CMU Traveling Salesman Problem

Charles Hutchinson, Jonathan Pyo, Luke Zhang, Jieli Zhou

December 16, 2016

1 Introduction

In this paper we will examine the Traveling Salesman Problem on the CMU Pittsburgh Campus and attempt to find the minimum-length tour that visits every place of interest exactly once. This is a classic Traveling Salesman Problem. Our problem is an expanded version of a traveling salesman problem originally proposed for campus tour guides. While our problem visits many places that will not be covered by the typical campus tour, the expanded set of places to visit presents challenges to our algorithms' accuracies and efficiencies that would not be as significant if the problem was limited to the scope of the typical tour. In the paper we will discuss the formulation of the problem, multiple algorithms implemented to solve it, and analysis on the results.

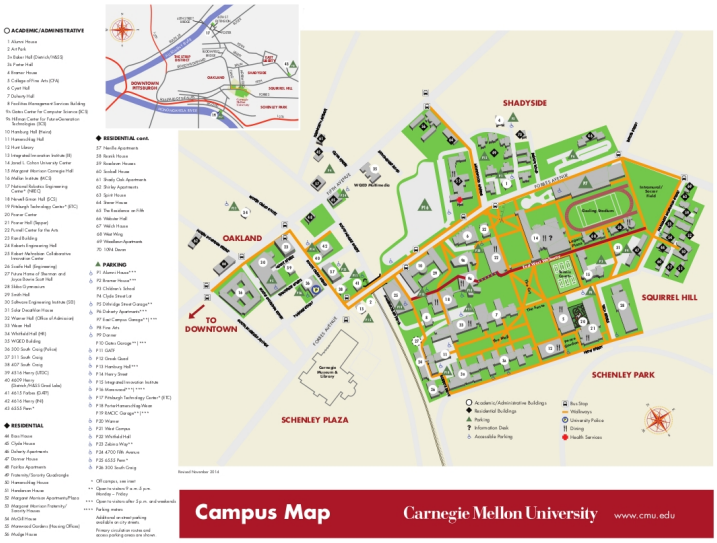


Figure 1: The Map of CMU Campus

2 Formulation

In a complete directed graph D=(N,A) with N being the set of nodes (or places of interest in this problem) and A being the set of arcs connecting the nodes and arc-costs c_{ij}, we look for a minimum-length tour (a directed cycle that contains all the nodes).
First we define our binary arc inclusion variable

$$x_{ij} = \begin{cases} 1, & \text{if arc (i,j) is included in the tour} \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Then we have the integer program constraints for our problem.

$$\begin{aligned} &\min \sum_{i,j} c_{ij} x_{ij} \\ &\text{s.t.} \sum_{i} x_{ij} = 1 \quad \forall i \\ &\text{s.t.} \sum_{j} x_{ji} = 1 \quad \forall j \\ &0 \leq x_{ij} \leq 1 \end{aligned} \tag{2}$$

The constraints of (2) are called *assignment constraints*. This integer programming assignment constraints, however, do not eliminate the possibilities of subtours, or directed cycles that do not cover all the nodes. Therefore, a subtour elimination constraint is needed.

$$\sum_{i \in S, j \in S} x_{ij} \leq |S| - 1 (S \subsetneq V, |S| > 1) \tag{3}$$

(3) eliminates subtours by dictating that for any strict subset of nodes, you can not have the same number of arcs originating from or ending at the nodes as the number of the nodes.

3 Data Collection

Searching for an accurate and efficient method to determine distances between buildings was a challenging element of this project. We began by marking CMU's buildings on Google Maps, and recording their latitude and longitude. We excluded buildings that were extremely far from main campus, as well as parking lots and buildings under construction, to come to a total of 76 points of interest. A list of the 76 buildings can be found in the Appendix.

3.1 Geometric Distance

We can calculate the 'distance' between any two points using Pythagorean Theorem

$$'Distance' = \sqrt{(latitude_1 - latitude_2)^2 + (longitude_1 - longitude_2)^2}$$

For example, the 'distance' between **Cyert Hall** (40.4442762, -79.9439342) and **Hunt Library** (40.4410927, -79.943752) is **0.0032**, while the 'distance' between **Hunt Library** and **Doherty Hall** (40.4423925, -79.9443068) is **0.0014**. This makes intuitive sense because the distance between Hunt Library and Doherty Hall is indeed about half of the distance from Hunt Library to Cyert Hall. Although this 'distance' measurement is meaningful proportionally, it cannot show how far apart two points actually are in standard distance units.

3.2 Metric Distance

Universal Transverse Mercator (UTM) coordinates fulfils this task nicely. UTM coordiante system was first developed for military purposes because of its accuracy in pinpointing targets' locations and its effectiveness in measuring distances. In essence, UTM measures any locations by three parameters: **UTM Zone, Easting, and Northing**. For our purposes, UTM zone is irrelevant since our points are in the same zone, thus we only need (Easting, Northing) to represent our point. Easting coordinates of a point measures in meter the point's 'Eastward' (x) distance to some origin in the same UTM zone, while Northing measures the 'Northward' (y) distance to the same origin. Therefore, it is obvious that given the (E, N) coordinates of any two points, we can calculate the metric distance between them using Pythagorean Theorem. For example, the metric distance between **Cyert Hall** (589559.341, 4477604.873) and **Hunt Library** (589579.019, 4477251.684) is **353.74 meters**, while the 'distance' between **Hunt Library** and **Doherty Hall** (589530.241, 4477395.401) is **151.77 meters**. Unfortunately, the conversion of (latitude, longitude) to UTM coordinates (E, N) is not easy, and a simplified formula can be found on this Wiki page. Using the metric distances found by this conversion formula, we began work on approximating the optimal tour.

4 Heuristics

A heuristic is a technique designed for quickly approximating solutions when exact algorithms have high space and time complexity. For any TSP, ideally, we would just examine all n! TSP tours and select the shortest one. However, this is almost always infeasible since n! grows extremely fast. In our case, 76! is an astronomically large number.

In statistics, if the sample space is too large to study, it is common to draw samples to study instead. Similarly, we can think of heuristics as sampling methods. We first use heuristics to select samples from the whole sample space, the n! TSP tours, then find the shortest-length tour or local minimum in each sample, and claim the local minimum can approximate the global minimum. For example, we can do 100 random permutations of n nodes to get 100 Hamiltonian paths, and then connect the two endpoints in each path to get 100 valid TSP tours. Finally, we select the minimum-length tour out of the 100-tour sample, and claim it as the shortest TSP tour of n nodes. This is obviously a very bad heuristic, but it illustrates how heuristics can be thought as sampling methods. So, as in statistics, we need to consider whether the tour sample we draw can represent the population well. If the sample is made up of outliers, then we can get stuck in local optimality and never progress to the global optimum.

There are two types of TSP heuristics based on how we construct each tour sample. If we start from nothing, and add edges one by one, then the algorithm is called a **Constructive Heuristic**. On the other hand, if we start with a TSP tour, and improve the tour by modifying some parts of it, then the algorithm is called an **Improvement Heuristic**. Nearest Neighbor and Insertion Algorithms are two simple constructive heuristics, while 2-opt and Genetic Algorithm are typical improvement heuristics.

4.1 Nearest Neighbor

The idea of Nearest Neighbor is used in many areas such as clustering, classification, and collaborate filtering. It is famous for being intuitive and is often used as benchmark to be compared with other more complex heuristics. Likewise, NN for TSP is also very intuitive from a traveling salesman's perspective: at each step travel to the nearest city and return to the beginning city from the last city. If the number of cities is very large, say millions, then we can only select a set of random starting points and run NN to get a set of NN-TSP tours. However, in our case, 76 is reasonably small, so we can do an exhaustive Nearest Neighbor, i.e. consider every node as starting points and generate 76 NN-TSP tours and select from them the shortest. The shortest NN-TSP tour is shown in Figure 2.

From the plot, there is one obvious problem: edge crossings. Crossings are not allowed in any optimal TSP tour, since they can replaced with two non-crossing edges, and by the triangle inequality, these two non-crossing edges must be collectively shorter.

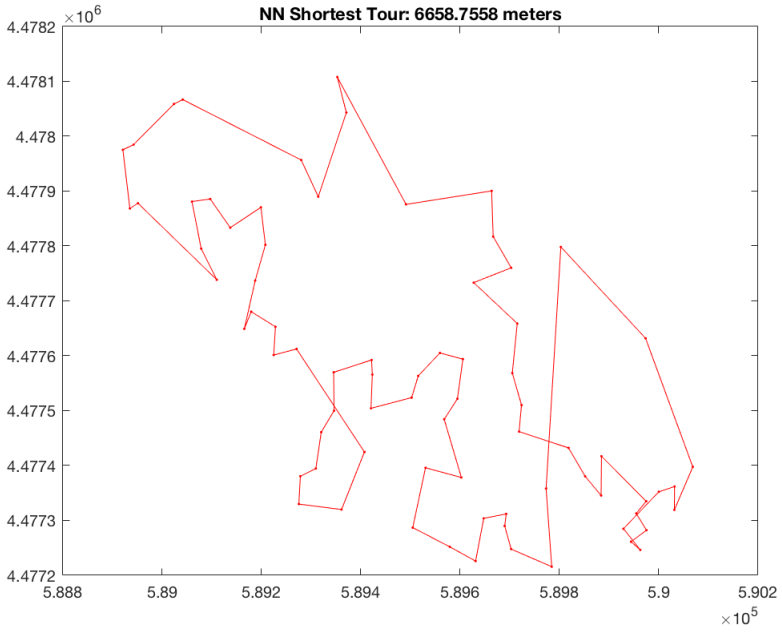


Figure 2: Exhaustive Nearest Neighbor (6658.8m)

There is a more urgent issue with NN: the inflexibility in adding new nodes. Every time we add a new node, we only add it to the end of the existing sub-tour. This inflexibility can produce a lot of crossings along the way and can potentially builds up to an unnecessarily long edge from the last node to the first node. To solve this problem, we consider a family of insertion algorithms.

4.2 Insertion Algorithms

Insertion Algorithms start with a TSP sub-tour and insert one new node at each step until we have a valid Hamiltonian path. Then, like NN, it connects the last node to the first to complete the TSP tour (notice here we ameliorate the crossings issue along the way, but there is still an unnecessarily long edge at last which can cause crossings).

Insertion of a new node (x) means deleting one old edge between two old nodes (u, v) and create two new edges connecting (u,x) and (v,x). There are four common ways to insert.

Nearest Insertion: At each step we select the node nearest to any node in our existing sub-tour.

Cheapest Insertion: Choose the x whose cost, dist(x,v) + dist(x,u) - dist(u,v), is minimized at each step.

Farthest Insertion: Opposite of Nearest Insertion. At each step choose the

node x that maximize dist(x,v) for any v in the existing sub-tour.
Random Insertion: At each step choose a random node not yet visited and insert it at the best possible position with respect to tour length. Notice that RI uses the criterion of CI at the node insertion step, but randomly chooses which new node to insert.

Sample TSP tour plots are in the Appendix. There are two points worth noticing. First, crossing issues still exist. Second, Farthest Insertion consistently produces shorter tours than NI and CI. One explanation is Farthest Insertion maintains its sub-tour structure like a convex hull, and this pushes the final TSP tour to get closer to the optimal tour. The convex hull is a convex polygon where all nodes of TSP are either its vertexes, or contained inside, as in Figure 3. A nice property of the convex hull in the TSP is that all ver-

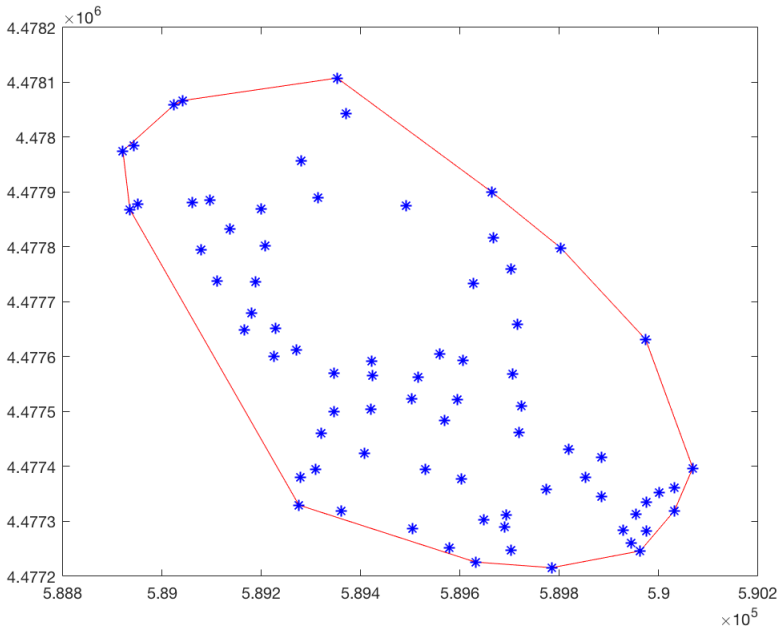


Figure 3: Convex Hull for 76-node Campus TSP

texes of the convex hull should appear in the optimal TSP tour the same order as they appear on the convex hull. It is easy to check that if that were not the case, then a crossing would necessarily emerge, which then makes the tour suboptimal.

Since there is no guarantee of producing non-crossing TSP tours from both Nearest Neighbor and Insertion Algorithms, we look at 2-opt which eliminates crossings in any tour.

4.3 2-Opt

Like the name suggests, in this improvement heuristic, we 'opt' 2 edges for another 2 edges to make the overall tour shorter. We can pick any crossing in the tours from Nearest Neighbor or Insertion Algorithms, and do a 2-opt on it. However, this elimination-by-hand process won't guarantee it is the best we can get, or 2-optimal tour. Two-optimal tour is a tour whose distance cannot be shorten by replacing 2 old edges with 2 new edges. To produce a Two-optimal TSP tour, we do a exhaustive 2-opt. Before we describe the procedure, let's look at how a crossing is eliminated technically. Consider a crossing section in a tour, ac and bd. We want this section to be like ab, cd. Ignore

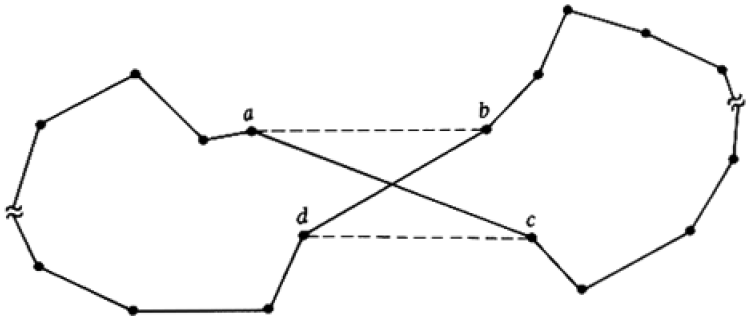


Figure 4: How crossings are eliminated?

the edges between b and c, we want the crossing section to change from *acbd* to *abcd*. This is straightforward to implement - we pick the appropriate 'b' and 'c' and swap them. What about the nodes between 'b' and 'c'? It is obvious to see, we should flip their orders to make the new 2-opt tour valid.

Now with the implementation of one 2-opt clear, we can do an exhaustive 2-opt on all the nodes in any TSP tour.

Algorithm 1 Exhaustive 2-opt

```
0: distance = dist(Tour)
1: for i ∈ {1, ..., N} do
2:    for k ∈ {i + 1, ..., N − 1} do
2:       New Tour = 2-Opt-Swap(Tour, i, k)
2:       New Distance = dist(New Tour)
2:       if New Distance <Distance then
2:          Tour = New Tour
3:    end for
4: end for=0
```

Figure 5 shows the 2-opt result of the output tour from Nearest Neighbor. This distance is less than 1 % longer than the optimal tour, which will be revealed later in the paper.

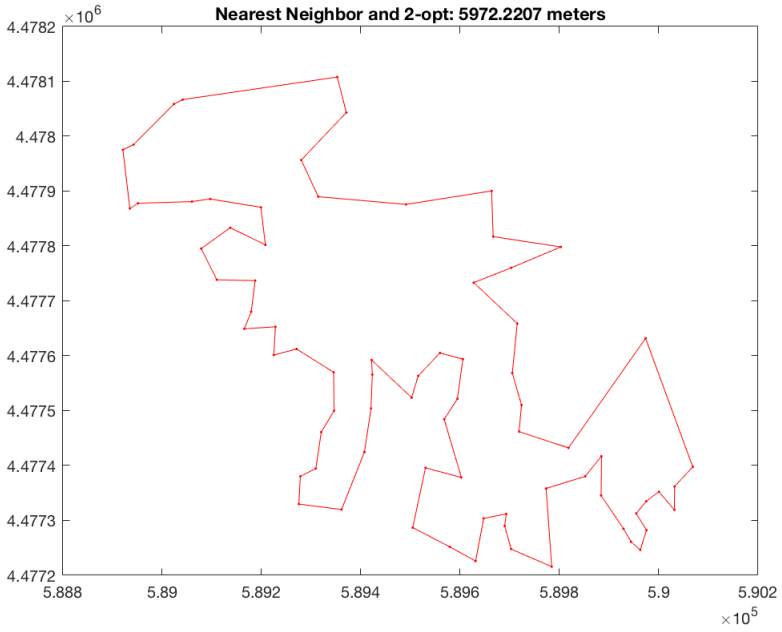


Figure 5: 2-opt on Nearest Neighbor Tour

5 Genetic Algorithm

Genetic Algorithm uses the idea of Natural Selection to evolve a sample of TSP tours. As in Darwin's Natural Selection Theory, Genetic Algorithm ensures that only the fittest tour 'survive' and have a chance to pass on its gene(sub-tour) forward.
Repeat the following process until the best solution remains the same in subsequent generations.

- Start with a population of complete TSP tours and call them generation 1. These tours can be randomly generated or from the results of other heuristics.
- Divide the population into families with equal numbers of tours.
- Select the best family member, i.e. the shortest tour.
- Put the best family members into generation 2.
- Mutate these best members chosen from generation 1 and make each of them a new family, thus completing one evolution.

All steps above should be very intuitive, except for the mutation step. Why not just keep the best family members and not including any new tours? There

are two reasons. First, we don't want our TSP tour species to die off quickly. Second, to avoid being stuck in local optimality. If all tours are the same in each generation, then there is no space left for improvement. So we mutate slightly some sub-tours of the best members from last generation. We considered three mutations.

- Sliding: randomly select two end points from the tour, and slide every node between these two end points by 1 to the right.
- Flipping (2-opt): randomly select two end points from the tour, and flip the order for all nodes in between. This will makes sure tours with crossings die off.
- Swapping: randomly select two end points from the tour, and swap them.

With these three mutations, we can ensure each generation has different members, but all members are related to the best members from last generation. Figure 6 shows the tour length of the best single member from the first to the 5000th generations, starting from random TSP tours.

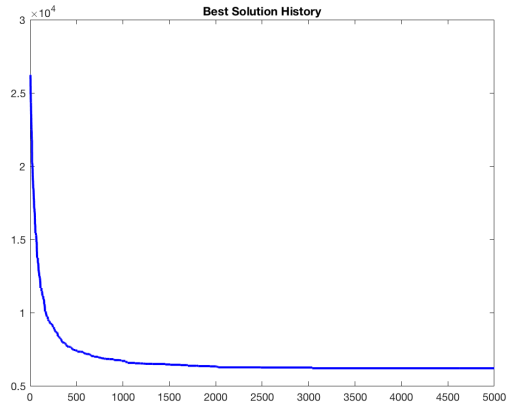


Figure 6: Distance of the best single member of each generation

If we start from the NN+2opt tour, which is already about 0.8% worse than the optimum, we improve the output to be only 0.4 % worse than optimum, as seen in Figure 7. Again, the optimal solution is found to the end of the paper.

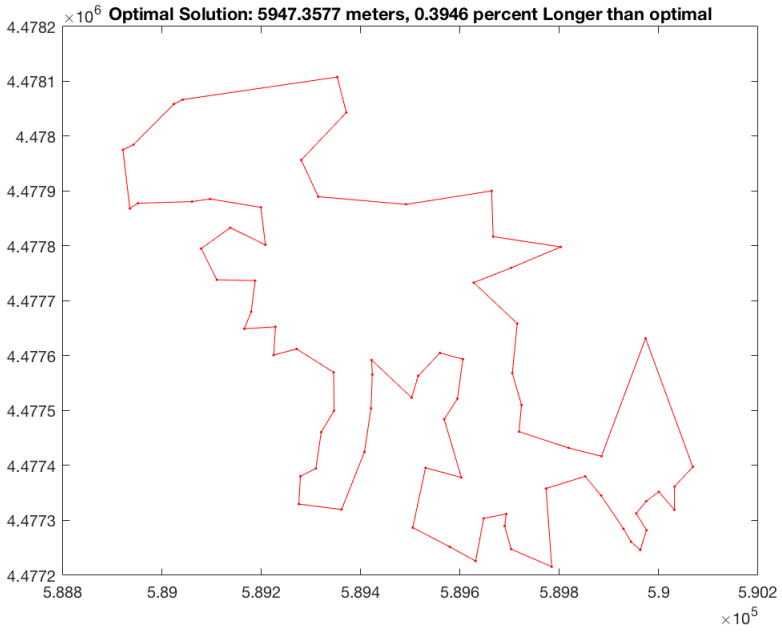


Figure 7: Genetic Algorithm on NN+2opt

6 Simulated Annealing

Another approximation algorithm we attempted for the CMU TSP is simulated annealing. Simulated annealing is a metaheuristic which approximates global optimization in a large search space. The algorithm is an emulation of the annealing process in metallurgy where heat treatment is used to alter a material's properties so that the structure can be altered as it cools down. The simulated annealing algorithm was proposed in Kirkpatrick, Gelett and Vecchi (1983) and Cerny (1985) for finding the global minimum of a cost function among several local minima.

6.1 The Algorithm

The Simulated Annealing algorithm takes a initial solution, an initial temperature, a cooling rate or cooling factor, and an iteration threshold as inputs, and returns a final solution. The solution for this problem is a sequence of numbers representing nodes. In each iteration we generate a neighboring solution by swapping the positions of two nodes in the sequence. We then compare the neighboring solution and the current solution by comparing the distances of the two routes represented by the two sequences. If the neighboring solution provides a better result, we accept the neighboring solution as our

new current solution. If not, we accept the neighboring solution with a probability of $e^{\frac{|diff|}{temp}}$ where $diff$ is the difference in total distance between the two solutions, and $temp$ is the 'temperature' of the algorithm. We accept worse solutions to provide the algorithm opportunities to escape local minima in hopes of eventually finding a better solution. With our acceptance probability, we will be less and less willing to accept worse solutions when temperature decreases in higher iterations. When the temperature is sufficiently low, or "frozen", we will take the final solution as our result. Algorithm 2 shows a summary of the Simulated Annealing algorithm.

Algorithm 2 Simulated Annealing

0: **procedure**
0: $solution \leftarrow$ random initial solution
0: $iteration \leftarrow 0$
0: $temperature \leftarrow$ initial temperature
0: $cooling \leftarrow$ cooling factor
0: $threshold \leftarrow$ iteration threshold
0: *loop*:
0: $iteration = iteration + 1$
0: $temperature = temperature \times cooling$
0: $current.distance \leftarrow$ distance(solution)
0: $neighbor \leftarrow$ swap(solution)
0: $neighbor.distance \leftarrow$ distance(neighbor)
0: **if** $current.distance > neighbor.distance$ **then**
0: $solution \leftarrow neighbor$
0: **end if**
0: **if** $current.distance \leq neighbor.distance$ **then**
0: $diff = current.distance - neighbor.distance.$
0: $prob = e^{\frac{diff}{temperature}}$.
0: **if** $rand(1) < prob$ **then** $solution \leftarrow neighbor$
0: **end if**
0: **end if**
0: **if** $iteration \geq threshold$ **then return** $solution$
0: **else goto** *loop.*
0: **end if**
0: **end procedure**=0

6.2 Modifications

After running the classic Simulated Annealing algorithm described above on our problem, we soon discovered that given the vast amount of local minima in this problem and the probabilistic nature of the algorithm, we often arrive at one of the many local minima that do not offer satisfactory results, such as the one in Figure 8.

To overcome this challenge, we can either run the algorithm until we have exhausted the local minima, or modify the algorithm so that it can explore as many local minima as possible. In order to achieve this, we introduce a new variable – the stagnation factor. The stagnation factor is a threshold on the number of iterations that can pass without the algorithm accepting a new solution. If the stagnation factor has been reached, we "reheat" the process, allowing the algorithm to escape the local minimum instead of being stuck and eventually returning it as the solution.

With the introduction of stagnation and reheating, we can no longer be sure that the final solution reached is the best solution ever achieved. Therefore, we need to record the best solution as the algorithm goes through the iterations and return the best solution instead of the final solution.

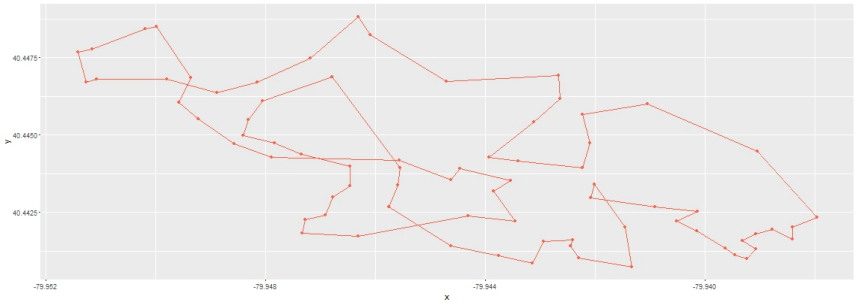


Figure 8: An Example of Local Minimum

6.3 Results and Analysis

With the introduction of stagnation, reheating, and enough iterations, simulated annealing returns a fairly satisfactory result with total distance of 6306.3 meters. This is obviously still a local minimum instead of a global one, despite our best efforts in trying to ensure the global minimum. Additionally, compared to the genetic algorithm, which is built on a similar philosophy, simulated annealing is less efficient and yields results about 7% worse. We suspect the difference mainly stems from simulated annealing's simpler and less efficient method for generating new solutions.

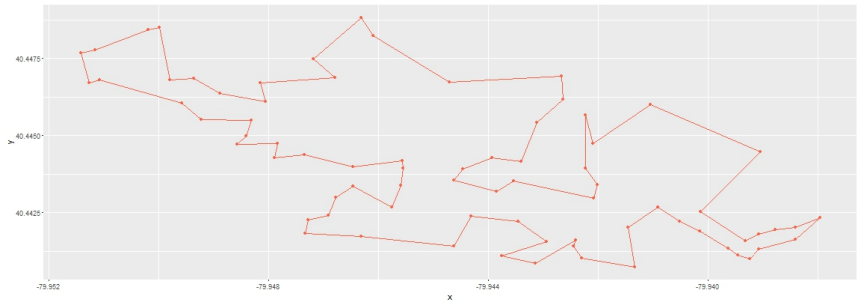


Figure 9: Best Result from Simulated Annealing (6306.3m)

7 An "Intuitive" Solution to the TSP - Held-Karp Clustering

7.1 Background Information - The Held-Karp Algorithm

The Held-Karp algorithm, created by Michael Held and Richard Karp in 1962, has the lowest asymptotic complexity of any algorithm that solves a TSP to date. The critical idea behind the dynamic program is the calculation of suboptimal shortest paths that must a) traverse an explicit set of nodes and b) end on a specific node. Here is an excerpt explaining the algorithm from Held and Karp's paper:

> Given $S \subseteq \{2, 3, \cdots, n\}$ and $l \in S$, let $C(S, l)$ denote the minimum cost of starting from city one and visiting all cities in the set S, terminating at city l. Then
>
> (5)
> (a) $(n(S) = 1)$: $\quad C(\{l\}, l) = a_{1l}$, for any l.
> (b) $(n(S) > 1)$: $\quad C(S, l) = \min_{m \in S-l} [C(S - l, m) + a_{ml}]$.
>
> To see this, suppose that, in visiting the cities in S, terminating at city l, city m immediately precedes city l. Then, assuming that the other cities are visited in an optimum order, the cost incurred is $C(S - l, m) + a_{ml}$. Taking the minimum over all choices of m, we obtain (5b). Finally, if $\mathfrak{C}$ denotes the minimum cost of a complete tour, including the return to city one,
>
> (6) $\quad \mathfrak{C} = \min_{l \in \{2,3,\ldots,n\}} [C(\{2, 3, \cdots, n\}, l) + a_{l1}]$.
>
> [2] This formulation, discovered independently by the authors, is essentially identical to one proposed by Bellman in [2].

It is important to fundamentally understand the variables S, l, a and the function $C(S, l)$ to comprehend how the Held-Karp algorithm works. $C(S, l)$ returns the distance of the suboptimal shortest path that traverses through the set of nodes in set S and ends on node l in S. Note that the calculations of the suboptimal shortest paths always take into account that the initial starting node will always begin the path. $a_{x,y}$ is the distance between node

x and y – when backtracking these distances are summed resulting in the distance of the suboptimal shortest path. Further explanation becomes easier with an example on $n = 4$ nodes given here:

$$Example: n = 4$$
$$Opt.Tour = \min\{C([2,3,4], 2) + a_{2,1}, C([2,3,4], 3) + a_{3,1}, C([2,3,4], 4) + a_{4,1}\}$$
$$C([2,3,4], 2) = \min\{C([3,4], 3) + a_{3,2}, C([3,4], 4) + a_{4,2}\}$$
$$C([3,4], 3) = \min\{C([4], 4) + a_{4,3}\}$$
$$C([4], 4) = a_{1,4}$$

Note that not all of the calculations were performed, but just enough to get an idea on how the backtracking is implemented. Every iteration of a backtrack requires finding the min between several suboptimal shortest paths and adding a distance between nodes. In English, the backtracking process calculates $C(S, l)$ by iterating through every second to last node m to find the min distance $C(S - l, m) + a_{m,l}$. This is done repeatedly until the base case is reached $S = \{l\}$, in which case that is given as $a_{1,l}$ (the distance from the first node to node l).

Unfortunately, the Held-Karp algorithm becomes difficult to use on large cases of the TSP because of its complexity. Here again is an excerpt from Held and Karp's paper.

> Again, the fundamental operations employed in the computation are additions and comparisons. The number of each in the first phase is given by $\left(\sum_{k=2}^{n-1} k(k-1)\binom{n-1}{k}\right) + (n-1) = (n-1)(n-2)2^{n-3} + (n-1)$. The number of occurrences of each operation in the second phase is at most $\sum_{k=2}^{n-1} k = [n(n-1)/2] - 1$. If one storage location is assigned to each number $C(S, l)$, the number of locations required is
>
> $$\sum_{k=1}^{n-1} k\binom{n-1}{k} = (n-1)2^{n-2}.$$

The time complexity of the Held-Karp algorithm is $O(n^2 2^n)$. Here is how you can interpret the summation: From the $n - 1$ nodes that must be traversed (we exclude the starting first node), choose k nodes to be the set S. There are k possible nodes that can be selected as the end node l, and $(k - 1)$ comparisons of C that must be made for each one. The space complexity of the Held-Karp algorithm is also bad at $O(n2^n)$. This is the total number of values of C that will be stored in the look up table. With these complexities, the Held-Karp algorithm often faces a memory problem when encountering a TSP with 30 or more nodes.

7.2 k-Means Clustering

Given the limitations of the Held-Karp algorithm, it will be extremely computationally expensive to solve our $n = 76$ TSP problem. Thus, we consider a method of representing several nodes as a few: k-means clustering.

k-Means clustering seeks to find locations of some k centers $(x_1, x_2, ..., x_k)$ for n observations $(y_1, y_2, ..., y_n)$ such that $\sum_{i=1}^{k} \sum_{j=1}^{n} ||x_i - y_j||^2$ is minimized. This is frequently done using Lloyd's algorithm, a greedy iterative process which alternates between assigning each y_j to the closest x_i (measured by Euclidean distance) and repositioning each x_i to the Euclidean mean of all y_j's assigned to said x_i. Since the value of the objective function cannot increase with each iteration, and there are a finite number of Voronoi partitions (assignments of each y_j to a certain x_i), this method must converge.

As is typical with greedy algorithms, Lloyd's method's performance is highly dependent on the initial assignment from which it begins. Normally, with a high number of nodes, this would require some sort of algorithm such as k-means++ to ensure a relatively accurate initial solution. In this particular problem, the low number of nodes ensures that brute force can find the optimal clustering with any initialization procedure.

First, we compared the value of the k-means objective function to k to determine if a specific number of clusters happened to minimize the k-means objective function on this particular problem.

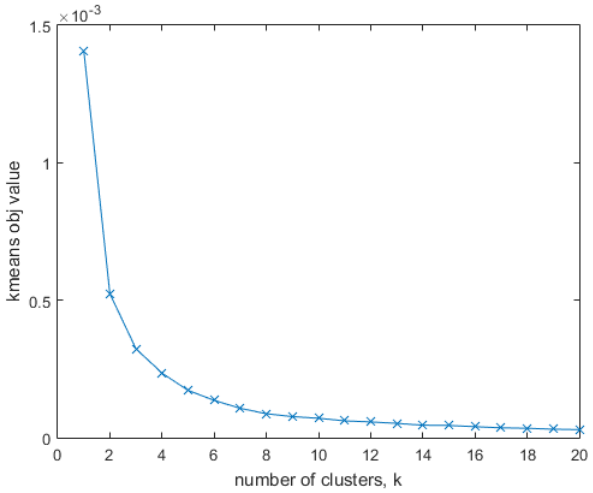


Figure 10: k-Means Cluster Sweep

Typically, a value of k is chosen at a bend in such a graph. Unfortunately, as Figure 10 shows, there is no such shape in the graph for this problem. Instead, we chose $k = 7$, $k = 9$, and $k = 11$ due to their proximity to $\sqrt{76}$, which we guessed would be the best way to divide up the complexity of the problem. We ended up dropping $k = 11$ almost immediately, due to the extremely small size of the clusters and the difficulty we had linking them together. Our $k = 7$ and $k = 9$ clustering assignments can be found in the appendix.

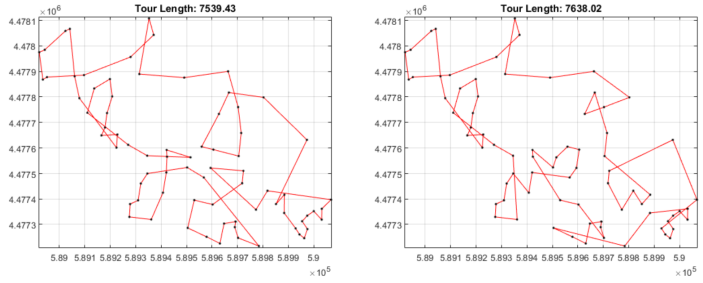
7.3 The Intuition Behind Held-Karp Clustering

Our intuition regarding this original method is as follows: if we use the Held-Karp algorithm between the k-mean clusters to find the optimal tour minimizing distances between clusters (which should incur the biggest cost based on the clustering) and then find the shortest paths in the clusters themselves starting at the entry node and finishing at the exit node, the tour length should approach that of the optimal tour. We borrowed Held-Karp code from Elad Kivelevitch to help achieve this algorithm, and as such, his licensing information can be found in Section 10.1.

Thus, our implementation started with the creation of a distance matrix of the smallest distances between each of the clusters without regard to which nodes are entry/exit nodes. Held-Karp is run on this distance matrix to find the optimal tour between the clusters to minimize the travel distance between the nodes (for 7 clusters this resulted in the order 1 3 5 4 7 6 1). Then we account for a problem that appears quite frequently: the entry and exit nodes of the cluster match. If this is the case, we note that to traverse the nodes within the cluster, this entry/exit node will violate the condition that each node must be entered/exited only once when it traverses through the nodes in its cluster. Thus, we find the smallest distance between the same clusters that utilizes a separate node for entry or exit. Finally, we run an alternate Held-Karp algorithm to find the shortest path within a cluster starting at the entry node and ending at the exit node. Ultimately, we run the Held-Karp algorithm once to find a tour to minimize the distances between clusters and k more times within each individual cluster to find the shortest path. This gets us a comprehensive tour that goes through all of the 76 points.

7.4 Results of Held-Karp Clustering

Unfortunately, the intuitive method yielded results that were visibly wrong as seen here:



7 clusters **9 clusters**

Note the frequency in which the triangle inequality is broken and the length of the tours. The Held-Karp Clustering run on 7 clusters yielded a tour length of 7539.43 as opposed to when it was run on 9 clusters yielding a tour length

of 7638.02. This may suggest the clustering size is significant to the accuracy of Held-Karp Clustering. Further discussion for improvements on our algorithm can be found in Section 10.2.

8 Cutting Plane Algorithms

Today, the best TSP solvers rely on cutting plane algorithms to quickly solve large TSP cases. These algorithms rely on the relaxation of the TSP into fractional tours, i.e., letting $x_{i,j}$ take fractional values. In this relaxed problem, we use the simplex method to efficiently find optimality. We begin with the constraint that $\sum_j x_{i,j} = 2$ for all j, or in other words, ensuring each node has an edge coming in and an edge coming out. Obviously, this sole constraint is insufficient to ensure a TSP, especially in this relaxed LP case. However, this gives an easy initialization for the cutting plane algorithm to begin.

The cutting plane algorithm iterates through three steps: find a violation of TSP conditions in the proposed fractional solution, impose an inequality on the LP to correct the violation (the cutting plane for which the algorithm is named), and find optimality under the new conditions. If the optimal solution is a valid TSP solution, it must the optimal TSP solution. Due to the complexity of the different cutting planes, we chose to implement a Gomory Cut into preexisting MATLAB code instead of solving the problem from scratch.

8.1 Gomory Cuts

In the standard integer programming problem, maximizing $c^T x$ such that $Ax = b$ and $x \geq 0$, x integer, a Gomory cut can be quickly generated from the simplex tableau of the LP relaxation of the IP. Repeated application of Gomory Cuts results in an integer solution to the modified LP which optimally solves the initial IP. First, we define $f(x) = x - \lfloor x \rfloor$. Any equation from the simplex tableau of the LP relaxation will be in the form $\sum_{i=1}^{n} a_i x_i = b$, and the corresponding Gomory Cut is expressed as $\sum_{i=1}^{n} f(a_i) x_i = f(b)$.

It is worth emphasizing that state-of-the-art TSP solvers, such as Concorde, utilize many more types of cuts than Gomory Cuts. Much of the sophistication of these advanced TSP solvers lies in the detection and prevention of subtours, which Gomory Cuts cannot prevent. Nonetheless, with our limited expertise, this was the only method which yielded any significant results. A complete implementation of a cutting plane algorithm feels more on the scale of a PHD project, instead of an undergrad term project.

The optimal tour can be found below, in the results section.

9 Results

Figure 11 outlines the optimal CMU TSP tour, and Figure 12 compares the tour length of the solution from each of our algorithms. The building indices

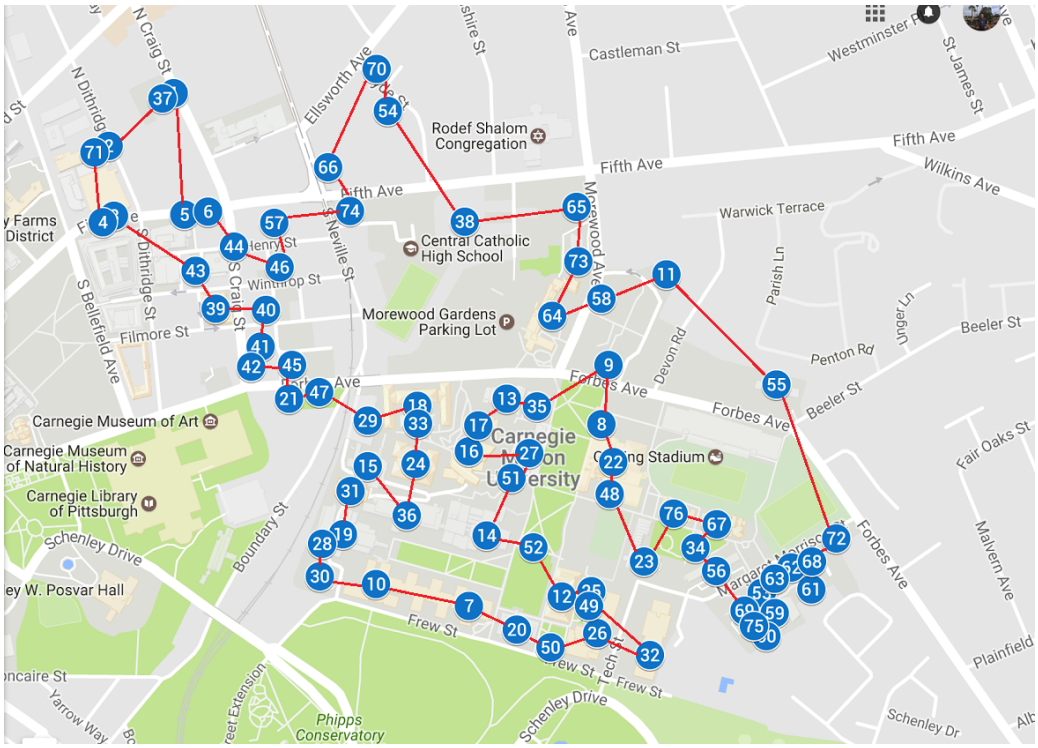


Figure 11: Optimal TSP Tour of CMU Campus

Held-Karp w/ 7 Clusters	7539.43 (27.270%)
Nearest Insertion	6876.6 (16.081%)
Nearest Neighbor	6658.8 (12.404%)
Cheapest Insertion	6586.9 (11.190%)
Farthest Insertion	6361.2 (7.381%)
Simulated Annealing	6306.3 (6.453%)
Arbitrary Insertion	6180.7 (4.334%)
Nearest Neighbor + 2-opt	**5972.2** (0.814%)
Nearest Neighbor+2-opt+Genetic Algorithm	**5947.4** (0.395%)
Lin-Kernighan (heuristic!)	**5923.98 (optimal)**
Cutting Plane (solver)	**5923.98 (optimal)**

Figure 12: Tour Lengths from All Methods

in Figure 11 are different from those on the map on the first page. Figure 11's building indices can be found in the appendix in the optimal order.

10 Further Research

10.1 Improving Distance Accuracy

Though our solution is provably optimal with our distance matrix, there are points on our tour where our solution has to be incorrect in the real world. Perhaps the most egregious example is the Gates-Purnell-Pausch Bridge segment of the optimal tour, when anyone familiar with campus would immediately say Gates-Pausch Bridge-Purnell would be faster. This problem stems from our use of Euclidean distances between buildings. Ideally, we would manually collect distances between buildings to determine a more realistic result. Since this seemed to be a significant time investment for an accuracy improvement irrelevant to using our operations research techniques, we chose to forgo this manual collection.

To increase accuracy without investing as much time, we considered using Manhattan distances in the Oakland area (due to the rectangular shape of walking paths), as well as weighted distances around the hill by Wean and Newell-Simon to account for the elevation change. The weights would have been found by comparing manually collected walking times. Unfortunately, we ran out of time to implement this in a sensible way.

10.2 Held-Karp Clustering Improvements

While the results of our "intuitive" solution were suboptimal, there are various locations that can be improved. In hindsight, there were several locations that could have yielded errors. After finding the second shortest distance to a cluster such that a different node is used for entry/exit, the clusters should have been run in Held-Karp again as this change in distances between nodes may yield a new tour between clusters. Also, varying the cluster size may lead to better results along with verifying the actual implementation of the methods involved are correct. We note that this approximation may yield better results when the clusters are more obviously gathered together or in a TSP with more nodes as the distances between clusters become increasingly important to minimize. Recent research has suggested that this kind of method is close to optimal on much larger TSP graphs. Undoubtedly, problems remain in our implementation, but the method may not be suited for a relatively small $n = 76$ TSP. Thus, we find further research necessary before counting out this method of calculating the optimal tour of a TSP.

10.3 Larger dataset and Advanced heuristics

During our research, we found that simple heuristics often can achieve near-optimal results. However, many papers using larger data set often have bad

results from simple heuristics while having much better results from their newly developed heuristics. For our problem, because the campus node dataset is small and straightforward, simple heuristics work well, but this is no guarantee that the same algorithms would work on larger dataset. So for further research, we are going to examine both larger benchmark data and collect larger data set by our own. For example, an interesting topic would be a TSP tour for all the Starbucks in the Country. As the size of node and complexity of the node structure increase, we can try more advanced heuristics like Ant-System, Reinforcement Learning, Self Organizing Map, etc. After we have solved many sample problems, we can start reflecting on the relationship between node structures and best heuristics. Intuitively, certain heuristics are likely to work well on certain kinds of node distributions. So we are curious whether we can build a 'black-box' which takes in a set of TSP nodes, and outputs suggestions of good heuristics. In some way, we want to automate the TSP solving process by matching the new node distribution to an existing one in our 'database'.

11 Appendix

11.1 Optimal TSP Tour CMU Campus with Building Names

1 - Whitfield Hall
5 - Software Engineering Institute
6 - Rand Building
44 - 4609 Henry Street
46 - 4616 Henry Street
57 - Fairfax Apartments
74 - Residence on Fifth
66 - Neville Apartments
70 - Shady Oak Apartments
54 - Clyde House
38 - WQED Multimedia
65 - Mudge House
73 - Stever House
64 - Morewood Gardens
58 - Fraternity/Sorority Quadrangle
11 - Bramer House
55 - Doherty Apartments
72 - Spirit House
68 - Roselawn Houses
61 - Margaret Morrison Apartments
62 - Margaret Morrison Fraternity and Sorority Houses
63 - McGill House
53 - Boss House

59 - Hamerschlag House
60 - Henderson House
75 - Welch House
69 - Scobell House
56 - Donner House
34 - Solar Decathalon House
67 - Resnik House
76 - West Wing
23 - Margaret Morrison Carnegie Hall
48 - Entropy
22 - Jared L Cohon University Center
8 - Carnegie Mellon University Store
9 - Alumni House
35 - Warner Hall
13 - Cyert Hall
17 - Hillman Center for Future Generation Technologies
16 - Gates Center for Computer Science
27 - Purnell Center for the Arts
51 - Pausch Bridge
14 - Doherty Hall
52 - The Fence
12 - College of Fine Arts
25 - Posner Center
49 - Kraus Campo
32 - Skibo Gymnasium
26 - Posner Hall
50 - Peace Garden
20 - Hunt Library
7 - Baker Hall
10 - Porter Hall
30 - Scaife Hall
28 - Robert Engineering Hall
19 - Hamerschlag Hall
31 - Scott Hall
15 - Facilities Management Services Building
36 - Wean Hall
24 - Newell-Simon Hall
33 - Smith Hall
18 - Hamburg Hall
29 - Robert Mehrabian Collaborative Innovation Center
47 - Art Park
21 - Carnegie Mellon Integrated Innovation Institute
45 - 4615 Forbes
42 - 417 S Craig St
41 - 407 South Craig
40 - 317 South Craig

39 - Carnegie Mellon University Police Department
43 - 4516 Henry St
3 - Webster Hall
4 - Mellon Institute
71 - Shirley Apartments
2 - Shirley Apt
37 - Whitfield Hall

11.2 Held-Karp Licensing

11.3 Heuristic Results and k-Means Clustering Assignments

Figure 13: Nearest Insertion

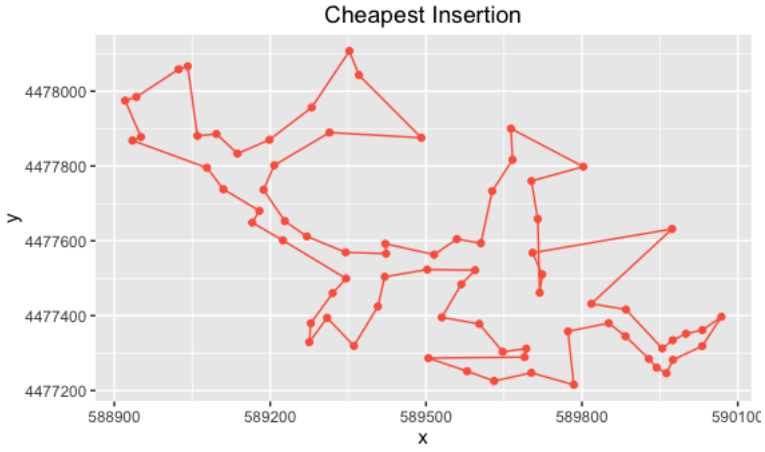


Figure 14: Cheapest Insertion

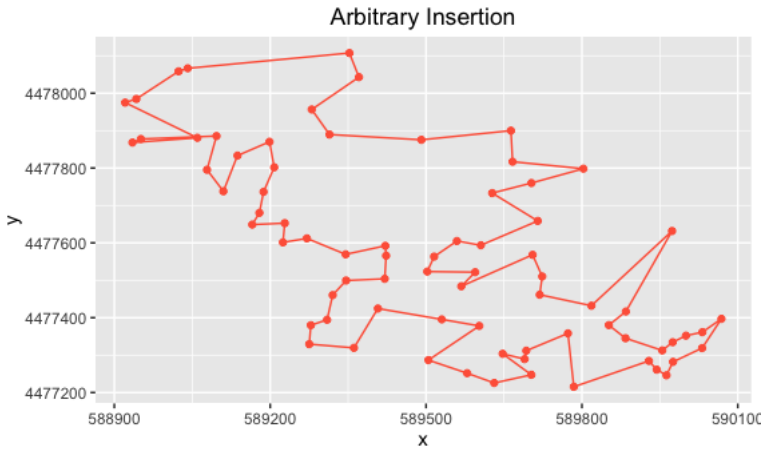


Figure 15: Arbitrary Insertion

Figure 16: Farthest Insertion

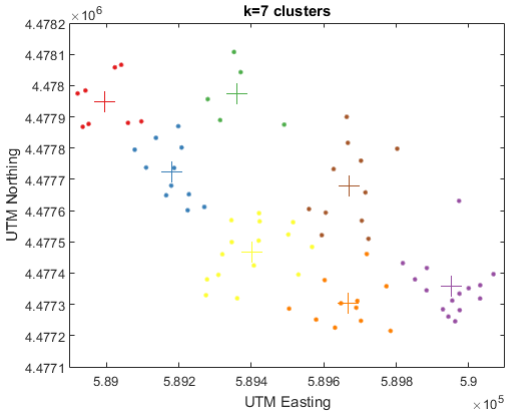


Figure 17: $k = 7$ Clustering Assignments

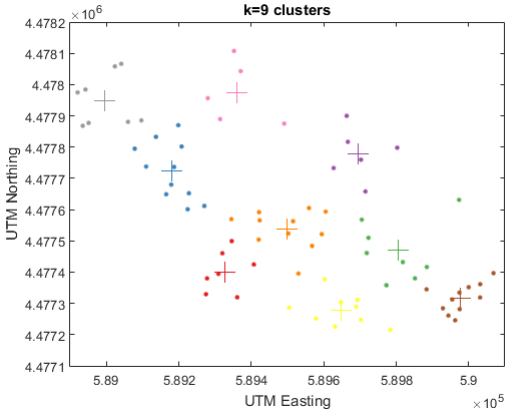


Figure 18: $k = 9$ Clustering Assignments