The Walk(ing) Side of Ithaca: Improving the Sidewalk Improvement Plan

Sam Boardman, Ian Paul, and Jeremy Wang

November 18, 2019

Abstract

We present three algorithms to make three different aspects of city sidewalk improvement computationally tractable: selection of blocks, project structuring among blocks, and repair cost minimization for each block. We provide theoretical analysis of properties and tight time bounds to argue that these algorithms are adaptable to other objectives and scalable. Moreover, we consider the relatively small sizes of the inputs they require in the space of a city sidewalk system to state modifications to achieve greater optimality of output. We form an iterative process involving all three algorithms to facilitate their application to a city sidewalk system.

Contents

1 Introduction

The city of Ithaca has a population of approximately 31,000 residents and 20,000 students [1]. When it comes to pedestrian accessibility, sidewalks are the fundamental element that must be maintained. Many people rely on walking the sidewalks of Ithaca to commute from one place to another and to immerse themselves in the beauty of nature. Despite the best effort of the City of Ithaca's Sidewalk Improvement Program, there still exists difficulties in making sidewalks more "walkable." Many problems can damage sidewalks, including uprooted tree growth and repeated intervals of freezing and thawing in the ground. In this paper, we explore various algorithmic approaches to optimizing the allocation of resources to and planning of large-scale public amenities, particularly sidewalks, culminating with a recommendation for Ithaca's Sidewalk Improvement Program strategies to improve the sidewalk experience for all of its constituents.

Specifically, we devise algorithms to

- assign priorities to all blocks in Ithaca to choose on which blocks sidewalks are improved;
- group nearby blocks to fix as to minimize moving costs for a construction team; and
- select a repair strategy for all the concrete slabs in a block to minimize total cost

A city-wide sidewalk system is multi-faceted in both its goals and its logistical challenges, and accordingly, each algorithm influences the efficacy of the next. Moreover, the algorithms can be unified in many ways to offer a holistic improvement to the state of sidewalks in Ithaca, and we attempt to synthesize them in pursuit of this goal. Therefore, we draw on concepts from mathematics, computer science, and economics to offer techniques that are socially optimal in various senses developed below. In some cases, our solutions retain degrees of freedom that allow them to be applicable to an assortment of cities, including Ithaca. We hope that they contribute insight to mathematical modeling, especially as applied to improving our cities.

2 Assumptions

We assume that the City of Ithaca has a yearly sidewalk budget of 865,000 dollars per year, and that our plan to improve the sidewalks in Ithaca can make use of the entirety of that budget; in other words, our plan for improving sidewalks is implemented over the span of one year. However, the algorithms in Sections 3, 4, and 5 are written generally enough to apply to almost any period of time; only in Section 6 does this assumption affect our recommendations, but even then there are analogous recommendations for other choices of period of time. The term *block* will refer to a small amount of land surrounded by street, with the blocks in Ithaca pre-determined by its governing body. Each block may or may not be spanned by a *sidewalk*, a sequence of concrete slabs adjacent to a road. The sidewalk condition of a block is said to be improved if sidewalk *repairs* are performed on a pre-existing sidewalk. We neglect the construction of new sidewalks due to the incompatibility of uncertain costs with the optimization nature of the algorithm in Section 5, a critical component in ascertaining which blocks may be selected for sidewalk improvement. The repairs performed on a block are solely done to ensure compliance with the Americans with Disabilities Act (ADA), which means that no repairs are performed on compliant blocks even if they are selected for repairs. In this case, our algorithm in Section 5 will indicate the cost of repairs is 0 dollars. Moreover, we assume that, in the inevitable case of constrained financial resources, the city officials of Ithaca are interested in using only the following criteria to distinguish prospective blocks: population density; proximity to schools, bus stops, governmental buildings; number of complaints; and the physical condition of concrete slabs—on a block, the worst violation of the Americans with Disabilities Act.

3 Priority Score for Blocks

3.1 A Notion of Optimality

Given their limited financial resources, many cities must choose where to allocate funds for sidewalk repair based on various criteria. Consider a list of such criteria, $l_1, \ldots, l_n$. Suppose we measure the extent to which a block satisfies each criterion l_i with a variable x_i, which takes on non-negative values. The only requirement of the measure is that, given any two values x_i can exhibit, the greater one indicates a block has satisfied the criterion to a greater extent. Alternatively, if they are equal, they indicate satisfying the criterion to the same extent. Denote by x_{is} a value of x_i for a block s within the set of blocks S. Similarly, define $\mathbf{x}_s = \{x_{1s}, \ldots, x_{ns}\}$. We say, with respect to sidewalk repair, that $s \in S$ is *Pareto optimal* if it is *not* the case that for some $s' \in S$, there exists $i \in \{1, \ldots, n\}$ such that $x_{is'} > x_{is}$, and for all $i \in \{1, \ldots, n\}$, $x_{is'} \geq x_{is}$ (otherwise, we would say that s' *strictly dominates* s). The Pareto front F consists of all $s \in S$ that are Pareto optimal. Intuitively, F contains every block s for which there is no other block that satisfies each criterion at least as well as s *and* better than s for at least one such criterion. Therefore, F is exactly the set of blocks a rational agent interested in improving the sidewalk condition of one block could justify selecting, given they value all the criteria; each $s \in F$ satisfies each criterion at least as much as any other block in S or satisfies one to a greater extent, but for all $s' \notin F$, there is some $s \in F$ that satisfies each at least as much and some better.

3.2 Selection of Blocks

A rational agent such as the one above could be a city official, in which case they may wish to improve the sidewalk condition of multiple blocks. In this case, while the first block selected for improvement should be some $s \in F$, it is unclear which blocks should be considered next, because some $s' \notin F$ may be in the Pareto front for the set of choices $S \backslash \{s\}$; if $s' \notin F$ solely because s' was strictly dominated by s, then it would still be rational in the sense above to select s' after s. In general, given a set of roads improved $S_I := \{s_i : i \in I\}$, we want that the next road improved be in the Pareto front of $S \backslash S_I$, denoted F_I. For any $S_I \subset S$, we observe that $F \backslash S_I \subset F_I$, since the conditions to be in the latter are weaker (i.e. there are less elements from S that can strictly dominate elements of $S \backslash S_I$). We could determine the elements of F_I not in F; however, we limit ourselves to the elements of $F \backslash S_I$, and instead recommend that a city official select *multiple* options from the Pareto front F at the onset. The rationale is that the benefit of high-quality sidewalks in particular areas is highly subjective and may include factors such as the sidewalk quality of nearby blocks that might be strolled during a walk, which would call for a holistic selection of blocks. Needing to select one block at a time, whose selection in turn changes what blocks can be selected next, may induce paralysis by analysis.

We now contrast this method by the current one employed by the City of Ithaca; afterwards, we provide the option of a more formulaic approach that would also improve the current formula in use. Ithaca prioritizes blocks based on (a) population density; (b) proximity to schools, bus stops, governmental buildings; (c) number of complaints; (d) and the physical condition of concrete slabs. Recall that, to determine the Pareto front, we only need to measure the above variables in a way where a greater value corresponds to greater urgency. Thus, we let a_s be the population density in units of people per square mile on block s. Now, for any two $s, s' \in S$, if $a_s > a_{s'}$, then block s has a greater population density than block s'; the converse also holds. Similarly, let b_s be the reciprocal of the shortest distance in miles that an endpoint of s is from a school, bus stop, or government building (we are required to give a subjective definition of *proximity*); c_s, the number of complaints received with respect to sidewalks on s; and d_s, the ranking (perhaps given by a city official) of the worst ADA violation on s out of all possible violations, where a smaller value corresponds to a worse violation. The resultant Pareto front eliminates all blocks where sidewalk improvement would be worse by Ithaca's own standards. The ad hoc formula currently used by Ithaca presumably has this quality as well—if the score they compute increases with an increase in any of the above variables. The score for a block s, for instance, may be given by

$$f(s) = a_s + b_s + c_s + d_s.$$

Then, for a block s that strictly dominates s', we observe $f(s) > f(s')$. However, the converse does not hold. If s and s' have the same values for b and d but $a_s = a_{s'} + 10$ whereas $c_{s'} = c_s + 9$, then effectively having 10 more people per square mile was deemed more important than receiving 9 extra complaints, resulting in $f(s) > f(s')$. This conversion, however, was completely arbitrary. It also assumes that an extra person per square mile is just as important as an extra complaint regardless of the previous quantities of those two values. However, this is not a robust interpretation; if the population density is sufficiently large, an increase in people will likely not make sidewalks more crowded or worn by a larger percentage, whereas an increase in complaints may suggest there is a bonafide safety issue, that the first few complaints were not frivolous. We could ameliorate these limitations by choosing a smooth function f with a suitable Hessian matrix, but we could still find that $f(s) > f(s')$ from an arbitrarily small difference in two variables in opposing directions. Hence, we still recommend that Ithaca city officials select blocks from the Pareto front. Still, since each block is characterized by four variables, it may be difficult to visualize the Pareto front.

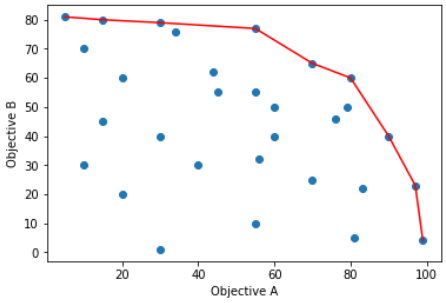


Figure 1: It is relatively easy to compare options in the Pareto set for 2-dimensional data [2]

3.3 Algorithmic Scoring of Blocks

In light of the above, we also provide as an alternative the following algorithm to assign each block $s \in S$ a priority score $f(s)$, with a larger value indicating a greater priority. Select one block $s^* \in S$ from the Pareto front for which it is the most desirable to improve the sidewalk condition. Then, for each $s \in S$, $s \neq s^*$, set $f(s)$ equal to the dot product

$$\mathbf{x}_{s^*} \cdot \mathbf{x}_s,$$

where the variable vectors are with respect to a, b, c, d. Set $f(s^*) = \infty$. Informally, this dot product gives the comparative amount (not proportion) of $\mathbf{x}_s$ that lies in the direction of $\mathbf{x}_{s^*}$, which can be seen by noting that

$$\mathbf{x}_{s^*} \cdot \mathbf{x}_s = ||\mathbf{x}_{s^*}||\,||\mathbf{x}_s|| \cos\theta,$$

where θ is the angle between $\mathbf{x}_{s^*}$ and $\mathbf{x}_s$; $||\mathbf{x}_s|| \cos\theta$ is the scalar projection of $\mathbf{x}_s$ onto $\mathbf{x}_{s^*}$, and $||\mathbf{x}_{s^*}||$ is constant (i.e. for each $s \in S$). The idea is that, since an Ithaca city official chooses s^*, it exemplifies ideal traits that a block receiving sidewalk improvement will posses; blocks s that are similar to s^* vis-à-vis the above variables will, in principle, have similar combinations of the variables (i.e. direction of the variable vector $\mathbf{x}_s$), resulting in a larger value of $\cos\theta$. Moreover, blocks s that have larger values of the variables, loosely representing higher urgency, induce a larger value of $||\mathbf{x}_s||$. Thus, our priority score $\mathbf{x}_{s^*} \cdot \mathbf{x}_s$ tends to increase in both subjective factors regarding how well sidewalk improvement of a particular block would enhance Ithaca and in the somewhat more objective assessment of the extent to which the block would benefit from sidewalk improvement via the four variables above. We can also use the coordinate-wise definition

$$\mathbf{x}_{s^*} \cdot \mathbf{x}_s = a_{s^*} a_s + b_{s^*} b_s + c_{s^*} c_s + d_{s^*} d_s$$

to verify that, since we have defined all of our variables as non-negative, for a block s that strictly dominates s', we have that $f(s) > f(s')$ if $\mathbf{x}_{s^*}$ has no zero component.

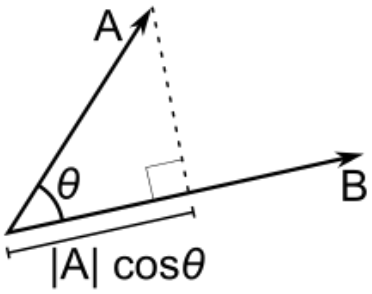


Figure 2: The side of the right triangle adjacent to θ is the scalar projection of $\mathbf{A}$ onto $\mathbf{B}$ [3]

Thus, if blocks are selected for sidewalk improvement iteratively in descending score order, each block selected is most likely Pareto optimal with respect to the set of blocks remaining. This was the main feature that the current ad hoc algorithm ostensibly boasted. Our algorithm also ensures that the block with highest score is the one the city official hand-selected, since all other blocks are assigned a score that is a real number. Because our algorithm merely just modified the coefficients in front of each term of the prospective old scoring formula, however, some of the above criticisms still apply. Notwithstanding, our algorithm enables a much more holistic evaluation of the blocks by not choosing these coefficients arbitrarily; rather, they appear as a byproduct of the geometrically rich interpretation of measuring the alignment of the features of a block with a model block.

3.3.1 Multi-Block Generalization of the Algorithm

Note that the above approach may not allocate sidewalk resources in a way that everyone considers fair. For instance, a city official may *primarily* wish to improve the sidewalks on blocks that have high population density and close proximity to schools and bus stops, i.e. blocks that have a high value for the first and second variables, but some may argue this is unfair as individuals in these areas already have convenience in transportation. There may be blocks in more desolate areas that have a high value for the last two variables (i.e. they have a lot of issues) but whose denizens would very much enjoy walking if not for the problems of their sidewalks. Focusing on an archetypal class of blocks may not ensure that a diverse group of community members are served under the Sidewalk Improvement Plan. Therefore, we offer a modification to the above algorithm for assignment of scores f to blocks. We ask city officials to provide a small list of blocks whose collective individual features exhaust all of the types of blocks they would most like to see improved with respect to sidewalks. For each s^* they select, we set $f(s^*) = \infty$. For all remaining $s \in S$, $s \neq s^*$, set

$$f(s) = \max_{s^*} \mathbf{x}_{s^*} \cdot \mathbf{x}_s,$$

where each vector $\mathbf{x}_{s^*}$ is normalized. This algorithm can be interpreted as preserving the tendency of the above algorithm to assign blocks with larger values for variables a greater priority. However, now we are only interested in comparing the maximum amount to which a block's variable vector lies in the direction of some $\mathbf{x}_{s^*}$, in the sense explained above. As a result, the revised algorithm allows for *multiple* subjective preferences for blocks on which we improve sidewalks. It allows a set of blocks with diverse circumstances to receive a high score, so long as they still satisfy *some* objective criteria given by the four variables. Note that, since the $\mathbf{x}_{s^*}$ are normalized, the maximum dot product for each $s \in S$ depends solely on the magnitude of $\mathbf{x}_s$ and its direction with $\mathbf{x}_{s^*}$, which is unchanged by the normalization. We can again observe that if a block s strictly dominates some block s', most likely $f(s) > f'(s)$. To see this, write $f(s') = \max_{s^*} \mathbf{x}_{s^*} \cdot \mathbf{x}_{s'}$ and $s_+ = \text{argmax}_{s^*} \mathbf{x}_{s^*} \cdot \mathbf{x}_{s'}$. We know that, if $\mathbf{x}_{s_+}$ has no zero component, $\mathbf{x}_{s_+} \cdot \mathbf{x}_s > \mathbf{x}_{s_+} \cdot \mathbf{x}_{s'}$ by our analysis of the previous algorithm, and by the maximization over the s^*, $f(s) \geq \mathbf{x}_{s_+} \cdot \mathbf{x}_s$, so combining inequalities, we obtain $f(s) > f(s')$ if $\mathbf{x}_{s_+}$ has no zero component. Thus, if blocks are again selected in descending score order, each block selected is most likely Pareto optimal with respect to the set of blocks remaining. As a final note, it would be ill-advisable to score the

blocks by some linear combination of the dot products with the $\mathbf{x}_{s^*}$; if the blocks are selected to reflect varied conditions, then such a linear combination may "nullify" the extreme ratios between variable values in some of the variable vectors, possibly degenerating to the prospective ad hoc scoring formula of $f(s) = a_s + b_s + c_s + d_s$, critiqued earlier.

3.4 Time Complexity

Lastly, we perform time complexity analysis of the techniques given above. All three of them generate a Pareto front; to compute this, we can use Kung's algorithm [4]. Given k blocks, each represented by a variable vector in $\mathbb{R}^n$, it computes the Pareto front in time $O(k(\log k)^{n-2})$ for $n \geq 4$ and $O(k \log k)$ for $2 \leq n \leq 3$ (if $n = 1$, we can simply find the maximum element(s) in $O(n)$ by iteration). This is the time complexity of the technique in which city officials choose blocks solely from the Pareto front. For the other two, we must compute $O(k)$ dot products, which entails computing n products and taking their sum. In the revised algorithm, we perform this multiple times and take a minimum, which can be updated as the dot products are computed for each $s \in S$; this merely scales the number of computations by a constant, since we assume that city officials will choose a number of ideal block that is constant in k based on their finite ability to characterize different types of blocks. Thus the two algorithms take time $O(kn)$ to compute the scores for the blocks, so in sum they take at most $O(kn + k(\log k)^{n-2})$. For n sufficiently large, the fraction of time spent computing scores after the Pareto front is generated becomes negligible. Moreover, the two algorithms that do this only require one or several choices from the Pareto front, whereas the technique of choosing all blocks from the Pareto front may take too much time, causing fatigue and sub-optimal solutions. In all cases, since the number of blocks in a city, like Ithaca, would very unlikely exceed a few hundred, all three techniques are extremely quick for n not too large; however, for a city with k fixed, the runtime increases exponentially in the number of variables n. Thus, the number of variables should be limited in using these techniques.

4 Optimal Contracts

4.1 Formalizing the Subproblem and Assumptions

Let us model the network of Ithaca roads and provided list of highest priority blocks as a graph. The set of nodes V of the graph will be the highest priority blocks. We can collect the following data from Google Maps by entering the addresses of the blocks to be repaired to construct two weighted edge sets:

1. For each node $v \in V$, the shortest **time** of travel to every other node $v' \in V, v' \neq v$ following Ithaca roads will be denoted $t(v, v')$
2. For each node $v \in V$, the shortest **path** of travel to other node $v' \in V, v' \neq v$ following Ithaca roads will be denoted $d(v, v')$.

With this data, we can construct two directed graphs $G_t = (V, E_t)$ and $G_d = (V, E_d)$. In both cases, the set of nodes V are the highest priority blocks selected for the given year. In the case of G_t, E_t is the set of all edges between every each $v \in V$ to every other node $v' \in V$, denoted $e_t(v, v')$. Each edge has a weight equal to the distance $d(v, v')$. In the case of G_d, E_d is the set of all edges between every node, only the weight associated with an edge from node v to v' is $t(v, v')$.

Note that the shortest distance and shortest time from block v to some block $v', v' \neq v$ is determined by where Google drops the Red Pin marker when the address is entered into the Google Maps search bar. We use Google's chosen location as a simplifying assumption for the location of any given block and choose to ignore the actual length of the block. This will allow us to utilize the Google Maps data described above as well as run the algorithm we develop below on the graphs G_d and G_t.

We chose to define "groups of nearby blocks" in two ways, depending on whether are are running the algorithm to determine groups of "nearby blocks" on the time of travel weighted graph G_t or the distance weighted graph G_d:

1. In the case of a G_t graph, intuitively speaking, nearby blocks are blocks that take the least amount of time to travel to from the block v where the construction equipment currently is located. Formally, for a G_t graph, the k nearest blocks of some block v in decreasing order are blocks $v_1, \ldots, v_k$ such that $t(v, v_1) < t(v, v_2) < \cdots < t(v, v_k)$

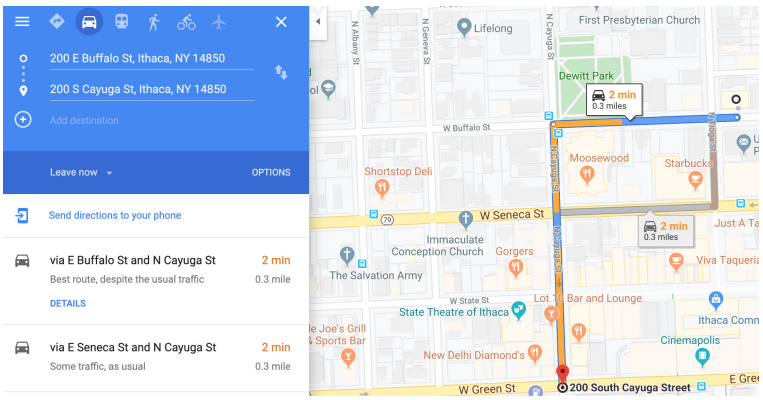
2. In the case of a G_d graph, intuitively speaking, nearby blocks are blocks that are the closest in terms of distance from the block v where construction equipment currently is located. Formally, for a G_d graph, the k nearest blocks of some block v, in decreasing order are the blocks $v_1, \ldots, v_k$ such that $d(v, v_1) < d(v, v_2) < \cdots < d(v, v_k)$.

We assume the cost and time associated with traveling to blocks within a "group of nearby blocks" is negligible. We will assume blocks that are not within the same group cause us to incur a transition cost of 1000 dollars and time penalty of 4 hours, which at a rate of 400 dollars an hour for construction crews on average, is another 1600 dollars.

Assume we have a list $v_1, v_2, \ldots, v_n$ of blocks that are the highest priority blocks to be repaired for a given year. We can search the names of these blocks and use the "get directions" feature of Google Maps with the "travel mode" feature set to car (since construction equipment must travel via roads) to determine the 1) fastest route by time between each block v_i to each other block v_j to obtain $e_t(v_i, v_j)$ and 2) shortest route by distance between each block v_i and other block v_j to get $e_d(v_i, v_j)$ for all i, j. Take for example the list of priority blocks to be repaired on the Sidewalk Improvement Program website:

Proposed Blocks 2020	
200 Block of E Buffalo St	500 Block of W Green St S
1100 Block of N Cayuga St	100 Block of Hawthorne Pl Sect 1
200 Block of S Cayuga St	100 Block of Hawthorne Pl Sect 2

If v_1 is the 200 Block of E Buffalo St and v_2 is the 200 Block of S Cayuga St, then we can enter this information into Google Maps as shown in the figure to obtain $e_d(v_1, v_2) = 0.3$ miles and $e_t(v_1, v_2) = 2$ minutes.



Note that the weights of edges from v_i to v_j will not necessarily be the same as from v_i to v_j, that is it may be possible to have $e(v_i, v_j) \neq e(v_j, v_i)$ for time weighted and/or distance weighted edges.

We can use the Google Maps data to construct two matrices M_d and M_t to represent the weighted directed graphs G_d and G_t, respectively. The dimensions of each matrix will be $n \times n$ where $n = |V|$, the number of highest priority blocks to repair. The entry in the i^{th} row and j^{th} column in matrix M_d is the shortest distance from v_i to v_j i.e. $e_d(v_i, v_j)$. The entry in the i^{th} row and j^{th} column in matrix M_t is the shortest travel time from v_i to v_j i.e. $e_t(v_i, v_j)$. Recall that since it is not necessarily true for $e(v_i, v_j)$ to be equal to $e(v_j, v_i)$, we cannot assume either matrix is symmetric. As an example using the notation defined above, Matrix M_d would then look like

$$M_d = \begin{bmatrix} e_d(v_1, v_1) & e_d(v_1, v_2) & \dots & e_d(v_1, v_n) \\ \vdots & \dots & \ddots & \\ e_d(v_n, v_1) & e_d(v_n, v_2) & \dots & e_d(v_n, v_n) \end{bmatrix}$$

Matrix M_t would be identical only the edge weights e_d would be edge weights e_t.

4.2 Input to Algorithm

Our algorithm takes in as input an $n \times n$ matrix M_t or M_d representing a graph G_t graph or G_d, respectively, where n is the number of the highest priority blocks to repair. The matrix rows and columns are labelled with the block each represents.

4.3 Description of Algorithm

The algorithm used to determine nearby groups of priority blocks can be described in words as follows:

1. Iterate through each row of the node labeled Matrix M (implemented as a Pandas DataFrame in Python with columns values and index values set to the blocks $v_1, \dots, v_n$). For example, if the blocks were a, b, c, and d, then we would have the DataFrame

	a	b	c	d
a	0	12	9	12
b	4	0	6	8
c	13	5	0	1
d	9	7	2	0

Figure 3: Pandas DataFrame with entries being arbitrary distances chosen for the sake of example. Note however that the diagonal must necessarily be all 0s as the distance from any node to itself is 0.

2. Row i of the matrix represents block v_i and each column j is block v_j with the ij entry being $e(v_i, v_j)$. Label the v_j blocks according the column values shown along the top of the DF by creating a list L of tuples with each tuple in the list for Row i taking the form $(v_j, e(v_i, v_j))$ for $j \in \{1, \dots, n\}$.

3. Sort the list of tuples L_{tup} in ascending order according to the value of the second entry in the tuple, $e(v_i, v_j)$.

4. For each block v_i over $i \in 1, \dots, n$, select the k elements in the now sorted list L_{tup} from index 1 to index $k+1$ and place them in a list N_i. These are the k nearest neighbors of block v_i. We start at index 1 in list L_{tup} because the "nearest neighbor" of v_i will always be itself since $e(v_i, v_i) = 0$. We are however only interested in other blocks in V, so we exclude this value when finding nearest neighbors. To clarify, this implies the $k = 1$ nearest neighbor of v_i will be the block $v_j, v_j \neq v_i$ with minimum $e(v_i, v_j)$ over all $j \in \{1, \dots, n\}$.

5. Create the key-value pair (v_i, N_i) and insert it in the dictionary nearestNeighbors.

6. We begin to determine the optimum value of k by calculating the sum of the squared errors (SSE) for the k-nearest neighbor group of each block v_i. We denote the k-nearest neighbors of v_i as $\text{KNN}(v_i, k)$ where $i \in 1, \ldots, n$.

$$avg(\text{KNN}(v_i, k)) = \frac{1}{k} \sum_{j=0}^{k-1} N_i[j]$$

$$SSE(\text{KNN}(v_i, k)) = \sum_{j=0}^{k-1} (N_i[j] - avg(\text{KNN}(v_i, k)))^2$$

7. We sum the SSE over all blocks to obtain the SSE for a given k.

$$SSE(k) = \sum_{i=1}^{n} SSE(\text{KNN}(v_i, k))$$

8. Now that we have the SSE for the k-nearest neighbors of each block for all values $1 \leq k \leq n$, store these computed values in a list L_{SSE}. Let $S = (S_x, S_y) = (1, SSE(1))$ be the first tuple in list L_{SSE} and $T = (T_x, T_y) = (n, SSE(n))$ be the last tuple in L_{SSE}. Let tuple $P_k = (P_{k_x}, P_{k_y}) = (k, SSE(k))$ be the k^{th} tuple in L_{SSE}. Iterating from $k = 2$ to $k = n - 1$, calculate the following quantity for each P_k and append a tuple with the first value in the tuple being k and the second value being the calculated quantity to a list L_{slopes}:

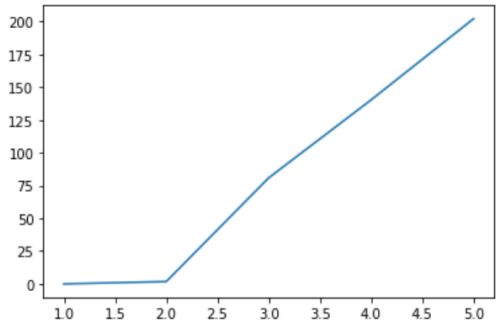
$$\frac{T_y - P_{k_y}}{T_x - P_{k_x}} - \frac{P_{k_y} - S_y}{P_{k_x} - S_x}$$
$$= \frac{SSE(n) - SSE(k)}{n - k} - \frac{SSE(k) - SSE(1)}{k - 1}$$

9. Sort the list of tuples L_{slopes} by the calculated quantities stored in the second position of each tuple. Set k^* equal to the k value in the first position of the tuple with the median of the calculated quantities in its second position. k^* is the optimal number of the nearest neighbors of each block v_i for $i \in 1, \ldots, n$ that constitute the group of nearby blocks of v_i.

10. Run the k-nearest neighbors algorithm described in steps (1) through (5) with $k = k^*$ to obtain a dictionary with the keys being the highest priority blocks to repair and the values for each key (block) being the k nearest blocks to the given key, as defined by the measure of closeness using time of travel t or distance d between blocks. Finally, we print out the k nearest blocks of each block in plain English so the algorithm's results are easily interpretable by whoever may want to implement our plan to begin repairing groups of nearby blocks.

A Python implementation of this entire algorithm can be found in the Appendix section. An example of the output for a test data set is provided:

```
In [32]: points = [(0,0),(1,1),(0.5,0.5),(5,5),(4,5),(4,7)]
         nodes = list(string.ascii_lowercase)[:len(points)]
         M = createM(points)
         df = matrixToDf(M, nodes)
         elbowPlot(df)
```




```
In [34]: optK(df)
Out[34]: 2

In [35]: findNearbyBlocks(df)

         The nearest blocks of a are c, b.
         The nearest blocks of b are c, a.
         The nearest blocks of c are a, b.
         The nearest blocks of d are e, f.
         The nearest blocks of e are d, f.
         The nearest blocks of f are e, d.
```

Figure 4: Elbow Plot with optimal $k = 2$ computed for an example data set and the English description of the 2 nearest blocks to each block

4.4 Conceptual Explanation of Algorithm

The inspiration for using a graph based k-nearest neighbors algorithm was that a group of k nearby blocks intuitively would be the k other blocks that one can travel to from the current block in the shortest amount of time or distance, depending on the measure of closeness.

4.4.1 Use of the Elbow Method

The portion of the algorithm after the k nearest neighbors were determined in steps (1) through (5) is based on the Elbow Method, a heuristic method of interpretation and validation designed to help find the appropriate number of clusters in a data set. Note that the k nearest neighbors algorithm is **not** a clustering algorithm, but rather a supervised classification algorithm. The Elbow Method is typically used to determine the optimal number of clusters k for the unsupervised learning problem of k means clustering. We believe that k means clustering is not applicable here since the algorithm clusters data points near boundaries of learned clusters into separate clusters even though these data points may be closer to one another than many points within the same cluster. An example of this phenomena is shown below:

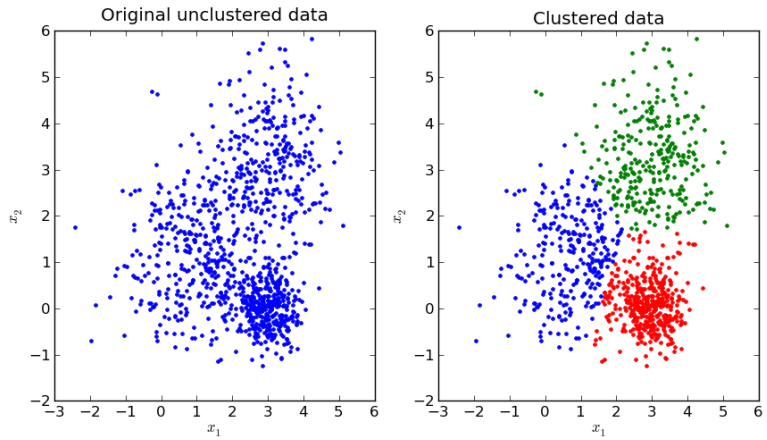


Figure 5: Example of clustering via unsupervised learning that would fail to identify nearby blocks effectively according to our definition of nearby blocks in section 4.1.

Typically, the Elbow Method seeks to identify the optimal k for k means clustering by finding a small value of k that still produces a relatively small SSE, but thereafter has a diminishing marginal decrease in the SSE as k increases:

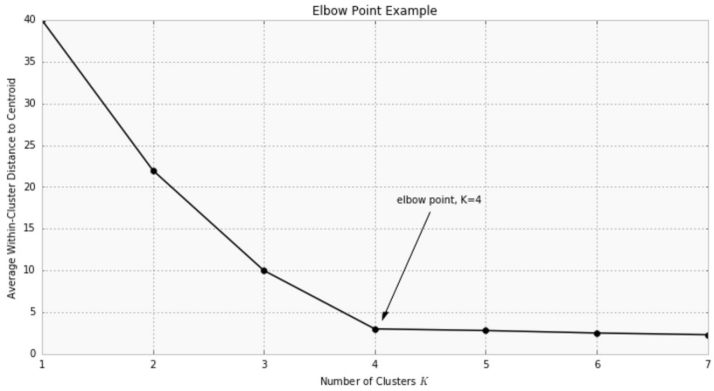


Figure 6: A typical Elbow Plot for k means clustering with the "elbow" point at $k = 4$ clusters

Note that for k nearest neighbors, the shape of the elbow plot will be flipped across the a vertical axis, roughly following the shape of an exponential growth curve rather than exponential decay as in the elbow plot for k means clustering depicts. This is the case because increasing the k in k nearest neighbors by one adds an another n neighbors for which to compute squared error terms for. These additional n terms increase the value of the sum of square errors.

Although the Elbow Method is typically used for clustering, we justify the use of it as the goal of the method is still applicable to k nearest neighbors. By identifying the elbow point for k nearest neighbors, we are finding a trade-off point between SSE and the number of neighbors to include in groups of nearby blocks. Assuming that blocks will not all be equally close to one another, we can conclude that on average, as we increase the number of k nearest neighbors, the variance in the distances to other blocks also increases. Such an increase is reflected in the SSE equation since we sum the square differences of the length from a given block to the nearest k blocks. Increasing k with only a marginal increase in

the SSE intuitively means that we can increase the number of blocks included in each nearby group of blocks at little expense to how far away blocks are from one another within the same group. This trade-off between SSE and k complements our assumption that transition costs between construction sites within the same group of nearby blocks is negligible as well as our assumption that transition costs are incurred when moving from one group of nearby blocks to another group, since we seek to maximize the number of nearby blocks k while keeping SSE relatively low, we can likely complete repairs on all the highest priority blocks by moving the construction crew less with a larger k than if we were to have groups of blocks formed using a small k.

4.4.2 Identifying the Elbow Point

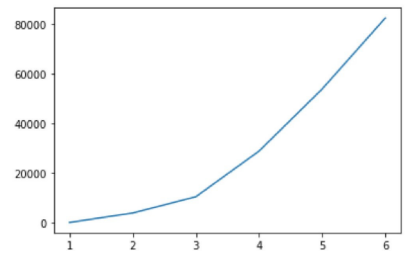
Since the Elbow Method is a heuristic method, the Elbow plot generated by the method is intended to be interpreted by the reader to discover the optimal value of k for themselves. However, as we are doubtful of the Ithaca city officials' ability to understand the trade-offs illustrated by an elbow plot which would allow one to conclude the optimal k, we devised a formula that attempts to determine the optimal value of k without any human intervention.

In order to do this, we created the formula included in section 4.3, copied again here for reference:

$$\frac{SSE(n) - SSE(k)}{n-k} - \frac{SSE(k) - SSE(1)}{k-1}$$

Intuitively, this quantity is a measure of the dissimilarity between the slope of the line from the point with largest k and therefore maximum SSE to point P_k and the slope from this point P_k to the point with smallest k and therefore minimum SSE. Note that this quantity is undefined for $k = 1$ and $k = n$, however this is not an issue for any realistic number of blocks we seek to group since $k = 1$ and $k = n-1$ are the extremes of the number of blocks to a include in a group defined to be nearby blocks, and as such are far from the elbow point where the trade-off between the value of k and SSE is optimized. Therefore, we are not concerned about discarding these values of k from our search space for an optimal k. The validity of this formula can be perhaps more easily be seen by an example:

```
n = 7
df = matrixToDf(np.random.randint(200, size=(n,n)), nodes = range(n))
elbowPlot(df)
optK(df)
```



Although it is clear that there is a large difference in slope from the point with largest SSE to elbow point P and slope from elbow point P to the point with smallest SSE, it is not clear from the plot itself how large this difference is relative to other points. We calculate this quantity iteratively for all values of k in between and notice that for a flat slope of SSE over small values of k and a very quickly increasing slope of SSE over large values of k, these points produce much larger values for the computed quantity based on the formula. Since such extremely small k and extremely large k are far away from where the elbow point (optimal k), I chose to find the median of the list of these calculated quantities to find the k at which the trade-off between k and SSE occurs.

4.5 Time Complexity

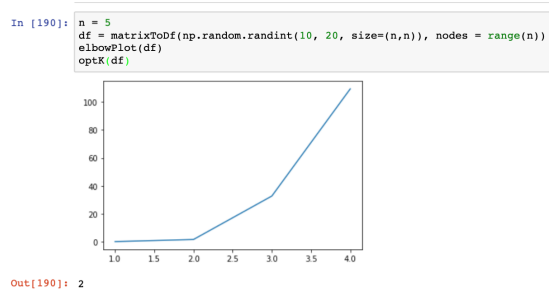
The number of proposed blocks with sidewalks to repair for 2020 on the Sidewalk Improvement website is 39. The algorithm was designed assuming that in the future years, the number of proposed blocks does not increase, i.e. remains on an order of magnitude 2. We believe this is a reasonable assumption since a budget increase of more than 2-fold would likely be needed if the proposed list of highest priority blocks to repair was to increase beyond an order of magnitude of 2. As such a brute-force implementation of k nearest neighbors was created. If there are n proposed blocks to repair and we sort the n blocks by closeness to each of the n blocks for every block, we achieve a time-complexity of $O(n^2 \log n)$, since there are n blocks to iterate through and sorting using `sort` as implemented in Python takes time $O(n \log n)$.

In computing the optimum value of k for the number of nearby blocks in each group, we iterate through values of k from $k = 1$ to $k = n$ and calculate the nearest neighbors for each value of k. Since the nearest neighbors algorithm had a run-time of $O(n^2 \log n)$ and we compute this n times, the run-time complexity becomes $O(n^3 \log n)$. For each iteration, after computing the k nearest neighbors, we must iterate through each block and its nearest neighbors to compute SSE. There will be n blocks to iterate through and at most n nearest neighbors of each block, so this takes at worst $O(n^2)$ time for each iteration. Again, there are n values of k we do this for, so run-time complexity becomes $O(n^3 \log n) + O(n^3) = O(n^3 \log n)$. This run-time dictates the time-complexity of the entire algorithm described in section 4.3, thus we conclude the run-time of the algorithm is $O(n^3 \log n)$.

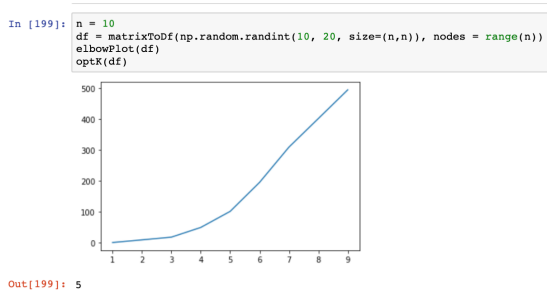
4.6 Further Considerations

4.6.1 Alternatives to the Elbow Method

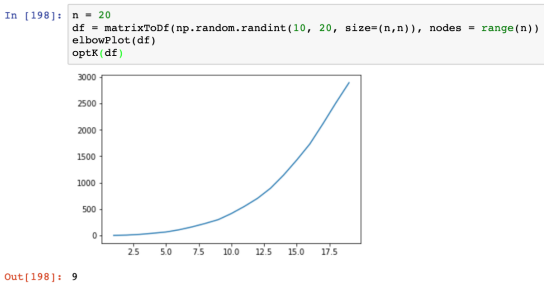
When applied for the purpose of k means clustering, the elbow method can be ambiguous and unreliable [5]. This is the case when the elbow plot has no clear "elbow point", that is there exists no successive values of k for which there is a sharp bend or angle, meaning the elbow cannot be clearly identified via heuristic interpretation. The formula we developed to help unambiguously identify the elbow point and thus the optimal k for our nearest neighbor algorithm to group nearby blocks will tend to return a k value around $n/2$ for uniformly randomly distributed data as the number of data points increases. This is not a flaw in the design of the formula for identifying optimal k, but rather the Elbow method itself because the shape of the curve becomes less and less of an elbow and more like that of a smooth exponential growth curve. To remedy this issue, we can turn to less ambiguous but more complex methods in the future such as the Silhouette Method, which also is typically used for cluster analysis, but perhaps may be re-purposed for k nearest neighbors as we did with the Elbow Method. The Silhouette method is known to provide a succinct graphical representation of how well each object has been assigned a cluster when used to validate the consistency within clusters of data [6]. For our purposes, cluster terminology used in discussion of the Silhouette method would be replaced with references to the groups of the k nearest blocks for each block.



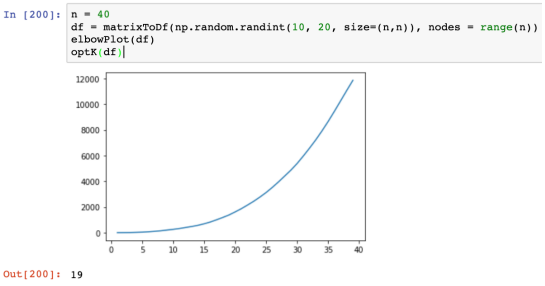
(a) $n = 5$ points with optimal $k = 2$



(b) $n = 10$ points with optimal $k = 5$



(c) $n = 20$ points with optimal $k = 9$



(d) $n = 40$ points with optimal $k = 19$

Figure 7: Elbow Plots for n random uniformly distributed data points between 10 and 20. We can see that as n increases, the curves become smoother and the elbow point becomes increasingly more ambiguous.

5 Optimal Repair Procedures

5.1 Formalizing Subproblem and Assumptions

Given a list which contain a finite number of slabs in a sidewalk represented as $[b_1, b_2, b_3, \ldots b_n] = B$, we wish to repair those which violate clauses (3)-(5) of the ADA while minimizing the cost spent.

Each slab has the same initial area and depth, A and d, and we are given information about the positioning and slope. As the ADA specify each slab should be at least four feet wide, we assume each slab is a square with an area A of 16 feet^2 and depth d equal to eight inches. We assume a thickness of one feet since the City of Ithaca's Code for concrete sidewalk explicitly mentions that " *The concrete sidewalk shall be of one course at least four inches in thickness; and at points where driveways are to be provided, the concrete shall be six inches in thickness.*" If we wish to cut no more than two inches in a slab, then we assume eight inches is the thickness initially in the slab [7]. We assume the cost from a contractor is already considered in the cost for repairing a sidewalk.

The positioning is given through eight three-dimensional coordinates of the corners of each slab, and the slope is given as a tuple with the running and cross slopes. There exists three general strategies for repairing:

1. **Raising**: changing the slope and position of the slab by lifting it by one or two of its corners, with an average cost of implementing as s_{Ra} per square foot.

2. **Cutting**: changing the slope and elevation of the slab by removing a top slice of the slab, with an average cost of implementing as s_C per linear foot.

3. **Replacing**: changing the slope and position of the slab by obtaining a new slab for a new position and slope, with an average cost of implementing as s_{Re} per square foot.

Using the three categories of repairing, we come up with a total of nine strategies. Strategies 1-3 involve raising either one or two corners of a given slab's front corners, where one is lifting the top front right corner, two is lifting the top front left corner, and three is lifting both to maximize level with the previous block. Strategies 4-7 choose to cut the top of a slab up to 0.5, 1, 1.5, or 2 inches, respectfully, since cutting is only possible for removing at most up to two inches. Strategy 8 involves replacing a slab to a position which completely changes the positioning and slope to be an average fit for the two neighboring slabs. Strategy 9 involves leaving the slab as is, which may only be possible if if the slab analyzed abides by the ADA.

We show an example slab in the figure below:

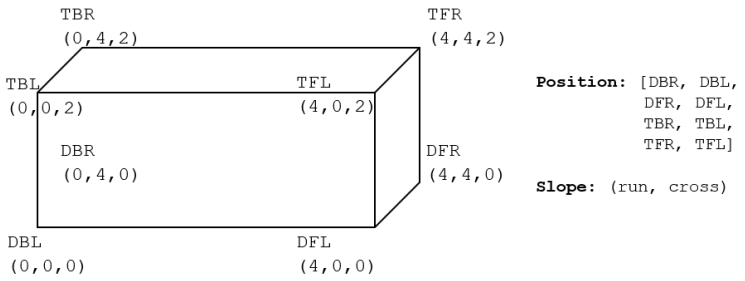


Figure 8: A slab with the eight corner coordinates provided in a list *Position* and the running and cross slopes in a duple. The eight coordinates are abbreviated with example points.

For the city of Ithaca, we are given the following parameters for the costs of repairing shown in Table 1.

Parameter	Value	Units
s_{Ra}	5.13	dollars per feet^2
s_C	16	dollars per linear foot
s_{Re}	22	dollars per feet^2
A	16	feet^2
d	8	inches

Table 1: Parameters for the cost of repair methods.

5.2 Slope Calculation for Repair Strategies in General Example

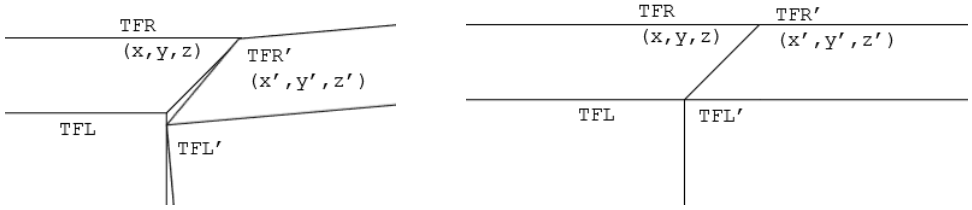
Since the slopes we are given are running and cross that are parallel and perpendicular to the road, respectfully, we are only interested in rotating a slab in the x and/or z axes. The new coordinates after a rotation from cutting or raising are calculated using the known basic three-dimensional rotation matrices seen below. Equations for rotating points along x axis on the left side and z axis on the right side. The three equations below the matrices are the new coordinates calculated in our algorithm.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \qquad R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$x' = x \qquad x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = y\cos(\theta) - z\sin(\theta) \qquad y' = x\sin(\theta) + y\cos(\theta)$$
$$z' = y\sin(\theta) + z\cos(\theta) \qquad z' = z$$

With the use of these formulas, we can calculate a transformation for an x or z coordinate given an angle or vice versa These formulas are used in the implementations of helper functions $rotateX3D$ and $rotateZ3D$ for calculating the transformations and elevations from strategies seen in the Appendix section. *getpt* is also a helper function which gets a specified three-dimensional point.

We now use our formulas mentioned to explain how the angle for rotating along an axis may be determined. Consider in the below, where we see a an adjacent slab that has a cross slope violating the ADA. We wish to find an angle to solve this. Consider looking at points TFR and TFR', with coordinates of (x, y, z) and (x', y', z').



(a) Slab whose cross slope violates the ADA

(b) Slab fixed from raising a corner using formulas shown

Figure 9: General example of fixing a slab using one point.

To find the angle, we consider finding what angles are currently between the two slabs, which we can discovery looking at a possible x and z rotation. We can take our equations mentioned to get

$$x' = x\cos(\theta_z) - y\sin(\theta_z)$$
$$y' = x\sin(\theta_z) + (y\cos(\theta_x) - z\sin(\theta_x))\cos(\theta_z)$$
$$z' = y\sin(\theta_x) + z\cos(\theta_x)$$

where θ_z represents the angle for a rotation in the z axis, and θ_x represents the angle for a rotation in the x axis. We can solve for θ_z and θ_x using WolframAlpha to get

$$\theta_z = 2(\arctan(\frac{\pm\sqrt{y^2 + z^2 - y'^2} - z}{y + y'}) + n\pi), n \in \mathbb{Z}$$
$$\theta_x = 2(\arctan(\frac{\pm\sqrt{z^2 + y^2 - z'^2} + y}{z + z'}) + n\pi), n \in \mathbb{Z}$$

For the given formulas, it is assumed that $y+y' \neq 0$ and $\pm z\sqrt{y^2+z^2-y'^2}+y^2+yy'+z^2 \neq 0$ for θ_z and $z+z' \neq 0$ and $\pm y\sqrt{z^2+y^2-z'^2}+z^2+zz'+y^2 \neq 0$ for θ_x. These equations are still valid despite this limitation as they only occur specific rotations not possible for a three-dimensional coordinate system above the ground. With these angles, we are able to calculate the necessary repair with a specific angle given two coordinates. These angles now known can be used to raise a slab to as seen in Figure 4b, in which using the angles calculated from the formula above may be used to correct the slab.

5.3 Description of Algorithm

Our algorithm $REPAIR$ makes use of helper functions and follows an overall bottom-up approach. We assume we are given slabs in a list called B, representing a sidewalk, which are objects containing its position in three-dimensional space as eight corners and two slopes for running and crossing. The parameters for the city of Ithaca exist in a list $Parm$, which has the general cost for each strategy, area, and depth for the slab. These values would be the same as those presented in Table 1.

We maintain an initial array Ar size 9 by length of the sidewalk, Bm as well as a traceback array Tr the same size. Ar is used for memoization to bottom-up from the first slab up to the last intersection. This way, we can look through all the possible combination of strategies to be able to select the best strategy which seeks to minimize the overall cost. Tr is used in order to traceback strategies used in each index of Ar. After completing the formulation of Ar, a traceback function is run using the Tr and Ar to select the minimum cost given a strategy chosen. The $REPAIR$ algorithm then ultimately returns a list $Strategies$, which contains the best strategy for a block which minimizes overall cost. Initially, all values in Ar are ∞ since we seek to minimize cost overall. and all values in Tr are 0, assuming a valid strategy is not selected for the given corresponding index in Ar.

As our base case, we iterate through all the strategies for the first slab. As we are only looking at a single slab as opposed to an intersection, we are mostly interested in this first case for making sure no violation against the ADA exists for the running and crossing slope exist. These costs for strategies 1-9 are placed in the corresponding index $Ar[s, 0]$, where s represents the strategy s used and 0 represents the first slab. In addition, Tr stores the chosen strategy in the index of Ar at the index of Tr.

In our iterative case, we look at only intersections, We make use of helper of functions $validBlock$, seen in the appendix. to check the two slabs at the intersection if the next satisfies the ADA. For a given strategy s from 1-9 for an intersection b from 1 to length of B, we store the sum of the cost for this strategy with every strategy chosen from the previous slab as a temporary variable, $temp$. In this way, we are looking at every combination of strategies s used from the previous intersection $b-1$ to the strategy s being used for the intersection b looked at presently. For changing the coordinates of a slab, we use helper functions $getpt$, $rotateX3D$, and $rotateZ3D$ to rotate along a given axis and to determine either the coordinates given an angle change or vice versa, as mentioned in the previous subsection. The implementation for these functions can be seen in the Appendix section. Using the $validBlock$ function, we check if the strategies used in making $temp$ result in slabs which maintain and follow the ADA. If the strategies are valid, then the minimum between the value currently in $Ar[s, b]$ and $temp$ are stored at $Ar[s, b]$. This way, we maintain holding the minimum cost for whatever was created for the given slab. In addition, the strategy chosen at $Ar[s, b]$ is updated to be the strategy at $Tr[s, b]$.

Finally, we perform a traceback strategy using our completed Tr and Ar arrays using a helper $Traceback$ function. The best overall strategy which minimizes cost for the sidewalk can be deduced from the values of cells along the traceback path in Tr. We start from the minimum cost of the final intersection, and work backwards to the first slab. At each slab, we calculate the minimum cost and record the strategy used from Tr and record in our return list $Strategies$, which contains the list of strategies to use for the sidewalk. A pseudocode implementation can be seen following:

```
1: function REPAIR(B, Parm)
2:     Initialize array Ar size 9 by length of B with all elements initially ∞
3:     Initialize array Tr size 9 by length of B with all elements initially 0
4:     for s starting from the first up to the strategy 9 do
5:         Calculate the cost of strategies s to repair using Parm and store in Ar
6:         Store strategy chosen at Ar[s, 0] in Tr[s, 0] which match
7:     end for
8:     for b starting from the first intersection up to the last intersection do
9:         for s starting from the first up to the strategy 9 do
10:            Store temp as the sum of cost of strategy s for b using Parm and cost at Ar[s, b − 1]
11:            if the current slab strategies used in creating temp comply with the ADA then
12:                Store at Ar[s, b] the minimum between temp and the current value already in Ar[s, b]
13:                Update strategy chosen at Ar[s, b] in Tr[s, b]
14:            end if
15:        end for
16:    end for
17:    Traceback strategies chosen in Tr using Ar and store strategies chosen in list Strategies
18:    return Strategies
19: end function
```

5.3.1 Time Complexity

Calculating the cost of a repair using our helper functions for changing coordinates takes $O(1)$. The traceback helper function takes $O(s + b)$, as it is linear in the the number of slabs and strategies that exist. Thus, given our double for loop operations, our overall time complexity for s strategies and a sidewalk with b slabs is $O(sb)$ The space time complexity is $O(sb)$ as well since we are filling in a $s \times b$ table.

5.4 Proof of Correctness

We now wish to give a proof of correctness for the algorithm. We prove the correctness of our algorithm by proving our recurrence relation and proving our bottom-up approach for the problem is valid and terminates.

We first seek to prove our recurrences relation is correct. Our base case initializes all the values in Ar to be ∞, and we consistently seek to minimize this value given other strategies found. In the base case of one slab, we view all possible strategies, and we look at the strategy which costs the minimum for the one chosen slab. For all our strategies, they will always result in a cost less than ∞. This means we can easily update a minimum the current value in $Ar[s, b]$ given a s and b in Ar. We can easily return and look at the strategy which produces the minimum cost, as we update in our Tr which strategy is used in Ar.

We now consider an iterative case when at an intersection. With the addition of another slab, we now consider finding the minimum total cost. We explore this by looking at all the strategies which were explored previously in Ar, which means looking at $Ar[0, b-1], Ar[1, b-1], Ar[2, b-1], \ldots Ar[8, b-1]$. We then look at the strategy being explored at the current s and b, so we are able to successfully explore all different possibilities and only store the minimum of all possible values to produce $Ar[s, b]$. We are able to update our Tr with the chosen strategy resulting in the minimum value produced at $Ar[s, b]$.

This ultimately results in an optimal minimum cost, as we are always choosing the strategies which results in the minimum cost when looking at the previous and currently viewed slab. We seek to choose a repair strategy which results in slabs that may follow the ADA, so we are able to ignore strategies easily

that don't abide by the needed conditions for safety. In addition, explanation of choosing a correction angle given two points can be done in constant time as it is not dependent on the size of the slab, so the algorithm is always able to proceed and would not get stuck in finding the cost of a repair. We can consistently make progress based on our memoized matrix and make use of a traceback matrix to deduce the best alignment using our final strategy we chosen from our dynamic programming. Thus, our recursive relation is correct.

We seek to prove that our bottom-up approach for the problem is correct to the recurrence relation. This is because our for loop is choosing to the next strategy before going through each slab, and we make progress in our arrays and consider the minimized cost of the previous and current slabs. Our algorithm always terminates as our for loops are finite based on the limited number of slabs and strategies. Thus, we have proved our bottom-up approach is valid, proving the correctness of our algorithm and that no other algorithm can feasibly do better than ours given our assumptions.

5.5 Further Decreasing Time Complexity

Our algorithm currently runs in $O(sb)$. To decrease our time complexity to $O(sb)/log(b)$ using the Method of Four Russians. The main idea of the method is to partition the matrix into small squares size $t \times t$ for parameter t, and to use a lookup table to perform the algorithm quickly within each square. The index into the lookup table encodes values of the matrix cells on the upper left of the square boundary prior to the operation of the algorithm, and the result of the lookup table encodes the values of the boundary cells on the lower right of the square after the operation. Thus, the overall algorithm may be performed by operating on only $(\frac{n}{t})^2$ squares instead of on n^2 matrix cells, where n is the side length of the matrix. In order to keep the size of the lookup tables small, t is chosen to be $O(log(n))$. We can apply this same logic by partitioning our matrix as well.

6 Implementation in Ithaca's Sidewalk Improvement Program

We have now amassed enough components of a sidewalk improvement plan to apply our findings to Ithaca's corresponding program in a more *concrete* manner. First, we recommend that the City of Ithaca select the blocks they would like to improve or use one of the provided algorithms to compute a score $f(s)$ for each block s, both approaches detailed in Section 3. In either case, this step yields a list of blocks S^*; if one of the scoring algorithms is used, this list should be sorted in descending score order. In order to maximize the enhancement to Ithaca's sidewalks, we iterate through the list S^*, and for each $s^* \in S^*$ encountered, we input the slab data pertaining to s^* into our algorithm $REPAIR$, obtaining a cost $c(s^*)$ and repair strategy $r(s^*)$; we store these for later use and compute a cumulative cost, the sum of the $c(s^*)$ computed thus far. Once this cumulative cost exceeds 865,000 dollars, the sidewalk budget for the year, this process halts, and we have a list S_0^* of all blocks s^* processed before the cost threshold was exceeded, still ordered as in S^*.

Thus, S_0^* is the set of maximum priority blocks of largest size that can theoretically be afforded by Ithaca for the year (if one of the scoring algorithms were used to generate S^*, this is true tautologically; otherwise, we assume that city officials chose blocks from a Pareto front in the order they found them desirable), since we showed our algorithm in Section 5 was optimal. In reality, we must also account for the costs associated with transitioning between faraway blocks during construction. To do this, we provide the blocks in S^* as input to the algorithm in Section 4, which in turn gives an (approximately) optimal number of groups of blocks j, as elaborated therein. Since the groups derived from the algorithm are not disjoint, a distance metric may need to be employed to separate these groups into j disjoint ones, or perhaps a different number of groups. Then, a construction crew repairing all blocks in S^* must then make $j-1$ transitions to faraway sites, incurring a cost of 1000 dollars plus the 1600 dollar cost of the crew for the four-hour process, a total of 2600 dollars each. Then, $2600(j-1)$ is the cost (in dollars) of all of the transitions. The addition of this cost may result in exceeding the total budget of 865,000 dollars

if all the blocks in S_0^* are repaired, since the total cost (in dollars) would be $\sum_{s^* \in S_0^*} c(s^*) + 2600(j-1)$. Therefore, we iteratively delete blocks in S_0^*, decreasing $\sum_{s^* \in S^*} c(s^*)$ by the cost of the removed block and re-running the algorithm in Section 4, obtaining a new number of blocks j' from which we decrease the cost of transitions by $2600(j - j')$; this continues until the revised cost is at most 865,000 dollars. Unfortunately, it is possible that $j' > j$. However, in general we do not expect this to occur, since if only one block remains, we know that $j' = 1$, and since for any positive number of blocks we have that $j \geq 1$, the mean change in j when one block is removed, starting from k blocks until 1 block, must be non-positive. Thus as we remove blocks, the total cost of repairs and transitions generally decreases, and in particular the transition cost is guaranteed to become 0 dollars for some positive number of blocks remaining.

Thus, the revised set of blocks S_0^* after this iterative process must give the highest number of maximum-priority blocks that can be repaired given the budget and under our grouping algorithm. However, this is not fully optimal, since two blocks being within a certain distance is neither necessary nor sufficient for belonging to the same group of blocks, so we cannot make definitive statements about whether a transition cost truly exists within a group or between a group of blocks; the algorithm in Section 4 is a heuristic approach. Now, if $S_0^* = \emptyset$, this would indicate that the cost of repairing just the first street initially in S_0^* exceeded 865,000 dollars; using the data from Section 5, at most this cost would be $22 \cdot 16 = 352$ dollars per slab, implying the block has at least $\lceil \frac{865{,}000}{352} \rceil = 2458$ slabs, which is extremely unlikely. Thus, S_0^* is expected to contain actual blocks s_0^*, grouped for the selected contractor according to the algorithm in Section 4, who in turn is instructed to repair the block with strategy $r(s_0^*)$. We recommend that the contractor is assigned to improve the sidewalks in Ithaca exactly as dictated by these data.

7 Analysis

While each algorithm is developed with an extensive analysis of its properties, we now take a broader look at our methodology.

7.1 Discussion

7.1.1 Discussion of Section 3

Section 3 mainly took an agnostic stance on determining what was optimal in the face of ambiguity, granting city officials the ability to select all of the blocks they wanted to improve or select a sample of such blocks in order to score the remaining ones. This may be effective with city officials that have very strong opinions about which blocks to improve and/or what constitutes an ideal block to improve, but others may not have such convictions and merely want to enhance sidewalks however possible. Even if not, the aforementioned city official of a staunch mindset concerning which blocks to improve may consciously or subconsciously commit an abuse of power wherein they justify selecting their own neighborhood blocks for improvement. Therefore, while the algorithm in Section 3.3.1 is intended to be fairer than the algorithm preceding it, a natural variation of it would entail assigning points to blocks based on the blocks selected by the city officials *with* constraints on the distribution of points, e.g. requiring that the number of points received by each district identified by Ithaca's Sidewalk Improvement Program be roughly commensurate with the population contained therein. Possible efficient ways to incorporate these constraints would be linear programming and maximum flow.

7.1.2 Discussion of Section 4

Section 4 presents an algorithm that attempts to determine groups of nearby blocks in order for city official's to work on nearby blocks that need repairing when a construction crew is at the site of some block already in order to reduce transition costs. A limitation on the output of the algorithm designed

is that there could potentially be much overlap among the the same sizes sets of nearest blocks for each block. Once a block is repaired by a construction crew, we are no longer concerned with the block so it's inclusion in other sets of nearest blocks adds no value, and arguably detracts from the goal of determining an actual plan to handle repairing every block in the list of highest priority blocks. Therefore while the algorithm meets the requested goal of prioritizing groups of nearby blocks to be handled by a single construction crew at some given site, the question of how to go about minimizing overlap among groups of nearest blocks is left unanswered, and as a result an actual optimum course of action could still be developed in which we find disjoint groups of nearby blocks. This could then be suggested to city officials to remove any ambiguity regarding the best course of action in beginning repairs on the highest priority blocks.

7.1.3 Discussion of Section 5

Section 5's algorithm considers the different pair of strategies used in the previous and currently viewed slab and then ultimately selects the repair method which results in the minimum cost overall from the two. This method performs better than a greedy approach of consistently selecting the minimum repair method possible, since there may be circumstances in which we wish we have chosen a different strategy earlier on even if it was the minimum cost strategy at the time. Our algorithm considers a total of nine strategies, which makes our decision to solve each slab limited to those options. We chose a variety of repair options given the three categories mentioned initially to act as general cases which may fix a slab. Replacing a slab is always valid to solve a slab elevation different or slope, but it is also the most expensive strategy so it is only chosen if it results in a minimum cost overall. In the future, devising further strategies in raising and cutting a slab beyond the nine mentioned originally could minimize cost of the sidewalk overall. While this may increase the algorithm's runtime, the opportunity to consider more strategies may result in a lower overall cost that would change one slab's decision, and consequentially its following neighbors. Still, we are able to provide to the Sidewalk Improvement Program definitive answers given our strategies on how to fix a sidewalk.

7.1.4 Discussion of Section 6

Lastly, our implementation of the algorithms pertaining to each section in tandem was shown to be optimal only in an approximate sense. While creating an algorithm to optimally resolve specific subproblems lends generality to our findings, it also limited our ability to optimally resolve the overall problem: allocation of sidewalk resources. In certain political contexts, it may be desirable to solely maximize the number of highest-priority blocks can be served, but as far as offering a subjectively better experience to as many citizens as possible, this maximization is not preferential. For example, one block assigned very high priority may be prohibitively far away from all other blocks of substantial priority; the thousands of dollars spent to transition only to that block may have been usable to repair multiple other blocks that will no longer be able to repaired, whose repairs may have in turn made it more feasible for a construction team to transition to yet another distinct block. As a result, a reciprocal relationship, one involving many trade-offs, exists between the optimization in Sections 3 and 4; in our solution the algorithm in Section 3 merely serves as input for the algorithm in Section 4. Again, since the number of blocks a city will have is a relatively small number, it may not be too expensive to utilize a brute-force approach to ascertain the feasibility of a large number of subsets of S_0^*; though there are exponentially many, the number of subsets of some size k, $\binom{n}{k}$, is $O(n^k)$. We do note, however, that the cost of a block determined in Section 5 depends solely on the slabs pertaining to that block, so it provides invaluable information about the true cost of each block, which are additive, unaffected by any other algorithms employed; in that sense, it retains the optimality statements made about it above.

7.2 Robustness

7.2.1 Robustness of Section 3

Small inaccuracies in measuring the four variables in Section 3 may result in two things with respect to the Pareto front: a block $s \in S$ should not be classified as Pareto optimal but is, and vice versa. In the former, this most likely entails one of the variables was poorly measured, in which case s is similar to some truly Pareto optimal block s', but a small change in one of the variables pertaining to s caused it to no longer be strictly dominated by s'; in this case, s is qualitatively close to a Pareto optimal choice, and thus its inclusion—especially since a city official makes a subjective decision via the Pareto front—is not too harmful. In the latter case, we again expect that s does not differ too much from some block s' in the Pareto front that strictly dominated it under the improper variable measurement, in which case s' would be a viable substitute for s. However, if both s and s' were desirable, it is a limitation that now both of them cannot be selected from the Pareto front. However, if the scoring algorithm in Section 3 is used to select some of the points, this may not be a major limitation, since, using the coordinate-wise definition of the dot product, the score of a block changes linearly with perturbations to one of its variables. Therefore, the methods in Section 3 are generally robust, but systematic errors in measurement may induce an unfair selection for the part(s) of the city affected.

7.2.2 Robustness of Section 4

In Section 4, we assumed given the budget constraints of the city of Ithaca for all side-walk related activities and how many blocks that could be in a list to be repaired for a given year, a naive brute-force algorithm for nearest neighbors and optimal k identification would be adequate. However, it may be the case that Ithaca decides to dedicate more money to side-walk related activities, especially given the fact that the recently re-elected mayor of Ithaca Svante Myrick wants to better infrastructure [8]. As such, we would need to make the algorithm more efficient in order to compute nearby blocks for a larger number of proposed blocks with sidewalks that need repairing.

The run-time complexity can be improved by representing the blocks using the location of the blocks rather than as a matrix M with distances or times describing the location of each block relative to all other blocks. The reason an absolute location was not determined for each block was that Google Maps coordinates are provided in latitude and longitude coordinates and converting from latitude and longitude coordinates to Cartesian Coordinates requires performing the following operations:

$$x = R\cos(lat)\cos(lon)$$
$$y = R\cos(lat)\sin(lon)$$
$$z = R\sin(lat)$$

where lon is longitude, lat is latitude, and R is the radius of the earth if we assume it is roughly spherical. For a small number of blocks to group by nearby blocks, these computations seems unnecessarily complex. However, if we wish to run a more efficient algorithm, a more sophisticated data structure than the matrix M we described in Section 4.1 would be needed. An example of such a structure is the k-d tree, a space-partitioning data structure. Representing the three dimensional (x, y, z) coordinates of each block in a k-d tree, we can compute the k nearest neighbors in expected time $O(Dn \log n)$, where D is the dimensionality of the data, and since $D = 3$, this constant is negligible and we get run time $O(n \log n)$ to find the k nearest neighbors rather than $O(n^2 \log n)$ as we had with the naive brute force search [9].

7.2.3 Robustness of Section 5

The inputs given in Section 5 are the list of positions and slopes for the block and the parameters for calculating the cost of the repair method. For any type of sidewalk provided varying positions and slopes, our algorithm is able to still operate and terminate, as our algorithm would return a more expensive overall cost to repair since a path with wide ranging positions and slopes would need more repairs than a flatter sidewalk. We now discuss how individual changes in parameters may alter the final cost and overall strategy. Our costs for each strategy, s_{Ra}, s_C, and s_{Re}, can strongly influence the decision to choose one repair method over another. If objectively one strategy is better than another usually and has a lower cost than a cheaper but less effective one, then our problem changes where a greedy approach may be best, because the cheapest cost in this case will always result in the best overall strategy and minimum cost. This makes the given parameters for the cost important for the City of Ithaca, as they influence the decision to chose a strategy for multiple slabs. A change in the area wold result in an overall increase in costs for the replacement and raising strategies, as both of these rely on using the area of a slab to determine cost. This wouldn't influence the cutting of a slab, as this repair method cost does not depend on area. However, cutting does depend on the depth of the slab. As we are only able to cut up to two inches and must maintain a six inch slab thickness for driveways, increasing the depth may result in greater elevation differences possible from one slab to another. These greater elevation differences and our limit to cut a slab top up to two inches would result in a diminished efficiency cutting may provide, locking cutting as a possible solution for sidewalks.

8 Conclusion

In this paper, we have explored three different tasks intrinsic to most large-scale public amenities: selection of tasks, organization of tasks, and optimization of tasks. We developed most of the ensuing discussion and models in the context of a network of sidewalks within a city in order to shed particular insight into that system and how it may be fortified. We approached the selection of tasks by avoiding a prescriptive formula to measure the goodness of certain options, instead streamlining a hand-selection process by only presenting city officials with options that are well-aligned with the factors they care about, also giving the option of prioritizing many blocks with minimal input from a human. As for the organization of tasks, we opted to identify nearby blocks using an intuitive nearest neighbors method which returned the closest blocks to a given block for each block using either a time measure of closeness or distance measure of closeness. This choice is left to the discretion of city officials, and this does not influence the way the underlying algorithm operates. To eliminate the need for city officials to understand any details of heuristic interpretation, we created a formula to determine the optimum number of neighboring blocks of each block to include in a group and provide the results of the algorithm to inform which blocks should also be repaired when a construction crew is at any given block. Finally, the optimization of tasks. The granular nature of sidewalks, being that they are composed of uniform slabs, lended themselves to a dynamic programming approach wherein we reason about two slabs at a time, since the compliance of a slab with ADA depends solely on its state and the state of any adjacent slabs, building a repair solution that is always optimal as we walk through the slabs, and hence optimal for the sidewalk as a whole. Though many optimization problems pertaining to a set of tasks are not readily amenable to a dynamic programming solution, we maintain it here due to its unambiguous optimality, its elegance, and its efficiency. While each of these considerations are fairly self-contained, and constitute interesting results in their own right, we coalesce them by offering some strategies to combine them in a more dynamic way, acknowledging the difficulties that arise in balancing the different goals of the algorithms. In doing so, we hope to give the City of Ithaca a means of augmenting the immeasurable benefits of walking by tending to all components of sidewalk improvement.

9 Appendix

References

[1] *United States Census Bureau QuickFacts: Ithaca*. URL: https://www.census.gov/quickfacts/ithacacitynewyork.

[2] Python for healthcare modelling and data science. *Pareto Front*. URL: https://pythonhealthcaremodelling.files.wordpress.com/2018/09/pareto_2.png.

[3] *Scalar Product*. URL: https://en.wikipedia.org/wiki/Scalar_projection#/media/File:Dot_Product.svg.

[4] H.T. Kung. *On Finding the Maxima of a Set of Vectors*. Oct. 1975. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4843823/.

[5] *Elbow Method*. URL: https://en.wikipedia.org/wiki/Elbow_method_(clustering).

[6] *Silhouette*. URL: https://en.wikipedia.org/wiki/Silhouette_(clustering).

[7] Streets Ithaca New York and Sidewalks § 342-49. *Specifications*. URL: https://ecode360.com/8395455.

[8] *Election Night 2019: Myrick handily retains title as Ithaca's mayor*. URL: https://www.ithacajournal.com/story/news/local/2019/11/05/elections-2019-new-york-ithaca-tompkins-county-mayor-svante-myrick-adam-levine/4158192002/.

[9] *K-D Trees and KNN Searches*. URL: https://www.colorado.edu/amath/sites/default/files/attached-files/k-d_trees_and_knn_searches.pdf.

9.1 Code

9.1.1 Section 4 Code

```
# coding: utf-8

# In[21]:


from sklearn.neighbors import kneighbors_graph
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
import random
import pandas as pd
import math
import string


# In[22]:


# Function to sort the list by second item of tuple
# Taken from Stack Exchange
def Sort_Tuple(tup_list):
    return(sorted(tup_list, key = lambda x: x[1]))


# In[23]:


def matrixToDf(M, nodes):
    df = pd.DataFrame(M, columns = nodes, index=nodes)
```

```
    return df

def kNearestNeighbors(df, k=2, flag=0):
    # dictionary with (key = node, value = list of K nearest neighbors) pairs
    nearestDict = {}
    # labels of the nodes
    nodes = list(df.columns.values)

    for rowNum in range(len(df)):
        # distances from the current node
        neighborDistances = df.iloc[rowNum]

        # list of tuples of the form (current node, distance from nodes[nodeNum] to
                                                    current node)
        labelledNeighbors = []
        for i in range(len(neighborDistances)):
            labelledNeighbors.append((nodes[i], neighborDistances[i]))
        sortedNearestNeighbors = Sort_Tuple(labelledNeighbors)
        kNearest = []
        for idx in range(1,k+1):
            # appends the nearest K neighboring nodes to kNearest list
            kNearest.append(sortedNearestNeighbors[idx][flag])
        nearestDict[nodes[rowNum]] = kNearest
    return nearestDict

def kNearestNeighborsAdjMatrix(df, k=2):
    nodes = list(df.columns.values)
    numNodes = len(nodes)
    adjDf = matrixToDf(np.zeros([numNodes, numNodes]), nodes)
    nearestDict = kNearestNeighbors(df, k)
    for node in nearestDict:
        nearest = nearestDict[node]
        for neighbor in nearest:
            adjDf.at[node, neighbor] = 1
    return adjDf


# In[24]:


def optK(df):
    numNodes = len(df)
    sseList = []
    for k in range(1,numNodes):
        # sum the square distance over all K nearest neighbors for each node
        sse = 0
        nearestNeighborsDict = kNearestNeighbors(df, k)
        for node in nearestNeighborsDict:
            listOfNN = nearestNeighborsDict[node]
            distancesOfNN = kNearestNeighbors(df, k, 1)
            avg = np.mean(distancesOfNN[node])
            for neighbor in listOfNN:
                sse += (df.at[node, neighbor] - avg)**2
        sseList.append((k, sse))
    # returns k*, the optimum k
    k_opt = optimumKHelper(sseList)
    return k_opt


# In[25]:


def optimumKHelper(sseList):
    # first point (k=1,sse(k))
    S = sseList[0]
    S_x, S_y = S[0], S[1]
```

```
    # terminal point (k=n, sse(n))
    T = sseList[-1]
    T_x, T_y = T[0], T[1]
    computations = []
    for i in range(len(sseList)):
        P = sseList[i]

        P_x = sseList[i][0]
        P_y = sseList[i][1]

        # prevent division by 0
        if T_x != P_x and P_x != S_x:
            formula = ((T_y - P_y) / (T_x - P_x)) - ((P_y - S_y) / (P_x - S_x))
            computations.append((i, formula))

    sorted_comps = Sort_Tuple(computations)
    # even number of points
    n = len(sorted_comps)
    if n % 2 == 0:
        med = math.floor((sorted_comps[math.floor(n/2)][0] + sorted_comps[math.ceil(n/2)
                                                          ][0]) / 2)
    else:
        med = sorted_comps[int(n/2)][0]
    return med


# In[26]:


def elbowPlot(df):
    numNodes = len(df)
    sseList = []
    for k in range(1,numNodes):
        # sum the square distance over all K nearest neighbors for each node
        sse = 0
        nearestNeighborsDict = kNearestNeighbors(df, k)
        for node in nearestNeighborsDict:
            listOfNN = nearestNeighborsDict[node]
            distancesOfNN = kNearestNeighbors(df, k, 1)
            avg = np.mean(distancesOfNN[node])
            for neighbor in listOfNN:
                sse += (df.at[node, neighbor] - avg)**2
        sseList.append((k, sse))

    plt.plot(*zip(*sseList))
    plt.show()



# In[27]:


def findNearbyBlocks(df):
    k_opt = optK(df)
    # flag = 0 (by default) returns the names of the NN blocks
    namesOfNN = kNearestNeighbors(df, k_opt)
    for node in namesOfNN.keys():
        english = "The nearest blocks of " + str(node) + " are "
        for neighbor in namesOfNN[node]:
             english += str(neighbor) + ", "
        english = english[:-2] + "."
        print(english)


# In[28]:
```

```
n = 12
df = matrixToDf(np.random.randint(1, 30, size=(n,n)), nodes = range(n))
elbowPlot(df)
optK(df)


# In[29]:


findNearbyBlocks(df)


# In[30]:


adjMatrix = kNearestNeighborsAdjMatrix(df)


# In[31]:


def createM(points):
    n = len(points)
    M = np.zeros([n,n])
    for i in range(n):
        p1 = points[i]
        for j in range(n):
            p2 = points[j]
            M[i][j] = math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
    return M



# In[32]:


points = [(0,0),(1,1),(0.5,0.5),(5,5),(4,5),(4,7)]
nodes = list(string.ascii_lowercase)[:len(points)]
M = createM(points)
df = matrixToDf(M, nodes)
elbowPlot(df)


# In[34]:


optK(df)


# In[35]:


findNearbyBlocks(df)
```

9.1.2 Section 5 Code

```
import math
import numpy as np

# gets the 3D point using the x, y, and z vectors
def getpt(x,y,z,pt):
    return (x[pt], y[pt], z[pt])

# does an x axis rotation given the x, y, and z vectors of a point pt
```

```
def rotateX3D(x,y,z,deg, pt):
    x2 = x
    y2 = y*np.array(math.cos(math.radians(deg))) - z*np.array(math.sin(math.radians(deg)
                                                    ))
    z2 = z*np.array(math.sin(math.radians(deg))) +
    y*np.array(math.cos(math.radians(deg)))
    return (x2[pt], y2[pt], z2[pt])

# does an z axis rotation given the x, y, and z vectors of a point pt
def rotateZ3D(x,y,z,deg, pt):
    x2 = x*np.array(math.cos(math.radians(deg))) - y*np.array(math.sin(math.radians(deg)
                                                    ))
    y2 = x*np.array(math.sin(math.radians(deg))) +
    y*np.array(math.cos(math.radians(deg)))
    z2 = z
    return (x2[pt], y2[pt], z2[pt])

# returns true if three conditions for the elevation and slope conditions
of a block are satisfied
def validBlocks(b1,b2):
    if b2.position[2]-b1.position[2] > 0.5:
        return False
    if b2.run > 0.02:
        return False
    if (b2.cross<0.01) or (b2.cross>0.02):
        return False
    return True
```

9.2 Letter for City of Ithaca Sidewalk Program Manager

108 E. Green St.
Ithaca, NY 14850
Dear John Licitra,

As students of Cornell University, we would like to thank you for your dedicated service to the Ithaca community. With the rise of Ithaca as an expanding city, and the reputation it has as a college town to institutions like Cornell and Ithaca College, there is a necessity to maintain safe sidewalks, as they are a major source of transportation. Despite the successful work of your Sidewalk Improvement Program, we believe our team has ways to improve it. After gathering data and discussing optimal plans, we present a trifecta of algorithms to help your Sidewalk Improvement Program select what blocks to fix and how.

We assumed we have a budget of 865,000 dollars for one year, so the outcome of our algorithms give all the sidewalks chosen and strategies for improving them within a given year. We ignore the creation of new sidewalks to focus on using our budget to fix existing infrastructure. Also, we perform repairs only to fix slabs in a sidewalk to comply with the American with Disabilities Act (ADA), assuming all ramps needed are provided and working as well.

We first sought to improve the system's existing selection process of choosing what blocks to prioritize. Unlike the one currently used, our algorithm considers possible trade-offs involved in balancing varying criteria like population density and ADA regulations. We were able to do this by presenting options where no criterion is better off without making at least one preference criterion worse off, and offering you an automated extension to score many blocks quickly. This computerized process is extremely quick, but it could become much slower if we added even a few more variables to track the condition of sidewalks on each block, so we recommend trying to maintain the current variables in use.

Using the prioritized blocks our algorithm finds, we wished to construct groups of nearby blocks to reduce the cost of moving from site to site for a construction crew. With a list of ordered priority blocks, we are able to successfully determine which blocks belong together in the same group and determine the optimal the number of blocks to contain in each group. We do this by considering the trade-off in transition costs that arise with moving equipment from various groups of blocks, which would happen more often for smaller group sizes, and the distance crews would have to travel within the same group of blocks—in order to characterize the blocks as nearby to one another.

Finally, in a given block, we designed an algorithm to determine the best method to fix each slab of a sidewalk. With different strategies like cutting or raising a slab, we are able to efficiently recommend what repair method to use for every slab in a sidewalk by considering all possible combinations of strategies that may be used. Our model is limited by the number of repair options we use, as every slab will get one of our finite number of strategies. However, we provide a wide range of repair options, and with the addition of more strategies that can be included, we can easily incorporate them into our model and provide a better overall plan for improving the sidewalk.

Despite the above limits of our algorithms, we still maintain their usefulness to your Sidewalk Improvement Plan. If you consider using all of our algorithms, however, we must mention that in conjunction they come with additional limitations. This is because of the challenge of making our algorithms work well both individually and as a cohort. Therefore, we leave the option for you to elect a single part of our solution instead of the whole package. We always seek to ensure the best for Ithaca residences, as your team always has. We hope you take our feedback into consideration and reach out to us if you have any comments or concerns.

Sincerely,
Sam Boardman, Ian Paul, and Jeremy Wang