Remy Wolf
CSC 490

Final Project – Transit Station Alignment Algorithm

The Problem:

A city has been growing rapidly and wants to build a new rail network to serve its citizens. The rail line should serve as many people as possible, and for it to be useful, stations should be within walking distance of where people live (or want to go). The city has the budget for a given number of stations. Where should the city build stations so as to maximize service?

Modelling the Problem:

The question of how to maximize service is not explicit in what "maximizing service" actually means, so a way to model the problem should be clarified before attempting to design the algorithm. There exist several algorithms for optimizing transit routes and frequencies[1] or even station locations along a given line[2], but these often depend on some pre-existing infrastructure (street layouts, predetermined routes, etc.). For the problem of designing the infrastructure from the ground up, a new approach was needed. Linear programming was considered, but as with other problems involving transportation networks, the objective function is unlikely to be linear[1] and consequently this would not be suitable. A gravity-based model was attempted, where station placements were scored based on distance to and population of certain locations, but this became rather convulted when trying to correct for stations serving the same population centers. To avoid the issues of the previous approaches, it was decided to model the problem as a flow network.

It seems intuitive to use a flow network to represent the problem: people flow from population centers to transit stations, and the goal is to maximize the total number of people using the system (with this total number of people henceforth being referred to as Level of Service, or LOS). We start with a bipartite graph, with population centers (henceforth referred to as destinations) in one set and stations in the other. We include a super-source s, and from it draw edges to each destination with the capacity being the population of that destination. We can also draw edges from each station to a super-sink t, each with an unlimited capacity as we will assume stations can handle an unlimited amount of people. Finally, we draw edges from each destination to each station, with capacity being some number representing the amount of people who would travel from that destination to that station. Suppose there are destinations A, B, and C, with populations of 40,000, 12,000, and 30,000, respectively, and two stations. This produces a network that looks like the following:

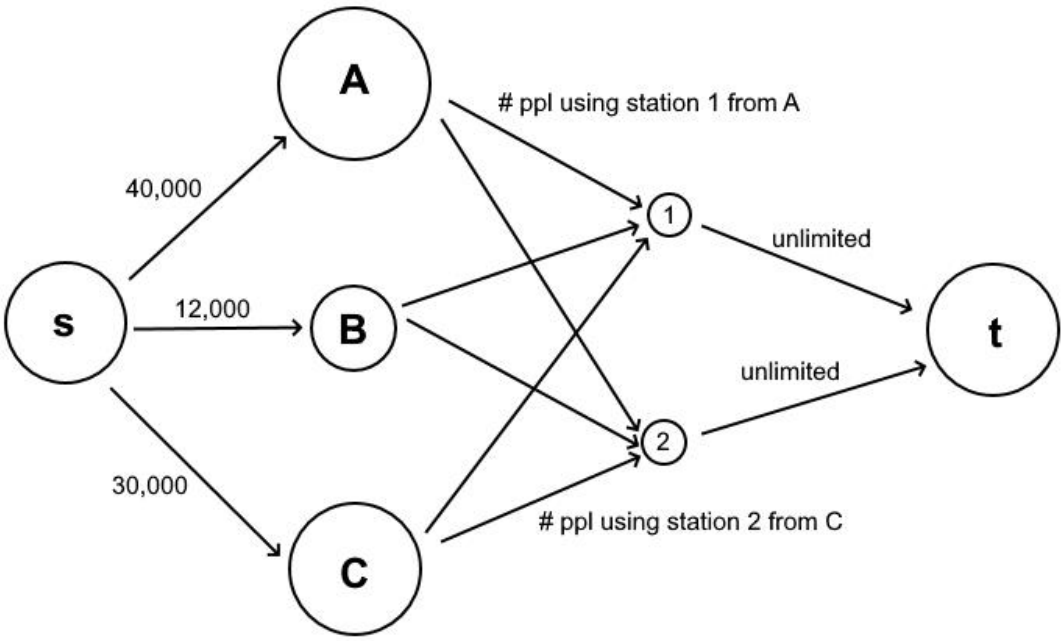

Fig 1. Sample flow network with three destinations and two stations. Some edge labels have been omitted for clarity.

Calculating LOS:

In order to determine the optimal alignment, we need a way of calculating the LOS of any given alignment. Because of the way the problem is modeled, the LOS of a given alignment is equal to the maximum flow through the network. We could use Push-Relabel or Edmunds-Karp to calculate this, but the simplistic nature of the model allows for a simpler approach. Because the graph is bipartite, and the capacity of the edges between each station and the super-sink are always unlimited, it is impossible to have a blocking flow in the graph. Consequently, we can calculate the maximum flow simply by iterating over each destination and adding the edge capacities between the destination and each station to the total flow, up to the population of the destination. Effectively, we are pushing as much flow from each destination to each station, either until all the edges are used up destination's population has been reached. This simpler algorithm has a running time of O(ds), where d is the number of destinations and s is the number of stations, as it can be implemented using a nested loop iterating through each station for each destination.

Determining Number of People Travelling to Each Station:

Next, we need a way to determine how many people will travel to a given station from a given destination. Intuitively, this number should be related to both the population of the destination as well as the distance between the destination and the station. Several different metrics were attempted, including population divided by distance squared and population divided by taxicab distance. However, these metrics had the downside of punishing small differences in distance too harshly. Someone who would travel five minutes to a station would likely travel seven minutes as well. Consequently, a more complex approach was required.

A function was desired that would send roughly the same amount of people from a destination to a station up until some specified maximum distance, and then drop off sharply after that. The sigmoid function, defined as $\frac{1}{1+e^{-x}}$, seemed to be the ideal candidate. The sigmoid function remains close to 0 up until the inflection point x = 0, after which it rises to and approaches 1.

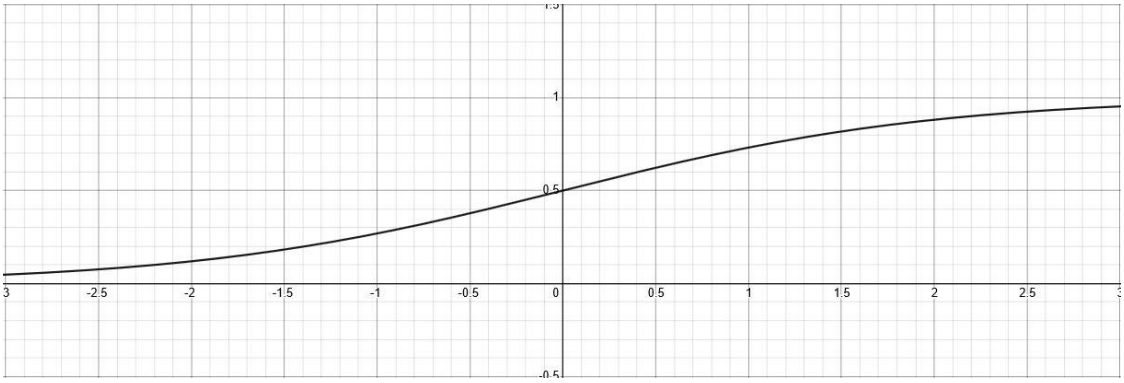

Fig 2. The sigmoid function.

A few adjustments needed to be made:

1) The function should approach the population of the destination as distance decreases.
2) The point of inflection should be the maximum distance estimated people will travel to a station.
3) A variable to determine how sharply the function drops off close to the inflection point should be included to allow for more flexibility depending on the scenario.

To accommodate these adjustments, the function was modified to the following:

$$\frac{P}{1 + e^{\left(\frac{t}{m}\right)(d-m)}}$$

where P is the population of the destination, m is the maximum distance estimated people will travel to a destination, d is the distance between the destination and the station, and t is the "tightness" determining how sharply the function should drop off close to the inflection point. If we let P = 40,000, m = 15, t = 5, and plot the function over distance, we get the following:

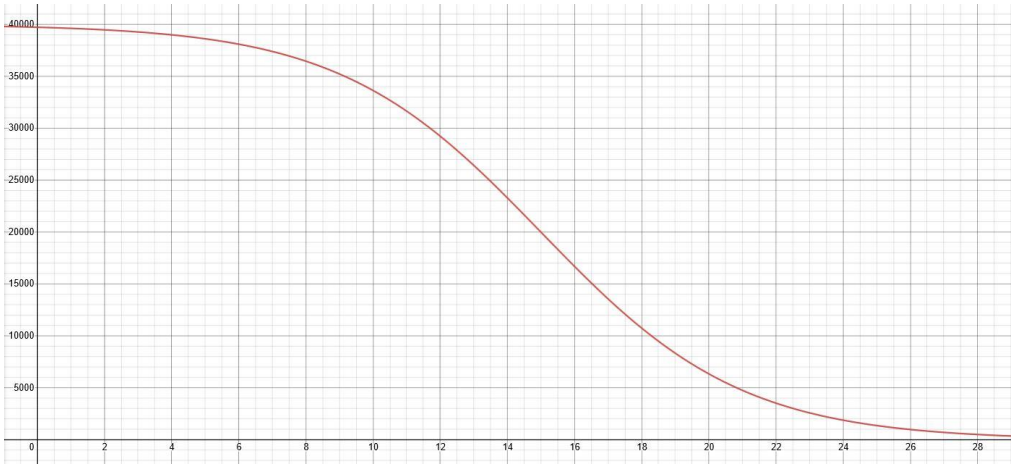

Fig 3. The modified sigmoid function, representing the number of people who will travel to a station from a given destination based on the distance between the two.

This function would allow us to calculate the amount of people travelling from a destination to a station for each destination/station pair, and therefore calculate the total LOS of a given alignment.

Designing the Algorithm: Brute Force Approach

Now that an applicable model has been established, we can design an algorithm that calculates the optimal alignment given a number of stations and set of destinations. (The actual algorithm implementation also requires the estimated maximum travel distance and tightness of the sigmoid function as inputs to calculate LOS, but we will ignore this for simplicity.) The function should output both the maximum achievable LOS as well as the (x, y) coordinates of each station to achieve said LOS. Transportation algorithms typically involve discrete variables and tend to take the form of integer progrmming models[3]. As such, it seems appropriate to limit the possible coordinates to integer values only. This allows us to take a brute force approach, by systematically trying every possible combination of station alignments and outputting the best one. This can easily be implemented using recursion, and the pseudocode is as follows:

```
brute_force(num_stations, destinations, current_alignment):
```

Input: The remaining number of stations to be placed, a list of destinations, and the current alignment of stations
Output: The maximum achievable LOS as well as the alignment of stations that achieves it

```
    if num_stations is 0:
        // base case
        calculate the LOS of this alignment and return it, along with
        current_alignment

    max_LOS = 0
    best_alignment = []

    for every possible coordinate:
        add this station to current_alignment
        LOS, full_alignment = brute_force(num_stations – 1, destinations,
                                          current_alignment)
        if LOS > max_LOS:
            max_LOS = LOS
            best_alignment = full_alignment

    return max_LOS along with best_alignment
```

Time Complexity Analysis:

Let s be the number of stations, d be the number of destinations, X be the number of x coordinates, and Y be the number of y coordinates. The loop inside the function will run for XY iterations, and the function will recurse s times, giving us (XY)^s total iterations. Furthermore, it was shown earlier that calculating the LOS of an alignment can be done in O(ds) time, and this calculation happens on every iteration, giving us a final time complexity of O(ds * (XY)^s) time.

Improving the Algorithm

The brute force algorithm can be slightly improved. Note that the ordering of stations does not matter—the alignment with stations at (1, 2) and (3, 4) has the same LOS and is therefore equivalent to the alignment with stations at (3, 4) and (1, 2). If we can prevent the algorithm from iterating over duplicate combinations of coordinates, we could improve the time complexity. A small change to the pseudocode accomplishes this. We include the coordinates as an extra parameter to the function so we can ensure each recursive step only iterates over coordinates that have not been checked:

```
brute_force_improved(num_stations, destinations, current_alignment,
                     coordinates):
```

Input: The remaining number of stations to be placed, a list of destinations, the current alignment of stations, and the set of valid coordinates
Output: The maximum achievable LOS as well as the alignment of stations that achieves it

```
    if num_stations is 0:
        // base case
        calculate the LOS of this alignment and return it, along with
        current_alignment

    max_LOS = 0
    best_alignment = []

    min_index = 1
    // assuming lists are indexed at 0
    max_index = number of coordinates – num_stations – 1
    for every coordinate in coordinates[0 : max_index]:
        add this station to current_alignment
        LOS, full_alignment = brute_force(
                                num_stations – 1, destinations,
                                current_alignment,
                                coordinates[min_index : # of coords])
        if LOS > max_LOS:
            max_LOS = LOS
            best_alignment = full_alignment

    return max_LOS along with best_alignment
```

By splicing the set of coordinates in this way, we avoid iterating over identical combinations.

Time Complexity Analysis:

Using this method, there are (XY choose s) iterations of the main loop. Using the formula for combinations, this gives us $\frac{(XY)!}{s!(XY-s)!}$ total iterations. We can rewrite this as $\left(\frac{1}{s!}\right) * \frac{(XY)(XY-1)(XY-2)\ldots(XY-s)!}{(XY-s)!}$. The (XY-s)!

terms will cancel out, leaving us with (XY)(XY-1)(XY-2)… which is roughly equal to (XY)^s, leaving us with $\frac{(XY)^s}{s!}$. We still calculate LOS during each iteration, so if we multiply by O(ds), we get $\frac{((XY)^s)ds}{s!}$. We can cancel an s term from the top and bottom, which results in $\frac{((XY)^s)d}{(s-1)!}$. This gives us a final time complexity of O((d * (XY)^s) / s!), a slight improvement over the previous algorithm.

Greedy Approximation

Small differences in the placement of one station are unlikely to affect the optimal placement of other stations. Therefore, we can approximate the optimal alignment using a greedy approach. We find the best placement for one station by calculating the LOS at every coordinate, then find the best placement for the next station by calculating the LOS at every coordinate (this time including the first station in the LOS calculations), and so on until every station has been placed. No longer needing to use recursion, the algorithm can be implemented with a nested loop:

```
greedy_approximation(num_stations, destinations):
```

Input: The number of stations to be placed and a list of destinations
Output: An approximation of the maximum achievable LOS as well as the alignment that produces it

```
total_LOS = 0
best_alignment = []

for every station:
      // keep track of best placement for this station
      max_LOS = 0
      current_alignment = best_alignment
      for every possible coordinate:

            add this station to current_alignment
            calculate the LOS of this alignment
            if LOS > max_LOS:
                  max_LOS = LOS
                  best_alignment = current_alignment
      total_LOS = max_LOS

return max_LOS along with best_alignment
```

Time Complexity Analysis:

This approximation offers a massive time complexity improvement over the previous algorithms. The nested loops will iterate XY * s times, and we calculate the LOS during each iteration, giving us a final time complexity of O(XY * ds^2) time, which is polynomial.

Approximation Analysis:

A proof of how well the greedy approach approximates the optimal alignment was out of the scope for this project. The performance for any specific problem will depend on the placement of destinations as well as the values of the hyperparameters `max_distance` and `tightness`. In several examples, the greedy algorithm manages to find the optimal alignment or very close. In the worst known example, the greedy algorithm produces an alignment with 83% of the maximum achievable LOS. The true lower bound could be lower, but a more extensive analysis would be required to determine it.

Sample Runs:

For the following diagrams, red points are destinations with their populations labeled, and blue points are stations.

Example 1: 5 destinations, 2 stations, `max_distance` = 3, `tightness` = 5:

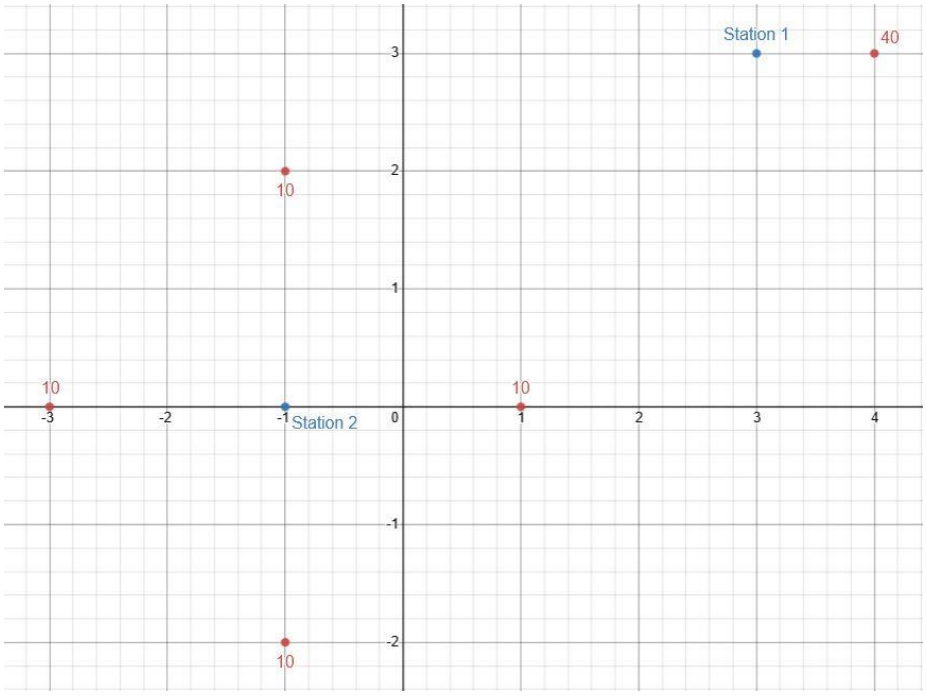

Fig. 4. Both the brute force and greedy approximation algorithms find the optimal alignment here, with LOS = 72.97.

Example 2: 10 destinations, 4 stations, `max_distance` = 3, `tightness` = 5:

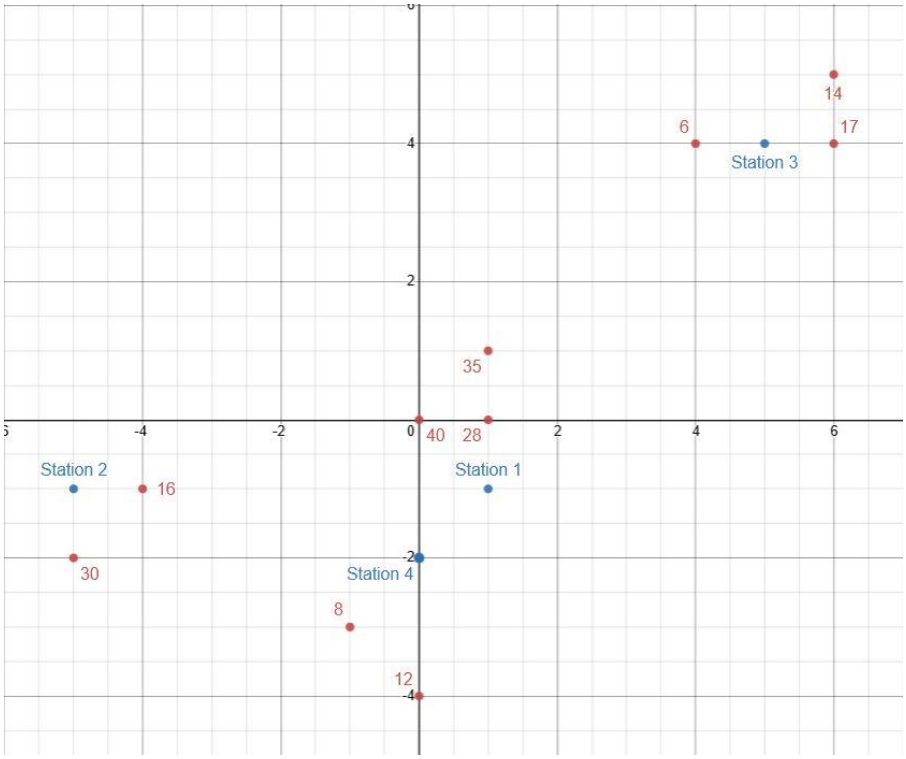

Fig 5a. The brute force algorithm finds the optimal alignment with LOS = 202.99.

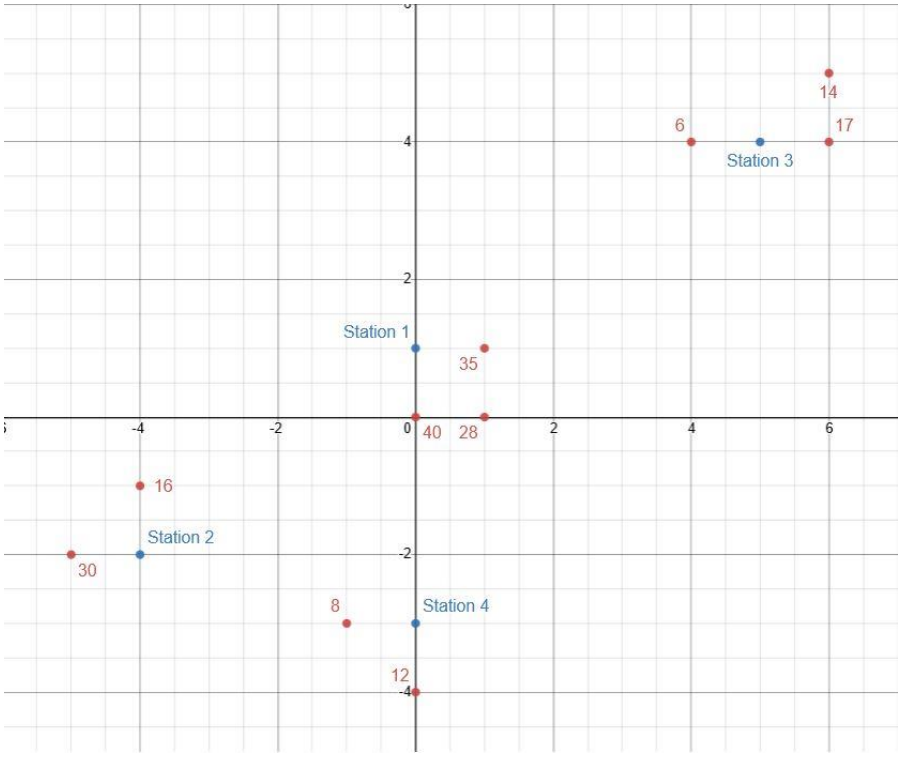

Fig 5b. The greedy approximation finds a near-optimal alignment with LOS = 201.83, 99% of the optimal solution.

Example 2: 3 destinations, 2 stations, `max_distance` = 3, `tightness` = 100:

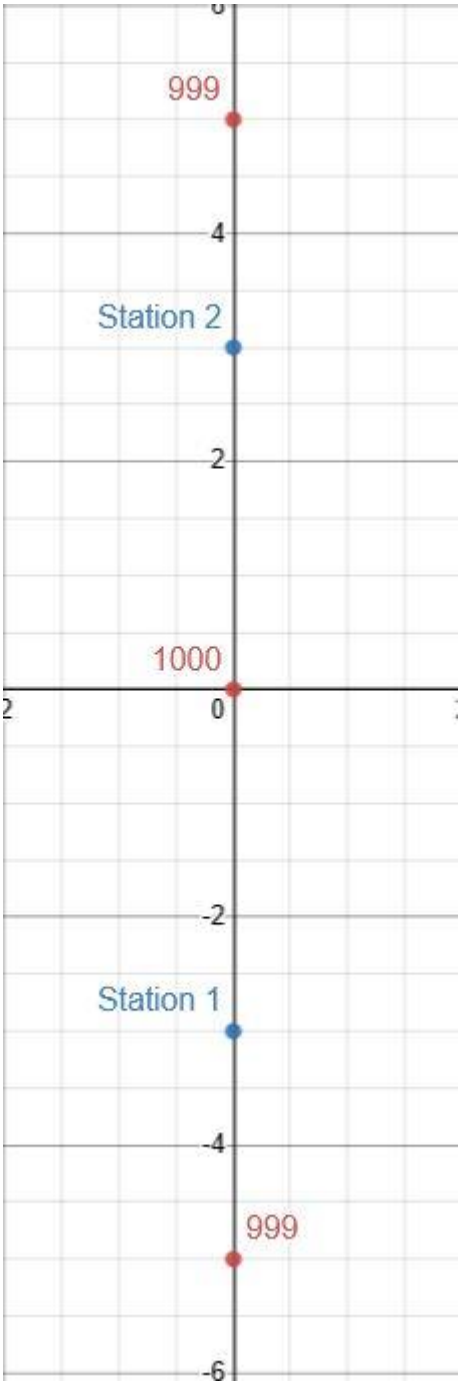

Fig 6a. The brute-force algorithm finds the optimal alignment with LOS = 2998.

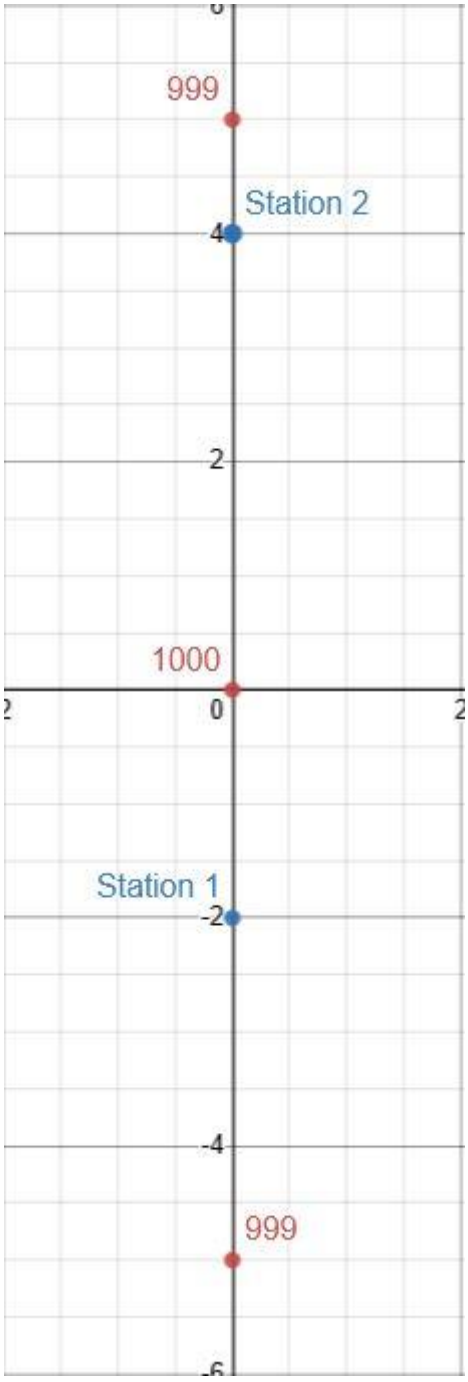

Fig 6b. The greedy approximation finds an alignment with LOS = 2498.5, 83.33% of the optimal solution.

Further Improvements:

The brute-force algorithm is unsatisfying, and there likely remain improvements to be made, either through improving the time complexity of the brute-force algorithm or increasing the efficiency of the greedy approximation. This problem is likely in NP or NP-Hard, so it is unlikely that a truly polynomial algorithm exists. Nevertheless, dynamic programming, calculus, linear programming, or some combination of other techniques could potentially be used to improve the time complexity.

Outside of time complexity, the practicality of the algorithm could be improved as well. As is, the algorithm only considers placement of stations, making the assumption that once people enter the transit network, they will be able to get wherever they want to go. This is an oversimplification: in real life, the transit network will only be used if it takes people to destinations they want to go. Other approaches make use of origin-destination matrices to determine where trips will be made[1]. By tweaking the model to include how many people want to go to a destination in addition to the population of the destination, the algorithm could produce more useful results.

Practical Applications:

The practical applications of this algorithm are admittedly limited. The time complexity of the brute force approach make it impractical to use in real-world scenarios, although the greedy approximation does perform well in most realistic scenarios with a significant time-complexity improvement. Furthermore, the algorithm only considers where to place stations, and ignores routing of lines between those stations or potential equity concerns that a real planner might have.

This algorithm could, however, be useful in combination with other techniques. Van Nes, Hamerslag, and Immers describe an algorithm that determines optimal routes and frequencies of those routes for an already existing network[1]; the station-alignment algorithm could be used to determine the location of stations, followed by the routing algorithm to determine where routes should be run to connect those stations.

Conclusion:

The transit-alignment algorithm successfully determines where a city should build a given number of transit stations based on the locations of its population centers. However, the brute-force approach comes with serious time complexity concerns that make it impractical for most real-world applications. The greedy approximation does manage to find an optimal or near-optimal alignment for most realistic scenarios, and further improvements could be made to improve either the time complexity or the efficiency of the approximation. A combination with other approaches, such as an origin-destination matrix or routing algorithms, could be used to make the algorithm more practical for real-life scenarios.

GitHub Repository

The code used to test the algorithm and generate examples can be found here.

References:

1. van Nes, Rob, Hamerslag, Rudi, and Immers, Ben H, (1998) Design of Public Transport Networks. Transportation Research Record. http://onlinepubs.trb.org/Onlinepubs/trr/1988/1202/1202-010.pdf

2. T. L. Magnanti, R. T. Wong, (1984) Network Design and Transportation Planning: Models and Algorithms. Transportation Science 18(1):1-55. https://doi.org/10.1287/trsc.18.1.1

3. Mathew, Tom V, (2006) Transportation Network Design. http://www.princeton.edu/~alaink/Orf467F15/TransportationNetworkDesign_Mathew.pdf