CS182 Artificial Intelligence Project
Safe Cycling Route Planning

Anna Sophie Hilgard, Nick Hoernle, and Nikhila Ravi

December 11, 2016

1 Introduction

Biker safety and biker safety awareness has been a topical issue for a number of years with the biker death toll at an unacceptably high rate (13 deaths in the past 5 years in Boston alone and 876 recorded accidents involving bikes and cars in Cambridge in the past 5 years. Current mapping applications like Google maps often route cyclists through busy streets and intersections, resulting in unnecessarily dangerous trips. The purpose of this project has been to explore AI graph search and local search algorithms to optimize a bike routing algorithm not only for 'distance minimization', but also for 'path safety' and 'elevation comfort'. We explore the use of A^* graph search under different cost and heuristic metrics to obtain a fast, yet reliable graph search between any two nodes on the intersection graph. Furthermore, we explore the use of local search to optimize for the ideal meeting point between two or more cyclists.

2 Background and Related Work

While our problem was fairly straightforward, we did do some research to get ideas for potential cost attributes and data sources. Our initial formulation of the problem, for example, did not include elevation differencing, but reading previous studies led us to believe that this was an important criterion [1].

3 Problem Specification

The graph of intersections and nodes forms a highly connected state space (Cambridge has 1831 nodes, San Francisco has 18410 nodes, and each node typically has a branching factor of 4). While it is possible, searching the entire state space for San Francisco would involve considerable computation, on the order of 4^{18410} possible routes to explore, and this search space grows exponentially with additional nodes.

To compound this search space problem, we aimed to determine central meeting areas for a number of cyclists to use as a fair and safe meeting place. The search problem here is therefore not only to determine the best route from a start node to a goal but also to determine the best goal node, optimizing for minimal cost for all cyclists.

Initially, we implement an A^* search algorithm. We then used that A^* search as a baseline for determining the route from any starting point to some goal node and we run a local search layer on top of this to find the best meeting intersection between a number of cyclists. We tailored three cost functions and three heuristics for comparing the performance of the A^* and Local Search (which depend on the A^* algorithm).

3.1 Cost Functions

We seek to minimize some cost function:

$$cost = \sum_{i=0}^{n} cost(path_i) \tag{1}$$

given that:

- $path_0 \in Connections_{\text{(starting node)}}$
- $path_i, path_{i+1} \in Connections_{node_i}$
- $path_n \in Connections_{\text{(target node - 1)}}$

Specifically for our problem, we created three different cost functions:

$$cost_{distance\_only}(path_i) = length_i \tag{2}$$

$$cost_{distance\_safety}(path_i) = (\alpha \times length_i) \times (\beta \times accidents_i) \tag{3}$$

$$cost_{distance\_safety\_elevation}(path_i) = (\alpha \times length_i) \times (\beta \times accidents_i) + abs(\Delta elevation_i) \tag{4}$$

Where α and β are scaling multipliers to weight the relative magnitudes of the different cost measurements. We used a value of $\alpha = 10000$ to scale the coordinate based length measurement to a number on the same magnitude of $\Delta elevation$. β was set to 5 to penalize the roads with accidents highly in the algorithm. These values can be tuned empirically.

In the optimization portion of our project, we seek to minimize the total cost to all parties:

$$min(\sum_{j}^{n} cost_j) \tag{5}$$

where $cost_j$ is the cost to participant j.

4 Approach

4.1 Data Structures

Our primary data structures are an intersection graph, which is stored as a Python dictionary, and a connection dictionary, which is also stored as a Python dictionary. The intersection graph maps nodes (intersections) by id to a list of paths (road segments) from that node. The connection dictionary maps a connection (road segment) to its source node, sink node, and various cost parameters (in our case distance, number of bicycle crashes on that road segment, and change in elevation over that road segment).

4.2 Data Collection, Extraction and Preprocessing

In this case study, we were able to use open source geolocation data from Cambridge and San Francisco city councils. These departments further supplied data on the number of bike related accidents for various road segments. Lastly an elevation layer from the open GIS websites was used to infer the elevation of certain roads and intersections.

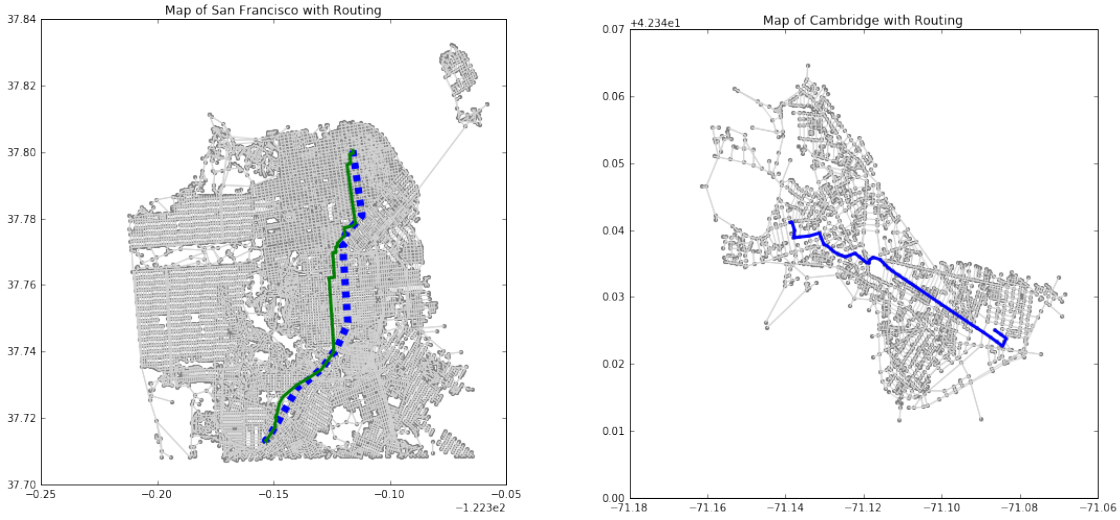
We tested our implementation on data from the cities of Cambridge, MA and San Francisco, CA. GIS location data is available on the local government websites in the form of a pandas geojson dataframe and was easily read into a pandas dataframe object using the geopandas library [1].

As discussed above, these geolocation data are used to create a set of nodes with coordinate positions and a number of connections which define the roads and the intersections that those roads are connected to. The data from Cambridge contained routing errors where some intersections were connected to other incorrect intersections resulting in roads that spanned the entire graph rather than simply connecting the two nearest neighboring intersections. The solution to this was to use the *geometry* data within the Pandas geolocation dataframe and the *shapely* [2] graphing library to compare the actual road length to our interpolated distance for the road. If the interpolated distance was incorrect by more than a factor of 10, we made the assumption that the nodes were incorrectly tagged in the data and we therefore dropped the connection attribute. The elevation and crash data was independently collected from the different government websites and the intersection id's were used to map this data into the 'intersection' and 'connection' graphs and dictionaries.

The resulting connected graphs for 'Cambridge' and 'San Francisco' are shown in figure 1

[1] `http://geopandas.org/`

[2] https://pypi.python.org/pypi/Shapely



(a) San Francisco

(b) Cambridge

Figure 1: Connected graphs of the San Francisco and Cambridge maps with a route shown on the map (using only distance as the cost metric on the Cambridge but distance and safety on the San Francisco map).

4.3 A^* Search

A^* is an adaptive graph search algorithm that is able to take different cost and heuristic functions as parameters and then traverses the graph based on the output of these functions. A^* search was chosen over the uniform cost alternative due to the efficient graph search under a good heuristic. We note that while a simple heuristic is to use the Euclidean distance to the goal node, incorporating accident and elevation data is non-trivial and may lead to inconsistent heuristic functions.

Please refer to Appendix C, A^* Search Algorithm for a formal outline of the algorithm.

4.3.1 Heuristic Functions

We used three main heuristic functions for testing the A^* algorithm. We firstly used a null heuristic that we use as a baseline to compare the other heuristics against. We used a simple euclidean distance heuristic that measures the distance from the current node i to the goal node as an admissible and consistent heuristic. Finally we used a heuristic that makes a simple delta elevation estimate from the current goal to the end goal and a minimum of the number of accidents on the connections from the current goal. This 'combined heuristic' also included the euclidean distance

element and a linear combination of these values was used.

$$heuristic_{null}(node_i) = 0 \tag{6}$$

$$heuristic_{euclidean\_distance}(node_i) = euclidean\_distance(node_i, goal) \tag{7}$$

$$\begin{aligned} heuristic_{combined}(node_i) = &(\alpha \times euclidean\_distance(node_i, goal) \\ &\times (\beta \times min(accidents\_node_i)) \\ &+ abs(elevation\_difference(node_i, goal)) \end{aligned} \tag{8}$$

4.4 Local Search: Simulated Annealing

The Simulated Annealing Search involved initializing a centroid and iteratively hill climbing the space around that centroid to find a local optimum point. Initially, a temperature attribute is set high, such that the hill climbing algorithm accepts non-optimal nodes with a high probability. As the algorithm proceeds, and converges on a local optimum, the temperature is decreased such that the hill climbing becomes more deterministic.

As the state space of the search graph is large, and the distance metric is highly interpretable, we opted to initialize the first centroid at the intersection that represents the Euclidean mean between all cyclists. Our aim was to initialize the centroid within the vicinity of an optimal meeting point and allow the high initial temperature to counteract any potential local optimum intersections that may have been encountered.

Please refer to Appendix C, Simulated Annealing Algorithm for a formal outline of the algorithm.

4.5 Local Search: K-Beam Search

K Beam search involved initializing a number of centroids within the state space of the graph and iteratively hill climbing from the K best centroids to find a local optimum.

Please refer to Appendix C, K Beam Search Algorithm for a formal outline of the algorithm.

5 Experiments

We aimed to test:

- The effect of the varying cost functions on the routes that were found.
- The effect of the A^* heuristic on the speed of search (and consistency of the route)
- The efficiency and effectiveness of the resulting search

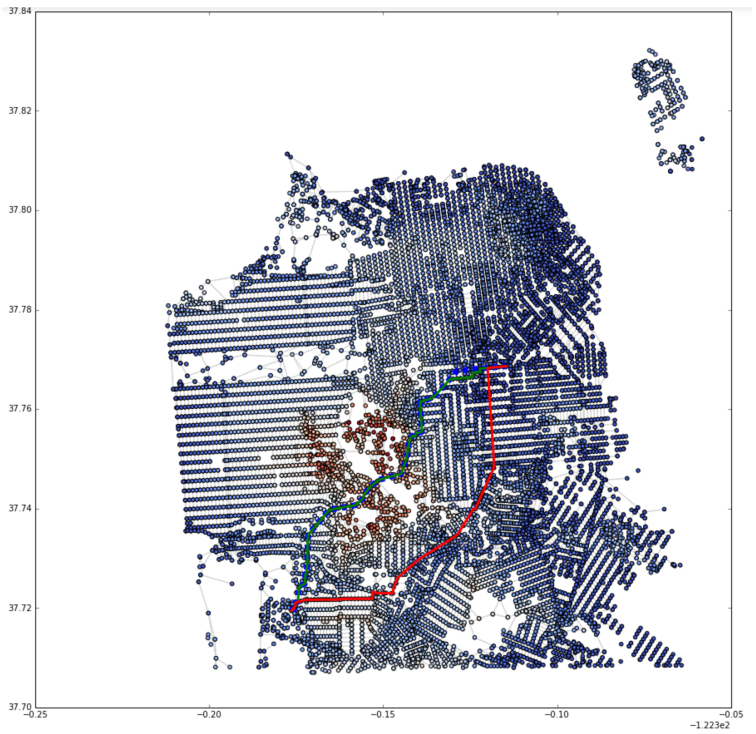


Figure 2: Paths between nodes in San Francisco. Blue path optimizes for distance only, green path avoids previous bicycle accidents, and red path minimizes altitude changes. Overlaid with an altitude plot of San Francisco, we can clearly see the red path avoiding a hill.

5.1 Testing A^* Search

We randomly selected nodes within the two graphs and ran A^* search to find the optimal route through the map. The figure 2 shows an example of A^* finding an optimal route under two different costs.

We then ran an iteration of 100 simulations for each cost function (2) and each heuristic function (6). Specifically the null and euclidean heuristics are both expected to find the optimal route. The euclidean heuristic should explore fewer nodes than the null heuristic. When the cost function is simple distance or simple distance and safety, the combined heuristic is neither admissible nor consistent as it is penalizing nodes for a cost that is not encoded in the algorithm. We expect to see this algorithm find a 'longer than necessary optimal path'. However, when the cost function also encodes all of these costs, we now expect the combined heuristic to out perform the other heuristics on all accounts. Please refer to Results for a further discussion on the above.

5.2 Testing Local Search

We ran Simulated Annealing and K-Beam Search on a number of different iterations of randomly selected nodes. While both algorithms can be tuned to run faster (by reducing the temperature faster for simulated annealing and reducing 'k' for k-beam search at the cost of being more susceptible to local minima) we can make a reasonable comparison of the two algorithms in terms of their total runtime and resulting cost of the returned node.

6 Results

6.1 A^* Search Results

Initially we ran a test of the different cost functions using a null heuristic. We are thus guaranteed to find the optimal solution for each of the cost functions (this is analogous to uniform cost search) but this search is inefficient. For this experiment, we are interested in comparing the cost functions and thus this is not important.

Figure 3 shows the route distance, number of nodes searched and the total change in elevation of the route vs the euclidean distance from source node to goal node for different routes, changing the cost function that is used in the A^* search. Here we see that the basic road cost will find the shortest possible route to the solution, but as additional costs are added to certain road segments the distance of the other solutions will increase. Interestingly, the three cost functions often require the same number of nodes to be searched. Lastly, we see that the combined cost function generally results in a solution that has the lowest total change in elevation (the uncommon event when it does not result in the lowest elevation is because it is also optimizing over the distance and the number of accidents on each road link).

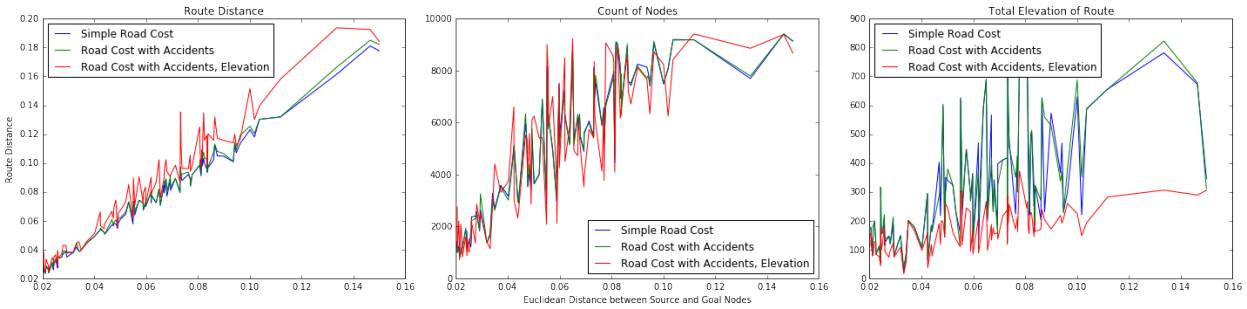
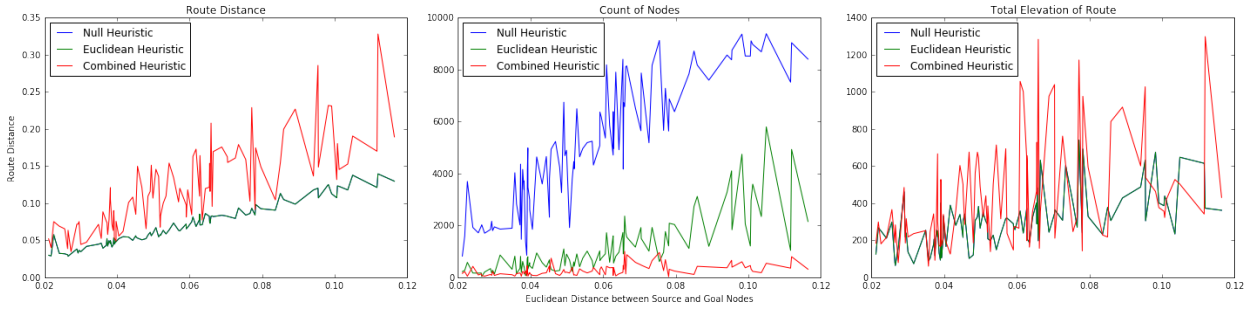


Figure 3: Basic metrics for the three cost functions.

In Figure 4 we see that the combined heuristic significantly under performs the null heuristic and euclidean heuristic in terms of distance as the heuristic does not properly model the cost function (it is adding expected costs for each node that do not actually exist). This heuristic is

therefore not admissible for these cost functions and it does not return the optimal path to the solution. The euclidean heuristic still finds the optimal path but it expands far fewer nodes than the null heuristic.

Figure 4: Varying the three heuristics for the basic road distance cost.



Again, figure 5 shows that the combined heuristic is not admissible. The euclidean heuristic again far outperforms the null heuristic in terms of nodes searched.

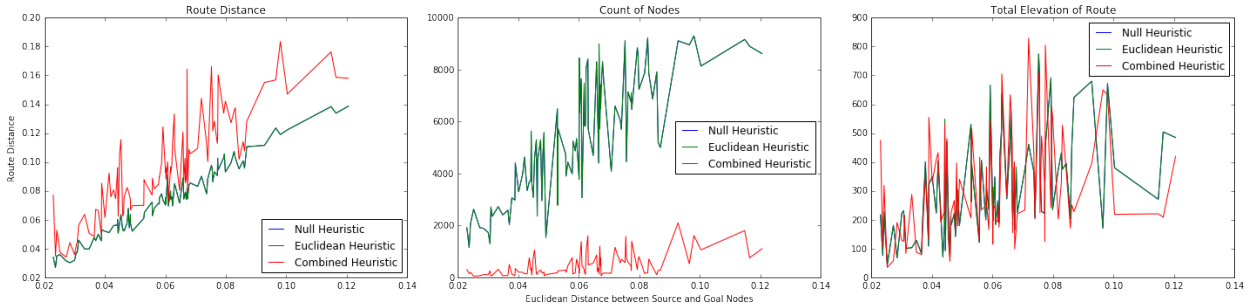
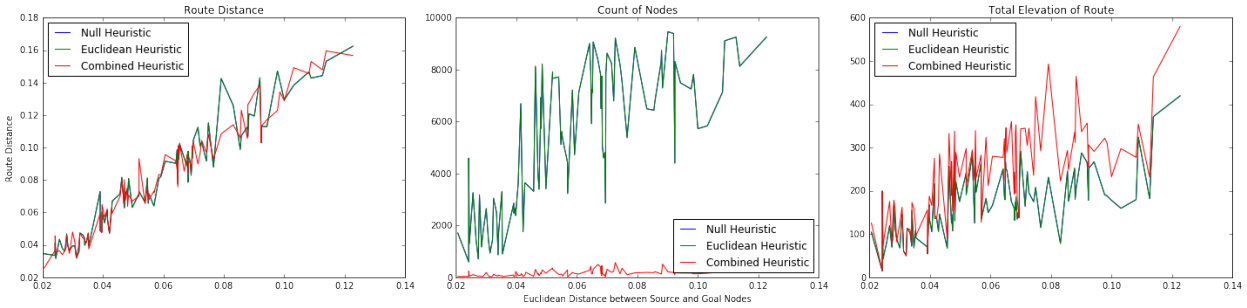


Figure 5: Safety road cost under three different heuristics.

Finally, we see in 6 that the the combined heuristic returns a path that is comparable to the other two heuristics (in terms of path distance). The fact that it is different suggests that the heuristic might not be admissible as the number of accidents from one node to the next may be dramatically different from what the heuristic is guessing (this is a very hard parameter to model). However, the fact that this heuristic now does model the accident and elevation costs that are present means that it is very efficient in finding an optimal route (in terms of numbers of nodes searched).

Figure 6: Safety, distance, and elevation cost under three different heuristics.



6.2 Local Search Results

As for the A^* search evaluation, we randomly selected 4 nodes and computed the time that it took the algorithm to converge and the overall cost of all routes to the final returned centroid location. In figure 7 we see that k beam generally finds a solution centroid with a slightly lower cost than simulated annealing. Furthermore, from 8, apart from under the null heuristic, we see that k beam search with $k = 5$ has a very comparable search time to that of simulated annealing.

Figure 7: Time to compute the centroid node for both Simulated Annealing and K Beam search (with $k = 5$)

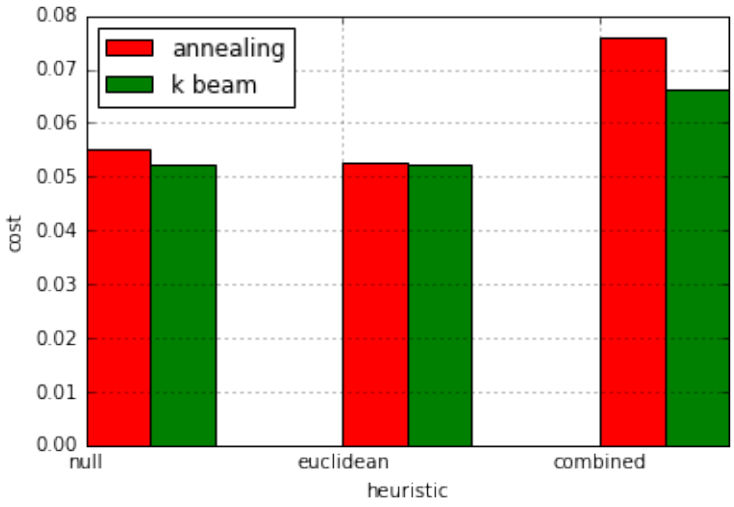
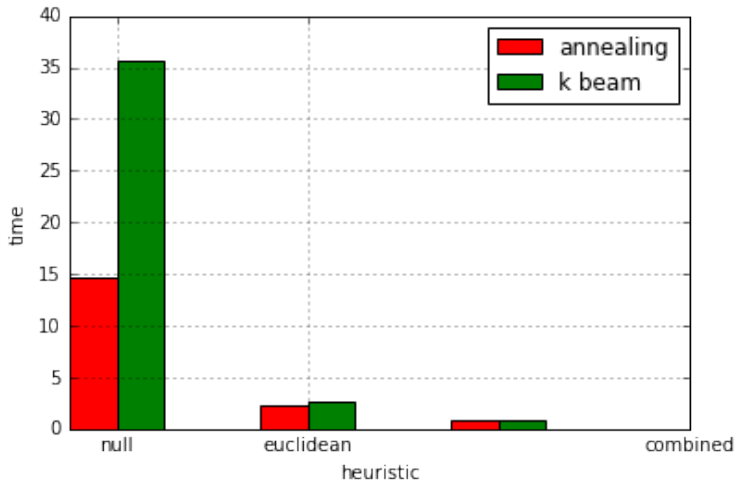


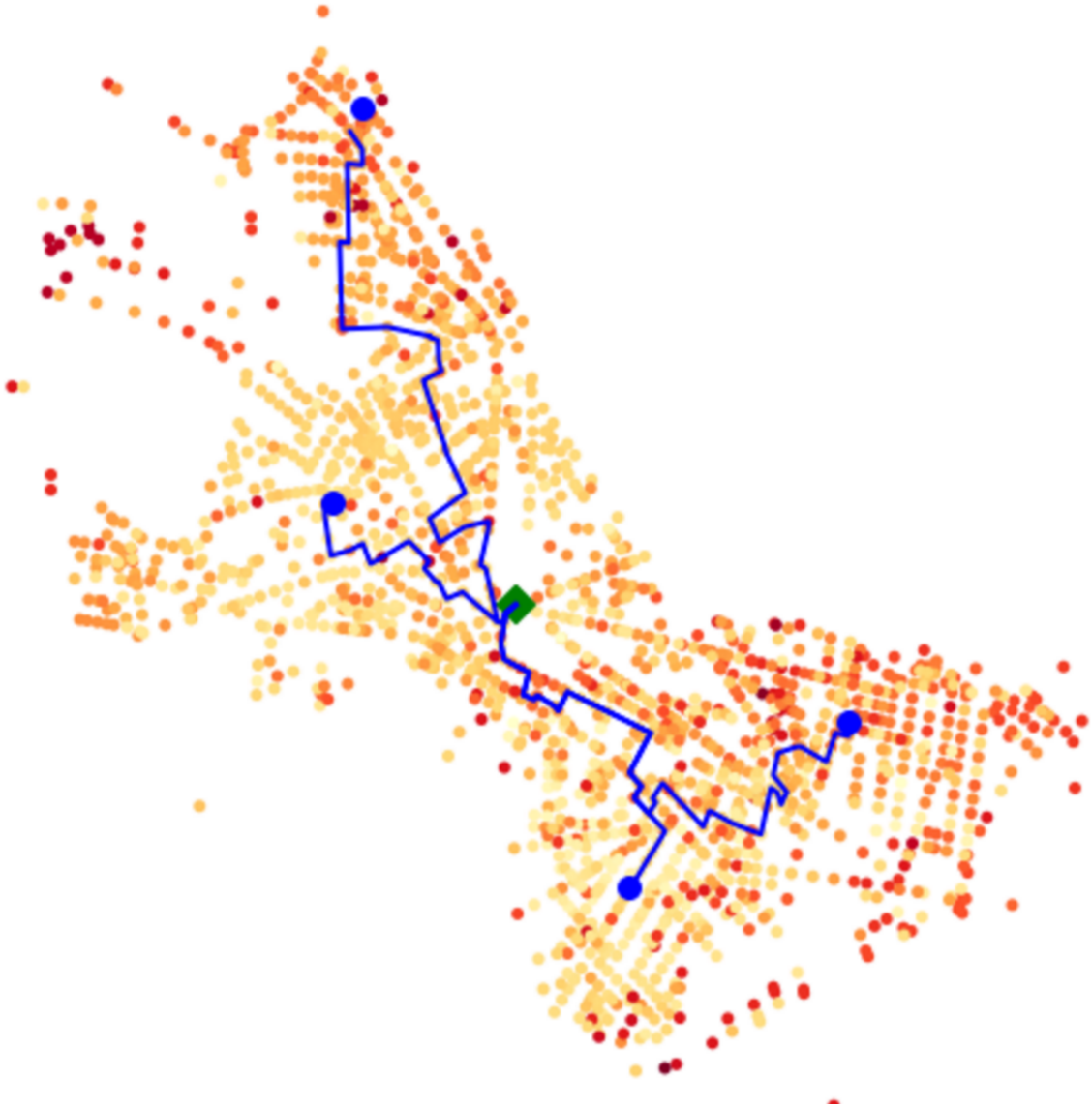
Figure 8: Total cost of routes from all starting nodes to the returned centroid for both Simulated Annealing and K Beam search (with $k = 5$)



The reason for k beam search out performing simulated annealing is the presence of very definite local optima. The initialization of multiple random start points within a reasonable area of possibilities has a greater chance of initializing close to the solution than the simulated annealing. Even with the temperature set at a high value, the simulated annealing struggles to fully leave the vicinity of the node that it is initialized at (and since this is deterministically the absolute center of all the cyclists, especially if the route is hilly or has many road accidents, it may be far from the global optimum point).

While the k beam solution does out perform the simulated annealing solution, in practice we do see that the costs are still comparable and thus the solution that the simulated annealing returns is still valid. The below figure 9 is a plot showing the centroid found between four cyclists (under the combined cost cost function and combined heuristic).

Figure 9: Plot showing the centroid found between four cyclists in Cambridge, and each cyclist's route to the solution. The plot encodes the cost of every other node in the map as if it were the solution (and thus shows the high cost nodes in red - i.e. unlikely centroids) and the low cost nodes in yellow (i.e. likely centroids). We further see from this plot that while the combined cost for the four cyclists is not necessarily the same as the cost of only one cyclist, the route that the A algorithm finds generally follows a path that avoids the red high cost nodes.*



7 Discussion

Our algorithms could have been made to run somewhat faster by incorporating pruning strategies but we were generally satisfied with run times (especially when evaluating the combined heuristic with the distance, safety and elevation costs). Additionally, it was difficult to generate a very useful admissible heuristic for bike crashes because one could almost always find an extremely convoluted path to avoid almost all of them, and so to have a heuristic which is always $\leq$ the actual cost to the goal is often not consistent. However, even without being consistent (i.e. it was returning a solution that was slightly different from the optimal solution), the heuristic did provide a very good solution for the problem.

Furthermore, calculating the number of accidents per road segment is a good estimate of how dangerous a given road segment is, but probably a better indicator would be accidents per units of bicycle traffic. If we were able to collect bicycle traffic data for each road segment, we could then calculate the Bayesian probability of:

$$P(\text{road segment} \text{ — } \text{accident}) \propto P(\text{accident} \text{ — } \text{road segment}) * P(\text{road segment}) \tag{9}$$

In particular, we may be over-penalizing roads with a large amount of cycling traffic.

The graph search approach shows a very feasible solution to this problem of intelligent routing. We felt that our results were very reasonable given our preexisting knowledge of Cambridge and San Francisco streets.

Takeaways from this project:

- You're only as good as your data. We had some lofty goals for the data that we'd be able to collect and use for this project, but even the data we ended up using was harder to collect and map than we had originally anticipated. Furthermore, while still applicable to testing these algorithm implementations, much of the crash data was a few years old and thus to be more applicable we would want to include a real time update to the data and an expiration of old data. Cambridge and San Francisco are relatively tech-forward cities, so we can imagine this would be even more difficult in most other environments.
- We initially built the system for the Cambridge data but then obtained the larger mapping data for San Francisco. It was therefore interesting to see how our algorithm scaled to the larger map of San Francisco. To plan routes through a larger area, it's clear that we would have to adapt the algorithm to get the runtime within a reasonable range. In particular, we found in our reading that many routing engines actually use inadmissible heuristics for these tasks and still find reasonable results but in a much quicker time.

A System Description

The easiest way to use our system is to open the provided iPython notebook and run each of the cells. All of the supporting functions can be found in $final\_project.py$.[3] [4]

B Group Makeup

1. Nicholas Hoernle
   (a) Creation of graph dictionary structure and A^*search algorithm
   (b) Simulated annealing
2. Nikhila Ravi
   (a) K-Beam Search
   (b) Visualization and analysis of results of graph search algorithms
3. Anna Sophie Hilgard
   (a) Construction of Datasets
   (b) Research and Implementation of more complicated cost functions and heuristics

[3] *final_project.py*: https://github.com/NickHoernle/Artificial-Intelligence-CS182-Project/blob/master/final_project.py

[4] *final_project.ipynb*: https://github.com/NickHoernle/Artificial-Intelligence-CS182-Project/blob/master/final_project.ipynb

C Algorithms

Algorithm 2 A-Star Search

```
function A-STAR-SEARCH(graph, startnode, targetnode)
    node ←a node with STATE = startnode
    PATH-COST ← heuristic(startnode, targetnode)
    frontier ← a priority queue ordered by PATH-COST + heuristic(node, targetnode) with node
as the only element
    explored ← an empty set
    loop
        if EMPTY?(frontier) then
            return failure
        end if
        node ← POP(frontier) /*chooses the lowest cost+heuristic node in frontier */
        if node == targetnode then
            return SOLUTION(node)
        end if
        add node.STATE to explored
        for each path in PATHS(node) do
            child ← child-node(node, path)
            if child.STATE is not in explored or frontier then
                frontier ← insert(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST + heuristic then
                replace that frontier node with child
            end if
        end for
    end loop
end function
```

Algorithm 3 Simulated Annealing Algorithm

function SIMULATED ANNEALING MEETING SPOT($graph, startingpts, cost, heuristic$)
  if length$(startingpts) < 2$ **then**
    return error
  end if
  $current \leftarrow$ **mean**$(startingpts)$.CLOSEST-NODE
  $temperature \leftarrow e^{10}$
  $\gamma \leftarrow .5$ /*schedule to manage $temperature$ */
  while $temperature > e^{-2}$ **do**
    $temperature \leftarrow temperature * \gamma$
    $next \leftarrow$ a randomly selected *child* of *current*
    $current$.VALUE $\leftarrow \sum_{pt \in startingpts}$ **cost**$(pt, centroid)$
    $next$.VALUE $\leftarrow \sum_{pt \in startingpts}$ **cost**$(pt, next)$
    $\Delta E \leftarrow next$.VALUE $-$ $current$.VALUE
    if $\Delta E > 0$ **then**
      $current \leftarrow next$
    else
      $current \leftarrow next$ with probability $e^{\Delta E / temperature}$
    end if
  end while
end function

Algorithm 4 K Beam Search Algorithm

function K-BEAM SEARCH MEETING SPOT($k, graph, startingpts, cost, heuristic$)

if length$(startingpts) < 2$ **then**

return error

end if

$\{candidatenodes\} \leftarrow node \,\forall\, node \in graph$ s.t.
$node.x \geq min(startingpts.x) \& node.x \leq max(startingpts.x) \&$
$node.y \geq min(startingpts.y) \& node.y \leq max(startingpts.y)$

$point_i \leftarrow$ a randomly selected $node \in \{candidatenodes\} \forall i \leq k$

$best$.VALUE $\leftarrow min_{i \leq k}(\sum_{pt \in startingpts}$ **cost**$(pt, point_i))$

while True do

$\{nextcosts\} \leftarrow \sum_{pt \in startingpts}$ **cost**$(pt, child_i) \forall i \leq k, child_i \in$ PATHS$(point_i).endnode$

$point_i \leftarrow$ i-th least $node \in \{nextcosts\} \forall i \leq k$

$next$.VALUE $\leftarrow \sum_{pt \in startingpts}$ **cost**$(pt, point_1)$

if $next$.VALUE $<$ $best$.VALUE **then**

$best \leftarrow next$

else

break

end if

return $best$

end while

end function

References

[1] Jan Hrncir, Palov Zilecky, Qing Song, and Michal Jakob. Practical multicriteria urban bicycle routing. *IEEE Transactions on Intelligent Transportation Systems*, PP(99):1–12, 2016.