CS182 Artificial Intelligence Project
Safety Optimised Bike Routing

Anna Sophie Hilgard, Nick Hoernle, and Nikhila Ravi

December 12, 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 danger during 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 '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 an intersection graph. Furthermore, we explore the use of local search algorithms to optimize for a central point between two or more cyclists.

2 Background and Related Work

While the problem we were trying to solve was fairly straightforward, we conducted some preliminary research to determine potential cost parameters and data sources. Previous studies in the London had explored the use of bike accident data to optimise routes for safety, and this was the starting point for our project. Our initial formulation did not include elevation differencing, but previous studies suggested that elevation is a key factor affecting riding comfort for cyclists, and elevation data was later incorporated into our cost functions. [**?**].

3 Problem Specification

The map of a city represented as a 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). Searching the entire state space for San Francisco, while possible, would

involve considerable computational expense. There are on the order of 4^{18410} possible routes to explore, and this search space grows exponentially with additional nodes.

To extend the problem, we tackled the task of determining central meeting areas for a number of cyclists. This search problem involves not only determining the best route from a start node to a goal, but actually determining what the best goal node is while optimizing for minimal cost for all cyclists.

Initially, we implemented an A^* search algorithm for finding the optimal route between two specific notes. We used this A^* search algorithm as a baseline, running a local search algorithm on top to find the best meeting location between two or more cyclists. We devised three cost functions and three heuristics to use with A^* and Local Search (which depend on the A^* algorithm) and compared the performance of each.

4 Approach

4.1 Data Structures

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

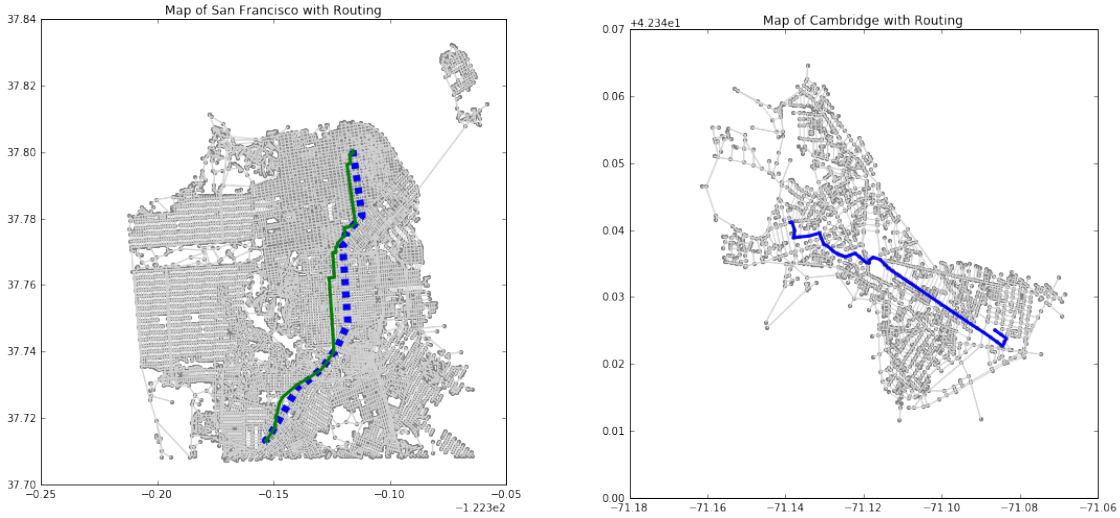
4.2 Data Collection, Extraction and Preprocessing

We were able to use open source geolocation data from Cambridge and San Francisco city councils. We were also able to obtain data on the number of accidents for various road segments from which we extracted the number of bike related accidents. Lastly an elevation layer from the open GIS websites was used to infer the elevation of certain roads and intersections.

We tested the implementation of our graph structure and search algorithms on data from the cities of Cambridge, MA and San Francisco, CA. GIS location data was 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].

The geolocation data were used to create a set of nodes with coordinate positions, and a set 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 incorrectly connected to other intersections, resulting in roads that spanned the entire graph rather than simply

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



(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).

connecting the two nearest neighboring intersections. Our solution 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 the connection attribute was dropped. The elevation and crash data were 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

4.3 A^* Search

A^* is an adaptive graph search algorithm that can be use to find routes based on different cost and heuristic functions passed in as parameters to the algorithm. A^* search was chosen over the uniform cost alternative due to the possibility of a more efficient graph search under a good heuristic. Initially we started with a simple heuristic, the Euclidean distance to the goal node. We were careful to note that developing more sophisticated heuristics involving accident and elevation data is non-trivial, and may lead to inconsistent heuristic functions.

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

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

4.3.1 Cost Functions

The goal of A^* search is to find a path which minimizes 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 which relate to the cost attributes, distance, safety, and elevation:

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

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

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

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 roads with accidents by increasing their apparent length . These values can be tuned empirically.

In the optimization section of the 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. Alternatively we could also consider minimising the difference between the maximum and minimum cost to the riders in the group.

4.3.2 Heuristic Functions

We used three main heuristic functions for testing the performance of the A^* algorithm. Our baseline was a null heuristic. We first tried a simple admissible and consistent heuristic based on the euclidean distance heuristic from the current node to the goal node. As a more sophisticated heuristic, we created a function which is a linear combination of the estimated change in elevation from the current node to the goal, and a minimum of the number of accidents on the connections (road segments) from the current node. This 'combined heuristic' also included the euclidean distance estimate.

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

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

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

4.4 Local Search: Simulated Annealing

The simulated annealing search involved initializing a centroid and iteratively hill climbing the space around the centroid to minimize the cost function to find a local optimum point. The temperature attribute is initialised at a high value 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 cost minimisation 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 **??**, **??** for a formal outline of the algorithm.

4.5 Local Search: K-Beam Search

K Beam search involved first initializing a number of centroids within the state space of the graph. The centroids were initialised in a region of the graph bounded by the starting points. All the successors of these centroids were generated, and the k best centroids based on the cost function were chosen as the successor centroids. This iterative hill climbing from the k best centroids was used to find a local optimum. Different values of the beam width 'k', were tested to see the effect on the optimum cost and running time of the algorithm.

Please refer to Appendix **??**, **??** for a formal outline of the algorithm.

5 Experiments

We carried out a number of experiments to test the performance of our A* algorithm and various parameters. We aimed to test:

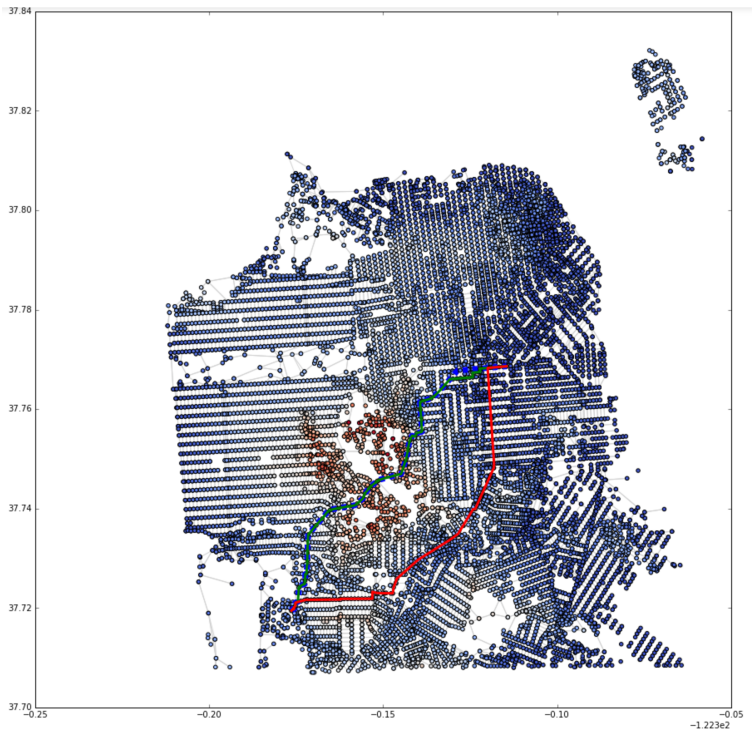


Figure 2: Paths between nodes in San Francisco. Blue path optimizes for distance only, green path avoids previous bicycle accidents, and red path minimizes altitude changes. Overlaid with an altitude plot of San Francisco, we can clearly see the red path avoiding a hill.

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

5.1 Testing A^* Search

We randomly selected nodes within the two graphs and ran A^* search to find the optimal route. The figure 2 shows an example of A^* finding an optimal route under two different costs with the safety and comfort optimised route returning a path which avoids area of high elevation.

We then ran 100 simulations for each cost function (2) and each heuristic function (6).

We expected the null and euclidean heuristics to find the optimal route, with the euclidean heuristic exploring 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 being used by the algorithm. With this heuristic we expect the returned optimum path to be 'longer than necessary'. However, when a combined cost function is

used which accounts for distance, safety and elevation, the combined heuristic is expected 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), this search is inefficient. For this experiment, we are interested in comparing the cost functions and thus this is not important.

Figure **??** shows the route distance, number of nodes expanded and the total change in elevation of the route for different cost functions that are used in the A^* search. Here we see that the basic road cost will find the shortest possible route to the goal, but as additional costs are added to certain road segments, the distance of the paths returned by the other cost functions 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).

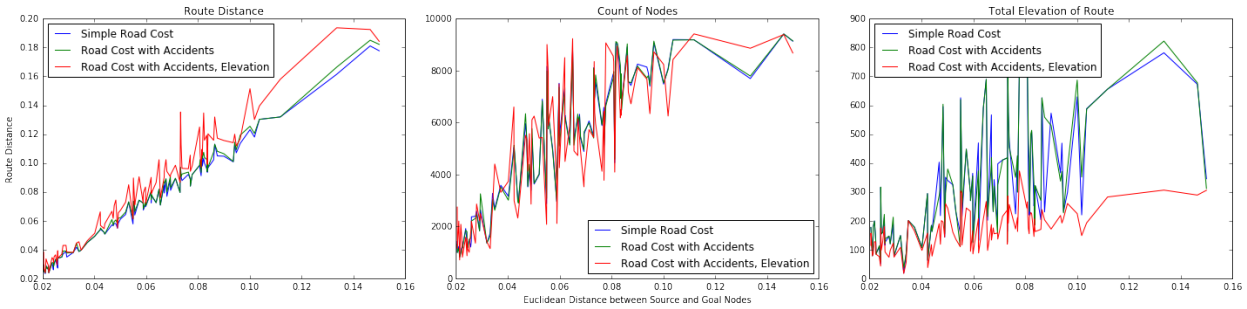


Figure 3: Basic metrics for the three cost functions with the null heuristic

Figure **??** shows a comparison of the different heuristic functions under the basic road cost function. We see that the combined heuristic significantly under performs compared to the null heuristic and euclidean heuristic in terms of distance, as the combined heuristic does not properly model the cost function (it is adding expected costs for each node that are not accounted for by the basic road cost function). This heuristic is therefore not admissible for this cost function and it does not return the optimal path to the solution. The euclidean heuristic is still seen to find the optimal path and as expected, it expands far fewer nodes than the null heuristic.

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

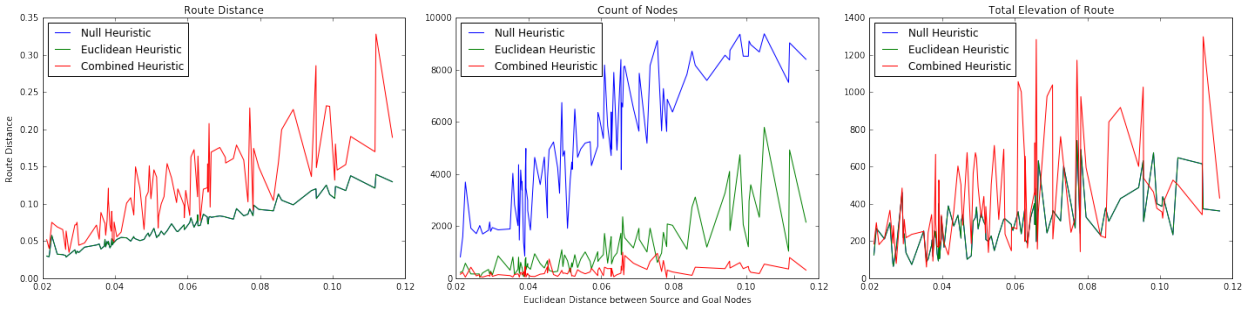


Figure **??** shows a similar comparison of the three heuristics now under the safety road cost (which accounts for road accidents). The results show that again, the combined heuristic is not admissible as elevation is not accounted for by the cost function. The euclidean heuristic again far outperforms the null heuristic in terms of reducing the number of nodes searched.

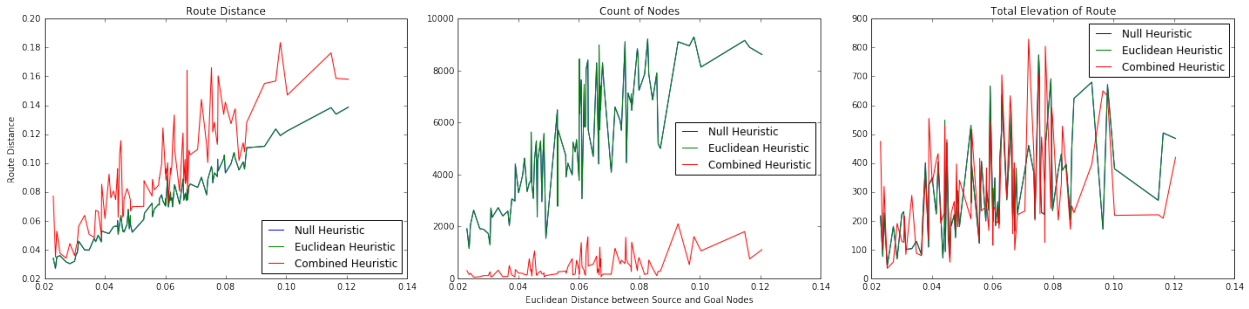
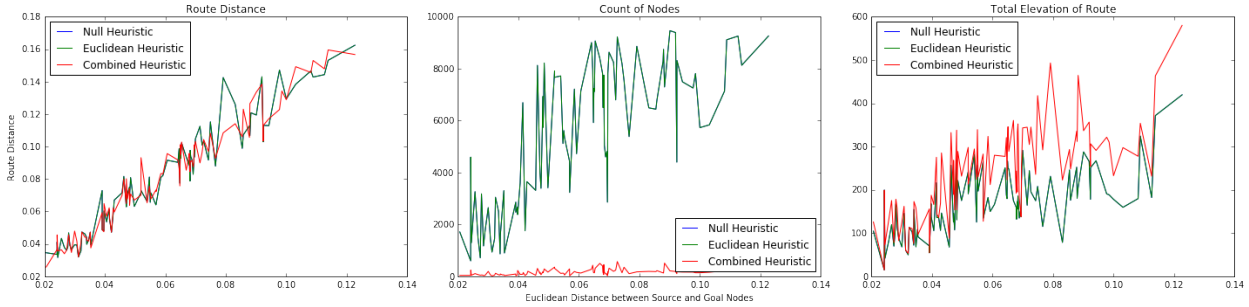


Figure 5: Safety road cost under three different heuristics.

Finally, **??** shows the comparison of the three heuristics under the safety road cost with elevation data also included. In this case, the the combined heuristic returns a path that is comparable to the other two heuristics (in terms of path distance). The fact the path length 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 difficult cost parameter to model). However, the fact that this heuristic is able to model the accident and elevation costs that are calculated by the cost function means that using this heuristic is very efficient in finding an optimal route (in terms of reducing the number of nodes searched).

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



6.2 Local Search Results

Similar to the approach used to test the A^* search algorithm, we randomly selected 4 nodes and computed the time that it took the algorithm to return a solution as well as the total cost. In figure **??** we see that k beam search algorithm generally finds a solution centroid with a slightly lower cost than simulated annealing. Furthermore, from **??**, 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 $k = 5$)

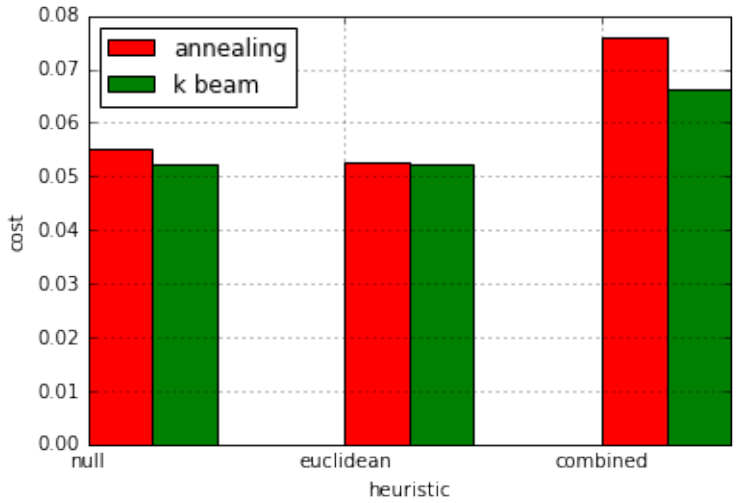
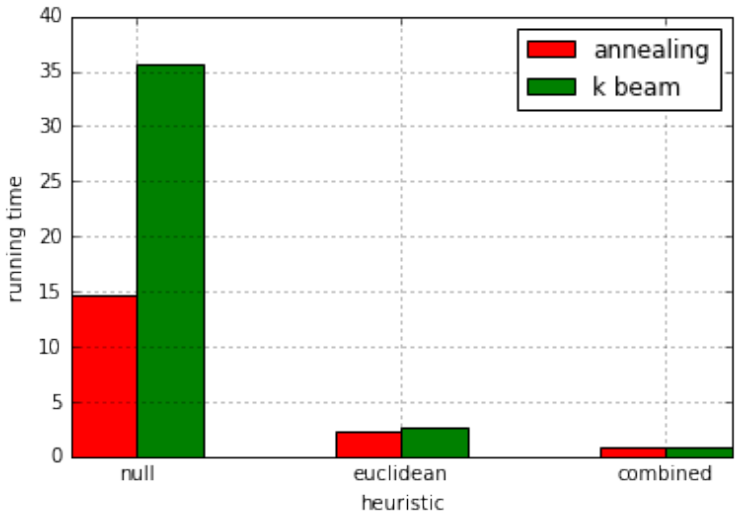


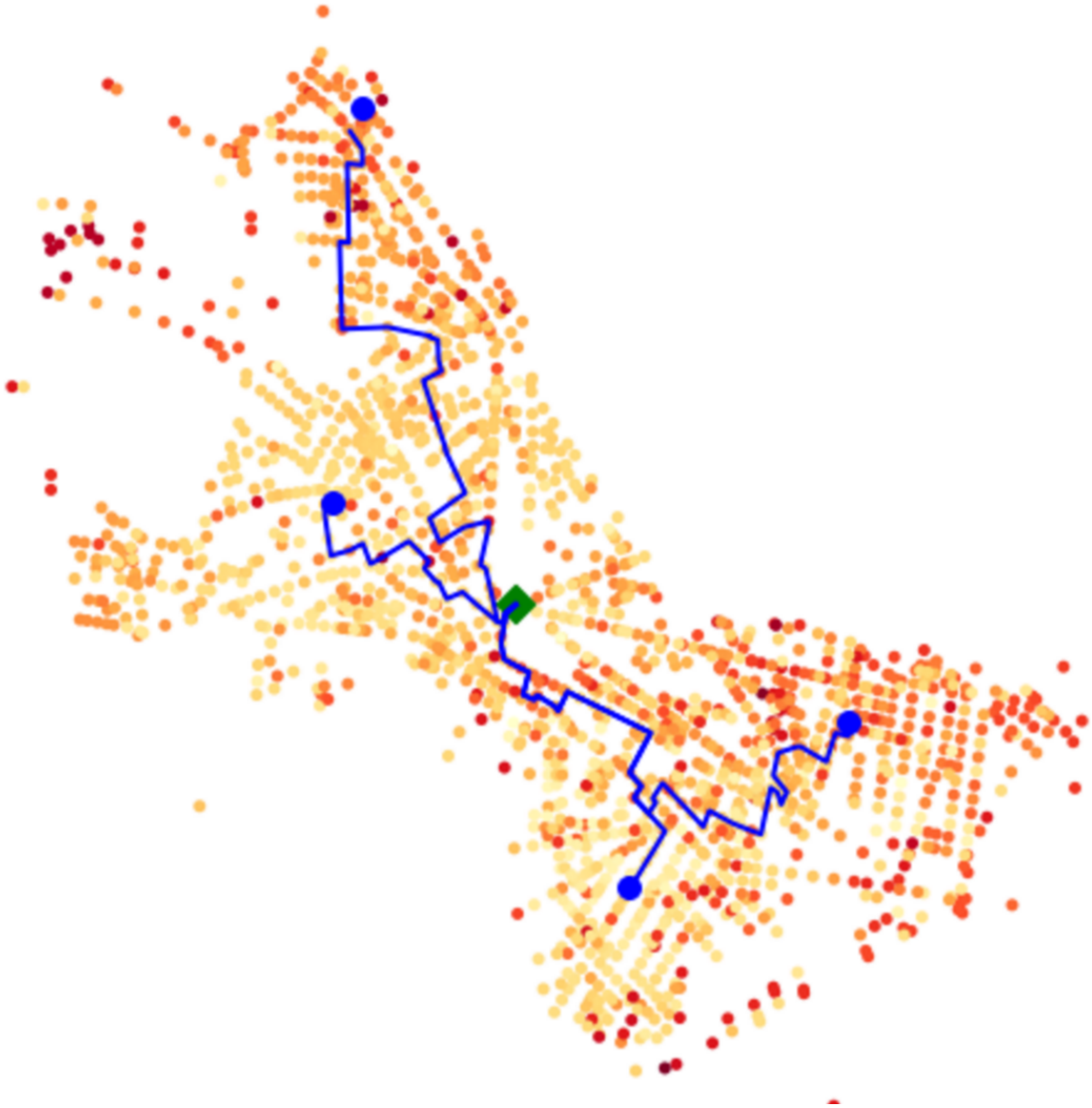
Figure 8: Total cost of routes from all starting nodes to the returned centroid for both Simulated Annealing and K Beam search (with $k = 5$)



The reason for k beam search out performing simulated annealing is the presence of very definite local optima. The initialization of multiple random starting points within the possible central region has a greater chance of resulting in the centroid being close to the optimal solution than with simulated annealing. Even with the temperature initialised at a high value, simulated annealing struggles to explore beyond the vicinity of the node that it is initialized at. Since this starting node 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 is returned by simulated annealing is still valid. The below figure **??** is a plot showing the centroid found between four cyclists (under the combined 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 centroid (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

Through this project we have explored a range of algorithms, cost, and heuristic functions for graph search in cities, optimising for different objectives and compared their performance. The graph search approach offers a highly adaptive solution to the problem of intelligent bike routing. Given our pre-existing knowledge of Cambridge and San Francisco city streetscape, we felt the results were reasonable.

The running time of the algorithms we have developed could have been decreased by incorporating pruning strategies, but run times were generally satisfactory (especially when evaluating the combined heuristic with the distance, safety and elevation costs). In developing heuristics, it was difficult to generate a useful non-trivial heuristic for bike crashes which was also admissible - an extremely convoluted path could almost always be found to avoid all bike accident locations. Therefore 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 combined heuristic did provide a good route.

Furthermore, while calculating the number of accidents per road segment is a good estimate of the danger posed by a road segment, a better indicator might 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:

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

In particular, we may be overpenalizing roads with a large amount of cycling traffic.

Takeaways from this project:

- You are only as good as your data - we had high expectations for the data we would be able to collect and use for this project, however 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 real time updates to the bike accident 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 cities.
- We initially built the system for the Cambridge data but then obtained the larger dataset for San Francisco. It was therefore interesting to see how our algorithms scaled to the larger map. To plan routes through a larger graph, it is clear that we would have to adapt the algorithm to improve the runtime performance to be within a reasonable range. Our background research suggested that many routing engines actually use inadmissible heuristics to reduce the running time but are still able to find reasonable results.

A System Description

The easiest way to use our system is to clone the repo , 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