Using MATLAB to Optimize Solutions to the Steiner Tree Problem

Joe Shymanski

May 4, 2023

1 Problem Description

The variation of the Steiner tree problem in this report focuses on finding the minimum spanning tree (MST) for a set of vertices V. These vertices consists of two mutually exclusive subsets: one (F) containing "fixed" vertices and the other (M) containing "moveable" ones. The goal is to place the moveable vertices such that the corresponding MST is as small as possible, meaning the sum of the edge lengths are minimal.

2 Algorithm Structure

I first had to create a function, called `pdist`, to calculate the pairwise Euclidean distances between each of the vertices in the graph. The result would then be used to define an adjacency matrix for the corresponding graph, where the distance between each vertex pair serves as the edge weight. This function already exists in the Statistics Toolbox in MATLAB, but I do not have access to it.

I then created a function, called `steiner_tree`, for building and plotting the MST given a set of vertices and returning the sum of its edge lengths. It utilized the `pdist` function to calculate the adjacency matrix, which was fed into the `graph` object constructor to create the corresponding undirected graph. The `minspantree` function in MATLAB uses Prim's algorithm to find the MST of the `graph` object it receives.

I created a separate function for automatically generating the vertices and labels for each test set I wanted to solve. The `generate_vertices`

function takes in a string corresponding to one of the eight pre-defined testing scenarios along with the number of moveable vertices. It then creates the corresponding F and M sets, with the latter consisting of randomly generated points within the bounds of the points in F.

Finally, the algorithm itself consists of three main loops. The innermost loop passes over each vertex in M and optimizes its location while holding all other vertices fixed. It uses the `fminunc` function from MATLAB's Optimization Toolbox. The surrounding loop will iterate this process a number of times, or until the MST length does not significantly change. Finally, the outermost loop restarts this process a number of times, each iteration having new, randomly generated initial positions for the vertices in M. The best overall solution is saved and plotted upon termination.

3 Parameters and Methodology

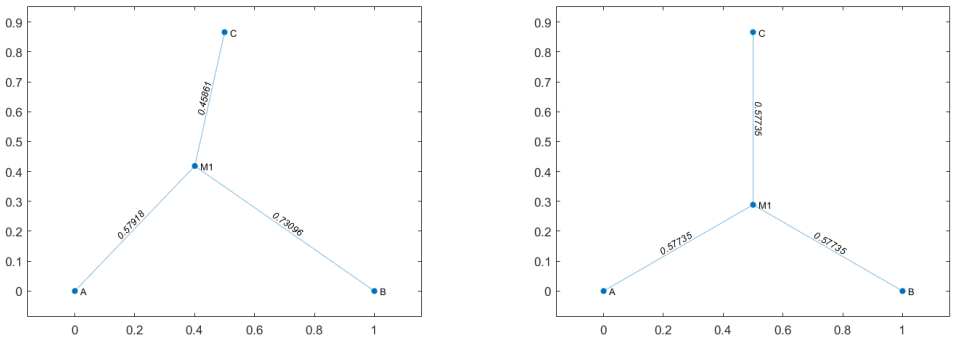
The number of guesses (or restarts) for each test was set to 20, the minimum acceptable change between consecutive MST lengths (ϵ) was set to 10^{-9}, and the maximum number of passes over M was set to 100.

The test domain and number of moveable vertices were the only parameters that varied in these experiments. Each of the eight different test domains featured regular and irregular shapes for the vertices in F, ranging from two-dimensional up to four-dimensional.

This algorithm does not explicitly detect when multiple points sit on top of each other or when more than two points lie on a straight line. Thus, the method I used to determine the best size of M was through incremental increases. Essentially, I would start with $|M| = 1$, obtain a value for the solution, then increment $|M|$ and repeat until the solution value stopped decreasing. So if $|M| = 3$ and $|M| = 4$ yielded the same Steiner tree length, I concluded that $|M| = 3$ was likely the optimal size.

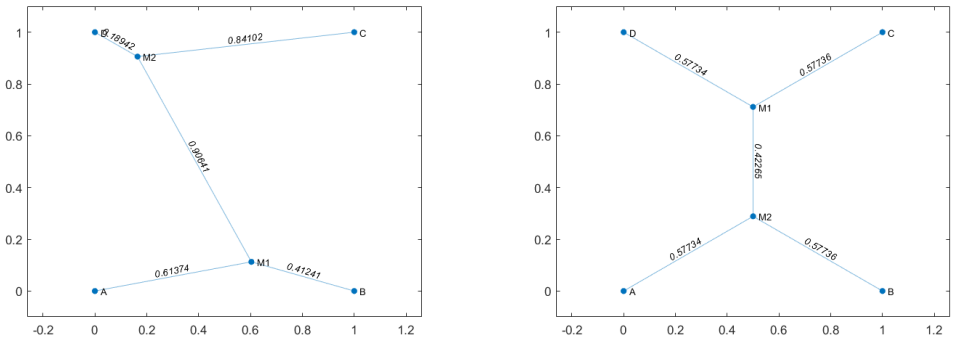
4 Results and Analysis

The first five test sets featured shapes whose side lengths were all one. In order, they were an equilateral triangle, a square, a regular pentagon, a regular tetrahedron, and a cube. Examples of their initial configurations as well as their final solutions are shown in Figures 1, 2, 3, 4, and 5.



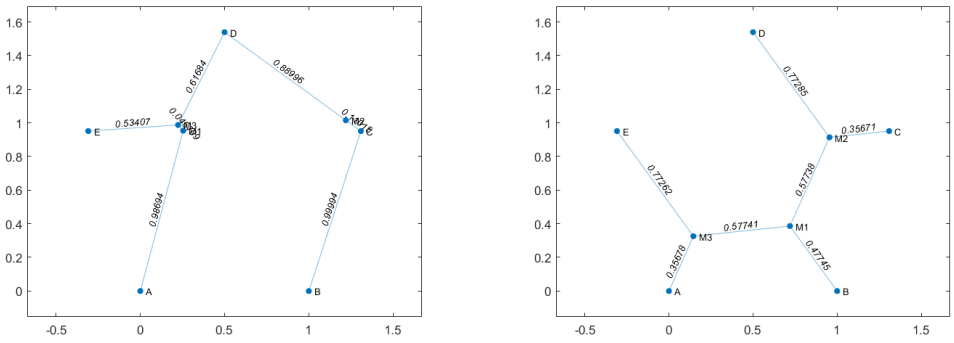
(a) Initial. (b) Final.

Figure 1: Equilateral triangle.



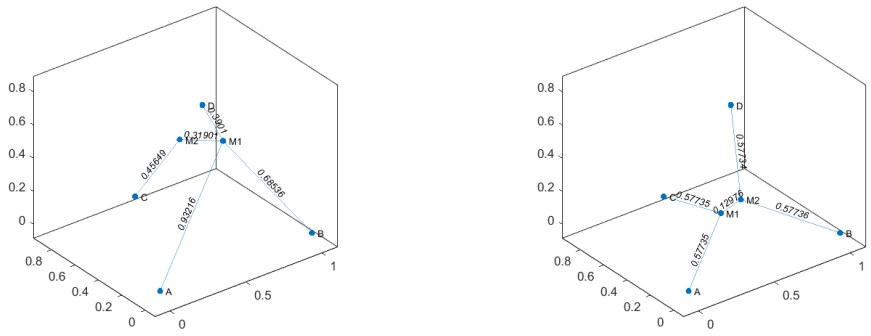
(a) Initial. (b) Final.

Figure 2: Square.



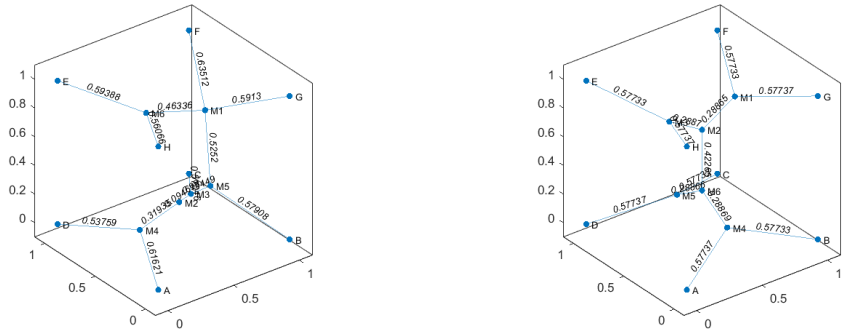
(a) Initial. (b) Final.

Figure 3: Regular pentagon.



(a) Initial. (b) Final.

Figure 4: Regular tetrahedron.



(a) Initial. (b) Final.

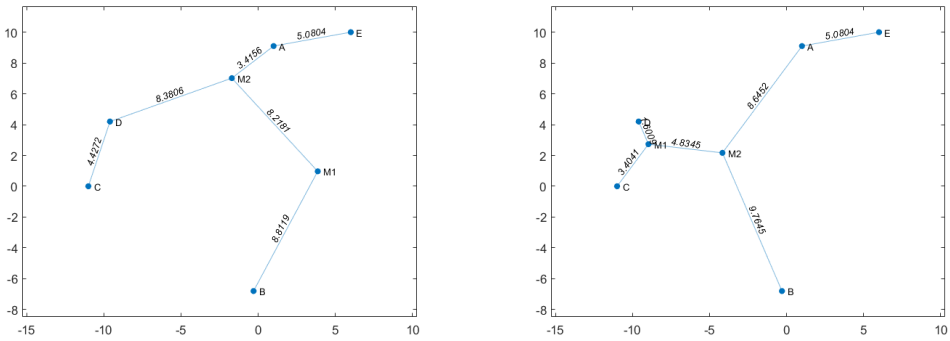
Figure 5: Cube.

Test	$\lvert M \rvert$	S
Triangle	1	1.7321
Square	2	2.7321
Pentagon	3	3.8912
Tetrahedron	2	2.4392
Cube	6	6.1962
Irregular 2D	2	33.3295
Irregular 3D	3	23.1914
Irregular 4D	3	44.2611

Table 1: Solutions

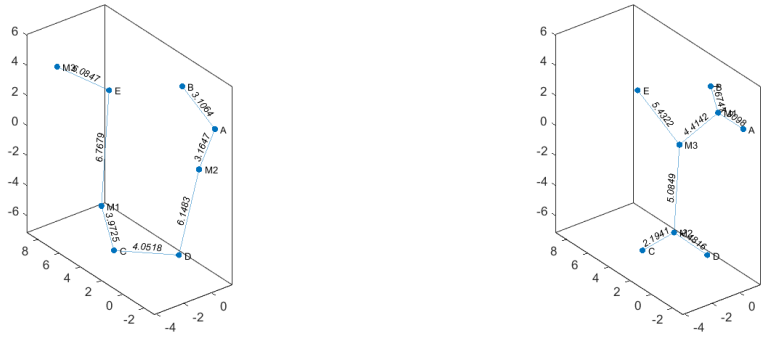
Following the regular shapes, I created three test sets of irregularly placed points in two, three, and four dimensions. Figures 6 and 7 show initial guesses and final solutions for the first two. Unfortunately, we are still unable to properly display four-dimensional images on two-dimensional reports, so the last test set had no accompanying graphs.

Finally, Table 1 shows the values for $|M|$ and the lengths of the final Steiner tree solutions (S). The first three values corroborate the values on the Wikipedia page for the Steiner tree problem. The value found for the unit cube agrees with the value in this Imgur post for 3D Steiner trees. Admittedly, I cannot confirm the reputability of the referenced values, so take these confirmations with caution.



(a) Initial. (b) Final.

Figure 6: Irregular 2D.



(a) Initial. (b) Final.

Figure 7: Irregular 3D.

5 Conclusion

In conclusion, I believe I was able to create a suitable MATLAB algorithm for finding optimal or near-optimal solutions to the Steiner tree problem. The function is not incredibly fast, but has been made as efficient as my abilities allow. All of the solutions were found within a couple minutes, while many took less than 30 seconds. I tested the algorithm's functionality with `fminsearch` but found `fminunc` to run significantly faster while also finding similar solutions. While none of these vertex sets were particularly large, the Steiner tree problem is NP-Hard and has a vast search space for even the smallest test sets.

Additionally, the description I was given for this problem implied the use of weights, one for every edge, along with the length of each edge. I was unsure what the distinction was, and after talking with Dr. Redner, we seemed to come to the common understanding that all weights were equal, boiling down the problem to a minimization of the sum of MST edge lengths. Thinking about the problem now, I could see potential for a weight matrix (paired with the distance matrix from `pdist`) to weight each edge in addition to its length. While this may add interesting complexity to the problem, I ran out of time. I imagine I might be as simple as multiplying the `pdist` matrix by with weight matrix (elementwise) and storing the result as the new adjacency matrix, though this may have unforeseen complications.