5 parameters that need to be considered when building a simulated annealing algorithm , the search space, goal function (energy), next state procedure, acceptance probability, and the annealing schedule. The final objective is to minimize the energy of the system which is modeled by the goal function. All of these parameters are considered to some extent in my algorithm with slight modifications to better facilitate the implementation to the given problem. I will provide an analysis on the effectiveness on the algorithm for varying sizes of search space and varying types of driven paths. Lastly I will provide further discussion on what I would want to do to improve the algorithm and fine-tune it to provide more robustness to varying paths and search spaces.



(a) Path one

(b) Path two.

Figure 1: Example of varying difficulty of paths

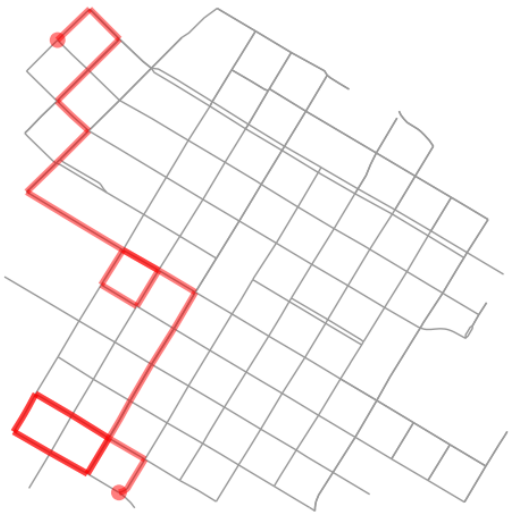
2. Design and Implementation

The following section details the two main parts of the algorithm process, feature extraction and algorithm design. Feature extraction is our method for acquiring relationships and features from the data of the driven path that can aid in fine-tuning the optimization algorithm. An important characteristic of the features that are extracted are that they are invariant to the scaling and units involved and are built on normalized values. This way the feature relationships in the noisy data capture are relatively the same in the true data path as long as the noise has a somewhat linear effect on the data.

2.1. Feature Extraction

This section will cover how the feature extraction was implemented and how these features tie into the algorithm design. First off, I thought I could get a real data capture from a vehicle that I would use to build these features but I had issues obtaining that so it won't be the full feature extraction. Although I won't be able to show the exact feature extraction I will go over how the same type of features can be determined through computer vision on simplified routes. Note that I'm not making an impossible assumption in obtaining the full feature set in a data capture given I have done it before and isn't a real difficult task, also the project is more about the optimization algorithm than the data pre-processing. I will model the noisy data values by computing the following parameters for a random path through the search space and adding some noise so that any sort of algorithm that marched along the path and match to the closest node would fail.

The first feature that is extracted is T2D (turn to distance) ratio. This measure provides a list of values that correspond to turns in the route and are normalized over the full distance of the path. What is considered a turn is a parameter that can be tuned based on the expected node density of the area you believe the path was driven in. For example, if the path existed in a dense city you would expect more turns and these turns would correspond to more 90 degree turns like city blocks. If you thought the path was in a smaller rural area you would expect the roads to be more curved and intersections to be sparser so you would set the parameter to be less strict in it's turn counting. The figure below shows the T2D values for the path shown. You can see they are normalized to the distance of the true path.



(a) Randomly generated path

```
In [11]: get_turn_info(G, route1, true, 45)
Out[11]:
[(1, 0.03118139174324386),
 (2, 0.060727502150196934),
 (3, 0.1221366072910537),
 (4, 0.15240369381195965),
 (5, 0.21323399046368813),
 (6, 0.3199917387171247),
 (7, 0.3493886084830928),
 (8, 0.3782623108203946),
 (9, 0.40699455283448777),
 (10, 0.4652540438791063),
 (11, 0.613106959093447),
 (12, 0.6721647578953723),
 (13, 0.7027142935282846),
 (14, 0.7622938451556442),
 (15, 0.7913177312027523),
 (16, 0.8503755300046777),
 (17, 0.88092506563759),
 (18, 0.9696197452205272)]
```

(b) Turn info corresponding to path

Figure 2: Example of T2D values for a random path

The second feature that is extracted is the D2E (direct to end) parameter. This parameter gives a measure on how "often" the path moves in the direction towards the end node. If this value is high corresponding to more of a direct path to the end node then our algorithm should take steps towards the end node with a higher probability. If the value is low this means the path does not take a direct approach and the algorithm needs to take steps in a wider range of directions to better fit the path. This parameter can be thought of as a kind of Lipshitz constant used in the Lipshitz gradient condition. For a constant step size gradient descent method one should choose a step size corresponding to $\frac{L}{2}$, which has similarities to the parameter I use. To actually obtain this value there a number of different techniques you could use. If you have access to the data capture you can break the capture into segments and on each segment compute the relative direction to the end node. Another way is to draw a line from the start node to the end node and consider the data above and below as a 1D function f(x). You could then rotate the straight line so that it is parallel to the x-axis as well as rotating the data points. Now you have a function f(x) that maps to a relative offset value y of a pixel. So you could fit a polynomial to the data and integrate to get the area under the curve or the total net offset from the straight line from the start node to the end node. This would give you a rough estimate since if the path goes equally far from the straight line on both sides then they will cancel out so it isn't perfect. You could instead make the function f(x) purely positive by negating any values below the line so they appear above and positive.

The last feature that is extracted is the LIL (loop index list). This is a parameter that I am still trying to improve but the goal is to provide a list of indices that match to data points where a loop is performed. This means the path circles around and crosses itself. These indices would be good positions to break the full path into sub paths and running parallel optimizations on each of the sub paths. At the end the intermediate end nodes can be combined with the start and end node to give the final optimized path. I don't actually use this feature in the current algorithm design due to its lack of being run in parallel but it is something that I would like to add in the future.

2.2. Algorithm Design

As mentioned earlier the algorithm is based in the simulated annealing algorithm and has 5 base functions the search space, goal function (energy), next state procedure, acceptance probability, and the annealing schedule, and I'll cover my implementations for each of the 5. Before going into the 5 base functions of the algorithm I'll go over some pre-processing of the graph network that was done to aid the algorithm.

An overview of the algorithm is shown below. The algorithm from a high level consists of looping through a fixed number of iterations, where on each iteration a probability parameter "p" is calculated that is based on the D2E and the current iteration number. This parameter is passed to a build path function that produces an estimate for the true path. Next the energy is computed for the estimated path using the true path info and if it is better than the current best then it is substituted for it. This continues for the number of iterations or until zero error is found, where it then exits. There are details left out like the piecing together of sub paths with current "locked" nodes and checking edge cases for constructing the path when near the end node.

Algorithm 1: Pseudo code for path minimization algorithm

```
Result: Minimum energy path
current node = start node;
best energy = inf;
best path = None;
for num iters do
    while current node != end node do
        current node = getNextNode();
        addNodeToPath(current node);
    end
    energy = getEnergy(path);
    if energy < best energy then
        best energy = energy;
        best path = path;
        path = None;
    else
        path = None;
    end
end
```

Graph Processing

Our initial search space state is a network graph of nodes that correspond to street intersections. To better facilitate the algorithm in finding the end node, I transform the edges of the graph network to have a corresponding direction vector attached to them. This direction vector is essentially a dot product of unit vectors where one is the unit vector pointing in the direction from the current node to the next node and the other is the unit vector pointing from the current node to the end node. The larger the value the more the edge points in the direction towards the end node. If the algorithm chooses a node with a large value then the next node in the path will be closer to the end node than all the other neighbors. You can think of this transformation as changing the node search space into a convex bowl with the end node at the global minimum of the function. Along with the previous change I added 'x' and 'y' coordinates in the UTM coordinate space to provide easier implementations with vector products and many other vector operations.

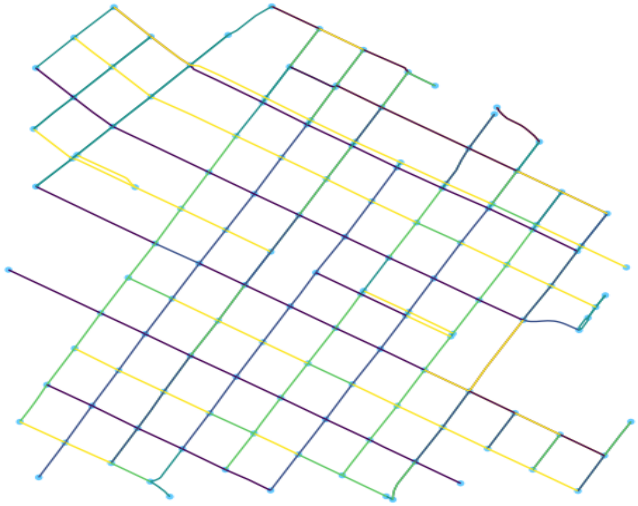
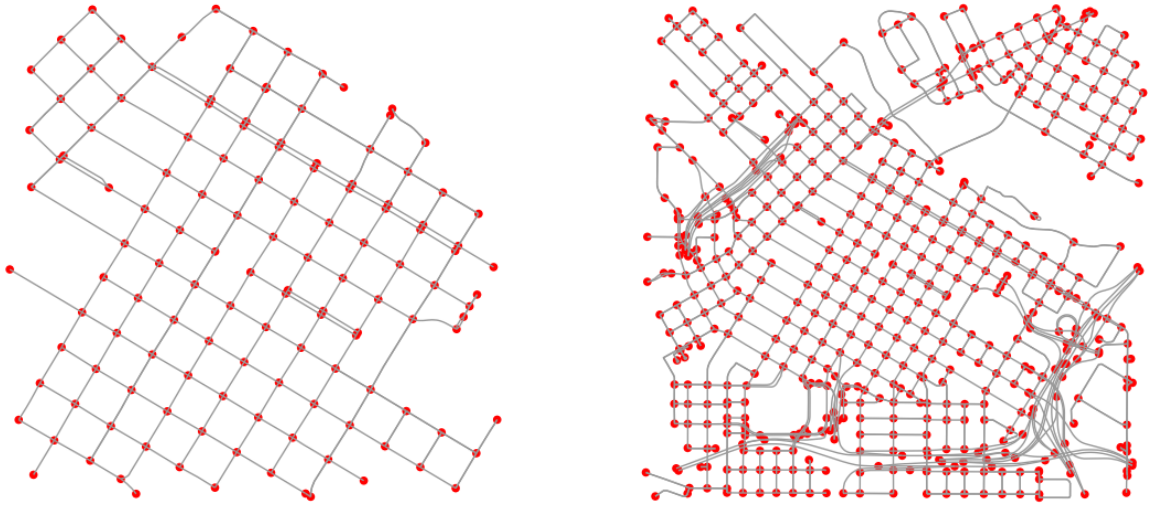


Figure 3: Plot of edges where color is proportional to the direction of edge relative to direction from start node to end node

Search Space

The search space for the optimization problem is determined by the density of the city or town you are considering and the size within that city you are considering. The density of street nodes is much higher in say London than some rural town so the search space would be much larger. The general rule of thumb is larger the search space the longer the run-time of the algorithm not considering the driven path etc. Figure 3 below show the same center longitude and latitude point but with a distance covered being twice as big for the picture on the right.



(a) search space with distance 750 has 132 nodes

(b) Search space with distance 1500 has 563 nodes

Figure 4: Comparison of number of nodes to distance considered

Energy Function

The energy function is the main driver of the algorithm since this is the function we wish to minimize for a built path. It takes 3 parameters the network graph, an estimate for the path as a sequence of nodes and the true path info that corresponds to the true path's feature parameters. It starts by computing the parameters that were mentioned in the "Feature Extraction" section so that it can use those for comparison with the true path info. The objective function is composed of two terms, one being a penalty for the estimated path containing more or fewer amount of turns in the path. This term effectively tries to bring the length of the path to approximately the same distance as the true path. The other penalty term is used to put higher penalties for getting the first "k" nodes wrong in the path estimation. This allows the algorithm to build path's sequentially, which will be discussed more in the "Annealing Schedule" section. Once the parameters are calculated the function loops over the turn-by-distance values of both the estimated path and the true path and compares the values. If the values differ by greater than some epsilon value that I define then it's penalized. The penalty is proportional to the index of the for loop since this would penalize wrong estimates for the first nodes more than later incorrect nodes.

Next State Procedure/Acceptance Probability

The next state procedure is the functionality that is in charge of determining the next state for the algorithm to choose to move to. In the classic simulated annealing algorithm the next state is chosen randomly from a distribution of neighboring states. I use a similar type of next state procedure except the "state" corresponds to the next network node to consider in the formation of a full path. So for example, if you are at a node with 4 neighboring nodes the next state procedure would return the node to move to next when building your full path. The next state procedure is parameterized by the D2E parameter as mentioned earlier. This parameter gives a probability on moving towards the end node so over the process of building a path if the D2E parameter is large the next state procedure would choose the node that is closer to the end node with a higher probability. This ensures the path is generally moving towards the end node, and if the D2E is low then the neighboring nodes have closer probability's to be chosen, thus a path that is less direct is chosen. The figure below shows two paths with their corresponding D2E parameters.

Annealing Schedule

The annealing schedule in the classic simulated annealing corresponds to the gradual reduction in the "temperature". This decrease in "temperature" in the algorithm has the effect of allowing the algorithm to search a broader space in the beginning and refining the search space more and more as the temperature decreases. I use this technique to fine tune the search space for the path. Initially the next state procedure will generate paths that vary considerably from path to path. Since the energy function is built to put more weight on the first k turns we would expect the path with the lowest energy after some number of iterations to have the first k turns correct. So I introduce a "locking" mechanism that after some amount of iterations the first "k" nodes of the path are locked into place and are part of the final path estimation. Now the search begins at a new start node which is the ending "k" node and is continued as before. This has the effect of building the path sequentially and reducing the search space in a way. It's important to note that I have to build a full path on each iteration since the features that I'm working with are normalized to a full path length.

3. Results

This section will cover the results I obtained for running the algorithm on varying paths and network distances. First off, the results are still not 100% consistent when scaling up to larger search spaces. I am still working on fixing the minor details that seem to make the algorithm get stuck in some places. Another thing that seems to be giving me issues is the handling of one way streets. My algorithm for building paths has edge cases where one-way streets cause it issues as well as culdesacs. If these types of nodes are avoided in the path building algorithm the regression performs fairly well and average size search spaces. The figures below show some of the random paths that were tested and the energy iterations when solving them.

```
[*] Generating test path...
[300] Current Min-e ==> 14 [33810354, 33545723, 33808647] 33295121
[*] Found Min
Min-E ==>  0
```

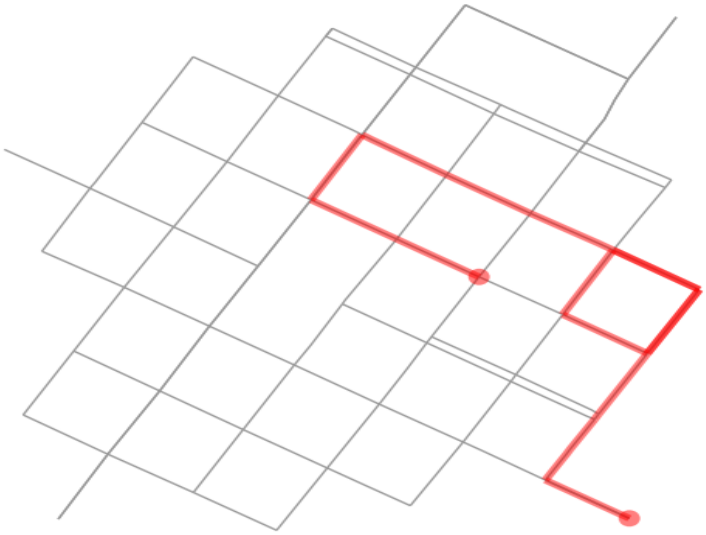


Figure 5: Regression ran on distance search space of 450. Algorithm found exact match in less than 600 iterations

```
In [9]: mine, path = regress(G, test_path_info, start_node, end_node, 3000, 3, 300)
Current Min-e ==>  35762 [582111787, 34559227, 33347842] 34644891
Current Min-e ==>  4352 [34644891, 33348808, 33348772] 1399967260
Current Min-e ==>  342 [1399967260, 33298139, 33808574] 33298030
[*] Found Min
```

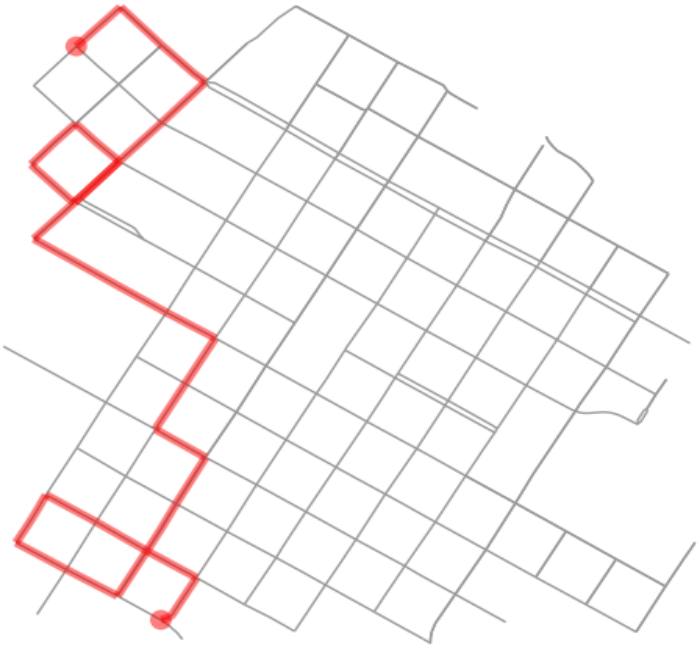


Figure 6: Regression ran on distance search space of 750. Algorithm found exact match in less than 1200 iterations

```
[*] Generating test path...
[300] Current Min-e ==> 66062 [33307500, 33307492, 33307493] 33307425
[600] Current Min-e ==> 16765 [33307425, 33386877, 33307507] 33307493
[900] Current Min-e ==> 6588 [33307493, 33307425, 33307438] 33373768
[1500] Current Min-e ==> 30 [33373768, 33495606, 33348808] 34644891
[*] Found Min
Min-E ==>  0
```

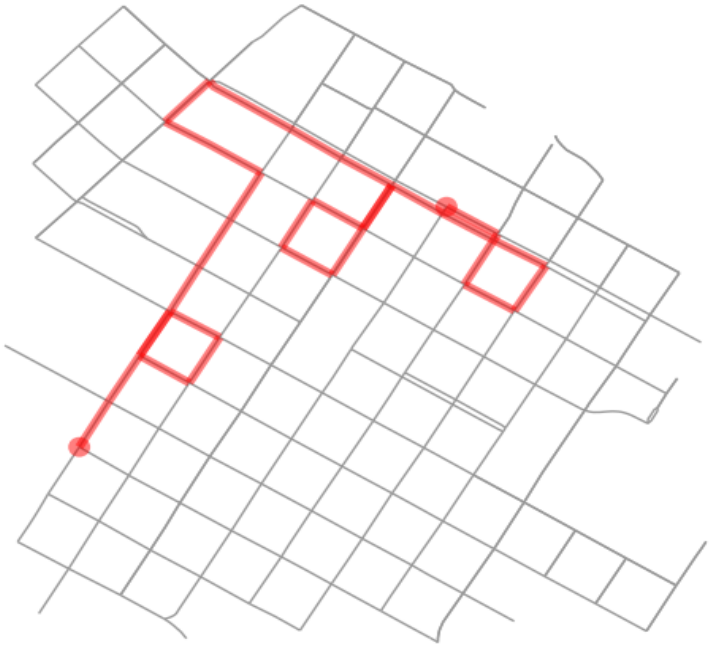


Figure 7: Regression ran on distance search space of 900. Algorithm found exact match in less than 1800 iterations

The previous plots and iterations results show that the algorithm performs fairly well and a average size node network. I had trouble testing on larger networks due to my algorithm getting stuck with one-way streets and culdesacs. Since the algorithm sequentially moves the next start node to the last sub path end node the algorithm converges exponentially faster when it gets closer to the end node. While the results are not showing the effectiveness on very large search spaces, these results are promising for improving on the algorithm to handle even large search spaces. As mentioned earlier, a modification to allow paralleling the optimization to a number of sub paths could help in optimizing over large search spaces and the results for this size search space could be added with other of the same size to produce a final result.

4. Future Work Discussion

Given that this is more of a proof of concept type project there is a lot of work that I'd like to do to improve the project. Firstly, I'd like to get my hands on some real capture data to see what other features I can obtain from working on the real data. This project was limited to using the street network nodes and properties and computer vision capabilities to obtain scale-invariant features. I believe one could use some methods from time-series analysis to determine when paths cross over each other or better estimates for the D2E parameter. While this improvement doesn't effect the algorithm design it does present better or more parameters for it to use to obtain better results.

The next improvement I would make is for the algorithm to handle edge cases for types of roadways. The algorithm as it sits now doesn't robustly handle one-way streets and culdesacs so I would improve on this detail so the algorithm does not get stuck in these areas when building new paths. As the search space continues to grow there may be other types of roadways that cause problems that will need to be addressed.

Currently one of the assumptions of the algorithm is that it has the start and end node of the path, but the goal would be to give the algorithm a confidence area for the both nodes and have it figure out what is the most probable start and end node. To solve this problem I thought it could be useful to use a different type of search algorithm like particle swarm optimization. This optimization initializes a swarm of state objects and

calculates energies for each, and coordinates between the objects to determine there next move. This technique may provide benefits over simulated annealing for the problem of determining start and end nodes.

Another modification or improvement that I would like to add is to the energy function. Obviously if I can obtain more important features from the data than those can be included in the energy function. Currently I use basic exponential weightings for errors but could there be other types of functions that I could to better accommodate large paths like normalized values? Also I could look at instead of matching exact start and end nodes I could compute losses over distribution of neighbors so the energy function would need to account for differences between distributions like the KL divergence. There are many discrepancies that can occur between the estimated path and the true path and the energy function needs to represent these appropriately.

The update stage of the algorithm currently locks "k" number of nodes of the best energy path after every "n" number of iterations. Another thing I'd like to improve is the number of "k" locked nodes being based on the smallest energy found during the iteration stage. This solves the problem of what if during iterating we build a path that is almost exactly like the true path? Right now that isn't taken into account only "k" number of nodes are locked. In that scenario we need to lock more nodes since the path is almost identical to the true path. Thus a locking parameter that is inversely proportional to the energy would be advantageous in terms of efficiency and reaching convergence faster.

Lastly to better scale to large search spaces and complicated paths I want to add the functionality to parallel the optimization. Points along the path could be chosen to break the path into sub paths and the optimization is done all of the sub paths. Using the confidence value technique for start and end nodes the sub paths could be pieced together to given the final estimated path.

References

- O., F. (2008). Structural Optimization Using Simulated Annealing. Simulated Annealing. doi: 10.5772/55
- Simulated annealing. (2019, December 16). Retrieved from https://en.wikipedia.org/wiki/Simulated_annea

Appendix A.

To run the program you need the libraries referenced in `node_regress.py` *installed. Once you have those installed you can run the program by issuing the command* `python node_regress.py`*. This will generate a random path in Minneapolis and then try to best fit it based on it's features.*

```
epsilon = .005
n_iters = max(len(turn_info_est),len(turn_info_true))
obj_fun = 0

for i in range(n_iters):
    if i > (len(turn_info_true)-1) or i > (len(turn_info_est)-1):
        obj_fun += (n_iters - i)**2
    elif abs(turn_info_est[i][1] - turn_info_true[i][1]) > epsilon:
        obj_fun += (n_iters - i)**4
```

Figure A.8: Snippet from energy calculation

```
p = (.95 -  .35*(n_iters - i)/n_iters)
path = build_path(net,start_node_,end_node,p)
full_path = node_list + path
e = energy(net,full_path,true)
```

Figure A.9: Snippet from main loop that calculates probability and builds estimated path