Lizzie Healy and Mitali Kessinger
Professor Manning
Linear Optimization
12 May 2023

Minimum Spanning Trees: An Investigation of their Applications and Algorithms

Minimum Spanning Trees

In minimum spanning tree problems, there is a graph consisting of nodes and edges. Each node is numbered, and each edge has a weight, which represents the cost to travel along that edge. A subset of a graph is where every edge or node may not be included, but there are no new lines drawn. The objective of minimum spanning tree problems is to find a subset of the graph such that every node is connected while minimizing the sum of the weights of the edges. In other words, this problem aims to find a part of the graph where every node is connected, but some edges are excluded to reduce the total weight of the edges.

To begin, we provide a few basic facts about minimum spanning trees that are central to understanding the relevant algorithms:

1. According to Cayley's formula, a tree containing n nodes will have n^(n-2) feasible networks
2. A beginning tree with n nodes will have n-1 edges
3. Every two nodes of a tree are connected by a single unique path
4. Adding a non-tree edge to the minimum spanning tree will create a unique cycle
5. A spanning subgraph T of a connected graph G=(V,E) is a spanning tree if any two of the three conditions hold true:
    a. T is connected
    b. T is acyclic
    c. T has absolute value V-1 edges

There are various real world applications of this type of problem. In real applications, the node is often a destination, and edges represent the way to get to a destination. Moreover, the cost of the edge can be used to represent the distance between destinations, the amount of supplies required to get to a destination, or any other cost that may be involved in getting to the destination. One example of an application is providing water to all parts of a building. In this case, each node would be each place that requires water, such as bathrooms and kitchens. Each edge shows the path between two of these locations, and the cost of the edge represents how much pipe materials would be needed to create a pipeline between these two spots. Therefore, the goal would be to find a pipe strategy such that everywhere that needs water gets provided water, while reducing the cost of creating the pipe structure. Similarly, this problem can be used to address connecting energy and wires to different locations. Another type of application is connecting roads within a village. For example, someone may want to generate a network of roads that best connects a whole village including its buildings and important infrastructure, while minimizing the amount of roadway necessary. Thus, each building would be a node and each edge would represent the path to get between locations, with the weight being the amount of road to get between locations.

This type of problem has a very similar setup to max flow problems. Both problems begin with an edge-weighted graph and, using algorithms, determine a substructure that either maximizes or minimizes the weights, respectively. In particular, max flow problems differ in that they are constrained by the need to travel from one specific point to another, whereas minimum spanning trees only require that each node is reached within the subgraph with no predetermined start or end points. However, where these two problems overlap is in their uses and applications, which is the aspect which we will be taking advantage of within our two-step real-world application.

Algorithms

There are two main algorithms used to find minimum spanning trees from an undirected, edge-weighted graph. These two algorithms are Kruskal's algorithm and Prim's algorithm which are named for mathematicians Joseph Kruskal and Robert Prim, respectively. The main difference between these two algorithms is that Kruskal's algorithm deals with the issue of cycling to ensure that each node is only visited once, while Prim's algorithm makes use of vectors to keep track of the nodes that must be visited. Cycling is defined as any connection of nodes within the tree that creates a path to move between the nodes in a continuous manner, or in more basic terms, any connection that creates a loop of nodes. In addition, Prim's algorithm allows for initiation at any node, whereas Kruskal's Algorithm must begin building edges between the two nodes with the smallest weight connecting them. Furthermore, Prim's has a time complexity of O(V^2) and Kruskal's has a time complexity of O(E log V), meaning that Kruskal's runs faster for more extensive trees. Finally, in terms of coding, Prim's algorithm generally works with data in list structures, while Kruskal's works with data in heap structures. Below, we have outlined the basic steps for each of the two algorithms, highlighting the caveats between the two:

Kruskal's

1. Begin with weighted tree and remove all edges
2. Sort edges in nondecreasing order of their weights
3. Remove all edges from graph creating a forest consisting of n-single node trees
4. At each n-1 iterations, pick a minimum-weight edge that does not create a cycle and add it to the graph
5. Continue until each of the vertices are connected in the tree
6. Calculate the minimum weight from the generated minimum spanning tree

Prim's

1. Begin with weighted tree, remove all edges, and create an empty vector of visited nodes
2. Initialize the tree with a random node to begin with and add it to the visited node vector

3. Choose the edge that connects this node to a non-visited node with minimum weight and add it to the visited vector
4. Continue choosing the minimum edge that connects the visited nodes to non-visited vectors until all nodes are visited
5. Calculate the minimum weight from the generated minimum spanning tree

Coding

To delve deeper into the usage and workings of the algorithms we decided to code Prim's algorithm. The code is outlined below:

1. Create a 2D array to map the graph, called G.
    a. Number the nodes 0-n.
    b. Each row of G will represent a different node in the graph, in the order created.
    c. Each column of G will also represent a different node in the graph, in the order created.
    d. Each entry at i, j will represent the cost of the edge between nodes i and j. If there is no path, the cost will be entered as zero.
2. Create variables to keep track of different parts.
    a. N_vert: number of nodes in graph
    b. Selected _node: numpy array of 0s, with the length being equal to the number of nodes. This variable will be used to keep track of which nodes are selected. Set the first index to True, so we have a node to start with.
    c. No_edge: set equal to 0. This shows the number of nodes that have been visited, and will be incremented up as nodes are visited.
    d. Total_weight: set equal to 0. This keeps track of the total cost of our new graph subset. As nodes are added, the cost of the edge to connect with them will be added.
3. Create a while-loop that will repeat the following steps while the no_edge (number of nodes visited) is less than n_vert (the total number of nodes).
    a. Create two variables, vert1 and vert2, which will be set to 0. These will be changed when we find a desirable edge to add to our subgraph.
    b. Create a variable, minimum, which will initially be set to infinity
    c. Find the first selected node, called v1.
    d. Find another node, v2, that is not selected, and has an edge connecting it to v1
    e. If the cost of the edge between v1 and v2 is less than the minimum value, store the new minimum value and set vert1= v1 and vert2=v2
    f. Repeat steps D and E until every edge potential v2 (non-selected node that connects to v2) is considered. Note: this process (D-F) will allow us to find the cheapest edge that connects v1 (selected node) to another node.
    g. Print out the decided edge
    h. Add the weight of the edge to the total cost
    i. Set v2/vert2 as a selected node in selected_node

   j. Increment no_edge (number of visited nodes) up by 1
   k. Loop will repeat until all nodes have been visited and connected by an edge
4. Return total weight. Each selected edge will be printed as it is selected.

In summary, our code will start with one node. It will find the cheapest edge that connects it to a second node. Then, it will consider each visited node to find the next cheapest edge to a non-visited node. This process of adding a node with the cheapest edge will continue until all nodes have been reached.

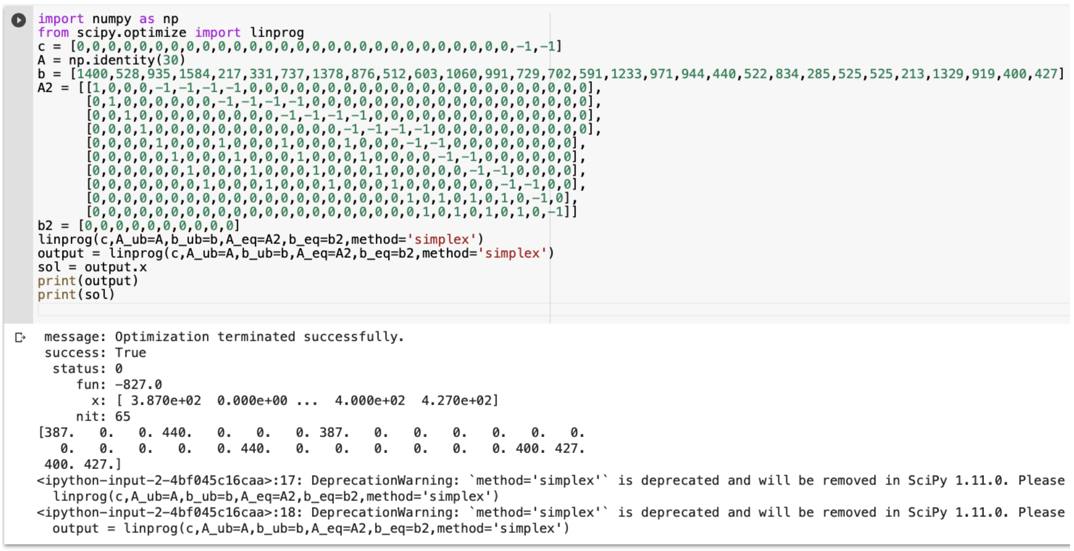
Real-World Application

Max Flow

To relate this type of optimization problem to work from class, we looked into the connections between minimum spanning trees and maximum flow problems. We will utilize maximum flow and linprog for the first portion and then find a minimum spanning tree using Prim's algorithm, using coding for the two. Both of these solutions will be found using the same network and flows, however, we will highlight the difference in interpretation.

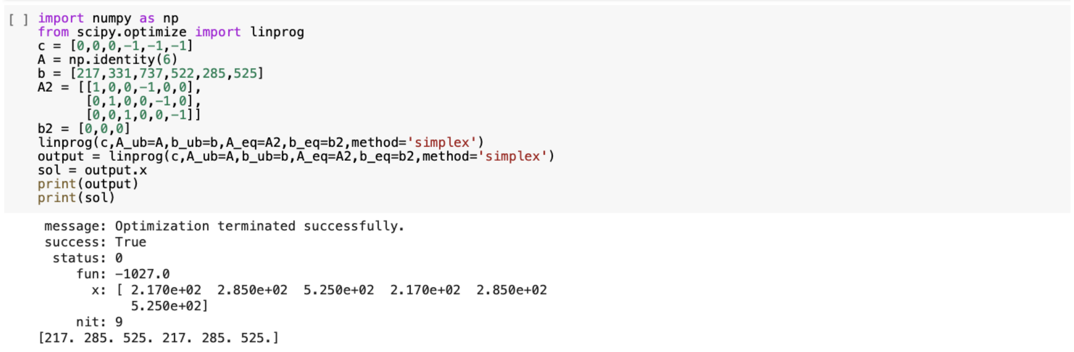
To create this real-world problem we utilized a simple map of Haverford College campus found on google maps. This map is shown in Appendix Image 1. From this map, we create a rough visual representation of the orientation of some of the main buildings on campus and then using the directions applications of google map, we calculated the rough distance between each of the landmarks in terms of walking distance. For clarity's sake and due to the smaller size of the campus, we utilized feet as the unit of measurement to further distinguish each flow value from one other (see image 2).

For the minimum spanning tree, the problem was inserted into code, which is explained below. For the maximum flow problem, we had to make some slight adaptations in order to find something that was more similar to a spanning tree than a path that max flows create. To achieve this we used a series of linprog solutions to create a quasi-snake web design. The fist step was to simply enter our network into a linprog program and allow it to give us its answer for a max flow solution. From this solution, we took the path it gave us from the duck pond to Lutnick (duck pond to tritton to field house to VCAM to Lutnick) and removed it from the graph. We then proceeded to run another max flow with the subgraph, programming it to find a path from Roberts to Lutnick. This gave us a path from Roberts to Barc to Founders to Lutnick. From there we were left with five unvisited nodes which we found the shortest path to each of these using the numbers given and under the condition that we could not create a cycle within the web. This produced a quasi-spanning tree (seen below), however, different from our discovered minimum spanning tree. This tree produced a much higher weight (6,048 compared to 4,646) due to max flow problems looking for the path that allows for the largest amount of flow from the beginning to the end and it is not meant to reach each node. Although creating a spanning tree using linear optimization methods, this highlights the need for Prim's and Kruskal's algorithm. Both of these are able to find a spanning tree much more efficiently and with a much smaller minimum weight.



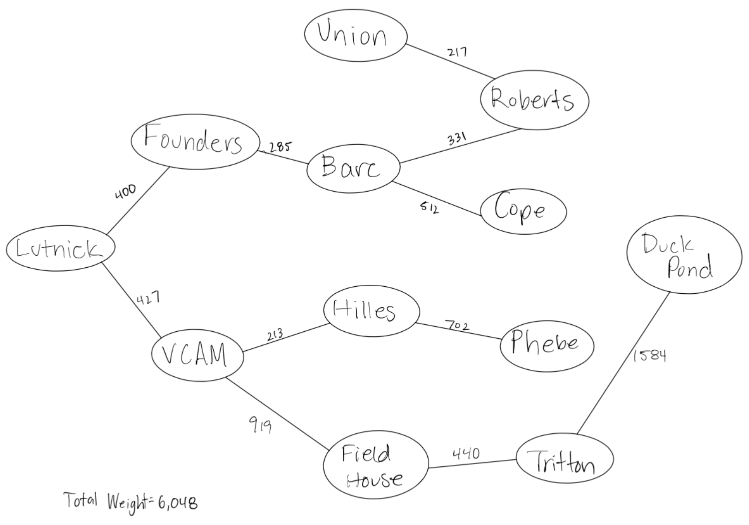
```
import numpy as np
from scipy.optimize import linprog
c = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,-1]
A = np.identity(30)
b = [1400,528,935,1584,217,331,737,1378,876,512,603,1060,991,729,702,591,1233,971,944,440,522,834,285,525,525,213,1329,919,400,427]
A2 = [[1,0,0,0,-1,-1,-1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
      [0,1,0,0,0,0,0,0,-1,-1,-1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
      [0,0,1,0,0,0,0,0,0,0,0,0,-1,-1,-1,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
      [0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,-1,-1,-1,-1,0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,-1,-1,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,-1,-1,0,0,0,0,0,0],
      [0,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,0,-1,-1,0,0,0,0],
      [0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0,0,0,0,-1,-1,0,0],
      [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,1,0,-1,0],
      [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,1,0,-1]]
b2 = [0,0,0,0,0,0,0,0,0,0]
linprog(c,A_ub=A,b_ub=b,A_eq=A2,b_eq=b2,method='simplex')
output = linprog(c,A_ub=A,b_ub=b,A_eq=A2,b_eq=b2,method='simplex')
sol = output.x
print(output)
print(sol)
```

```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: -827.0
       x: [ 3.870e+02  0.000e+00 ...  4.000e+02  4.270e+02]
     nit: 65
[387.   0.   0. 440.   0.   0.   0. 387.   0.   0.   0.   0.   0.   0.
   0.   0.   0.   0.   0. 440.   0.   0.   0.   0.   0.   0. 400. 427.
 400. 427.]
<ipython-input-2-4bf045c16caa>:17: DeprecationWarning: `method='simplex'` is deprecated and will be removed in SciPy 1.11.0. Please
  linprog(c,A_ub=A,b_ub=b,A_eq=A2,b_eq=b2,method='simplex')
<ipython-input-2-4bf045c16caa>:18: DeprecationWarning: `method='simplex'` is deprecated and will be removed in SciPy 1.11.0. Please
  output = linprog(c,A_ub=A,b_ub=b,A_eq=A2,b_eq=b2,method='simplex')
```



```
import numpy as np
from scipy.optimize import linprog
c = [0,0,0,-1,-1,-1]
A = np.identity(6)
b = [217,331,737,522,285,525]
A2 = [[1,0,0,-1,0,0],
      [0,1,0,0,-1,0],
      [0,0,1,0,0,-1]]
b2 = [0,0,0]
linprog(c,A_ub=A,b_ub=b,A_eq=A2,b_eq=b2,method='simplex')
output = linprog(c,A_ub=A,b_ub=b,A_eq=A2,b_eq=b2,method='simplex')
sol = output.x
print(output)
print(sol)
```

```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: -1027.0
       x: [ 2.170e+02  2.850e+02  5.250e+02  2.170e+02  2.850e+02
            5.250e+02]
     nit: 9
[217. 285. 525. 217. 285. 525.]
```



Coding Solution of Real World Problem

Going back to our coding steps outlined above, first we must create a graph G. We decided upon an order for the various nodes to be stored, and recorded the cost of paths between each node.

```
#Node order: [Duck Pond, Roberts, Cope, Phebe, Tritton, Union, Barc, Hilles, Field House, Founders, VCam, Lutnick]

#Duck Pond: [0, 1400, 528, 935, 1584, 0, 0, 0, 0, 0, 0, 0]
#Roberts: [1400, 0, 0, 0, 0, 217, 331, 737,1379, 0, 0, 0]
#Cope: [528, 0, 0, 0, 0, 876, 512, 603, 1060, 0, 0, 0]
#Phebe: [935, 0, 0, 0, 0, 991, 729, 702, 591, 0, 0, 0]
#Tritton: [1584, 0, 0, 0, 0, 1233, 971, 944,440, 0, 0, 0]
#Union: [0, 217, 876, 991, 1233, 0, 0, 0,0, 522, 834, 0]
#Barc: [0, 331, 512, 729, 971, 0, 0, 0, 0, 285, 525, 0]
#Hilles: [0, 737, 603, 702, 944, 0, 0, 0, 0, 525, 213, 0]
#Field House: [0, 1378, 1060, 591, 440, 0, 0, 0, 0, 1329, 919, 0]
#Founders: [0, 0, 0, 0, 0, 522, 285, 525, 1329, 0, 0, 400]
#VCAM: [0, 0, 0, 0, 0, 834, 525, 213, 919, 0, 0, 427]
#Lutnick: [0, 0, 0, 0, 0, 0, 0, 0, 0, 400, 427, 0]



G = [[0, 1400, 528, 935, 1584, 0, 0, 0, 0, 0, 0, 0],
     [1400, 0, 0, 0, 0, 217, 331, 737,1379, 0, 0, 0],
     [528, 0, 0, 0, 0, 876, 512, 603, 1060, 0, 0, 0],
     [935, 0, 0, 0, 0, 991, 729, 702, 591, 0, 0, 0],
     [1584, 0, 0, 0, 0, 1233, 971, 944,440, 0, 0, 0],
     [0, 217, 876, 991, 1233, 0, 0, 0,0, 522, 834, 0],
     [0, 331, 512, 729, 971, 0, 0, 0, 0, 285, 525, 0],
     [0, 737, 603, 702, 944, 0, 0, 0, 0, 525, 213, 0],
     [0, 1378, 1060, 591, 440, 0, 0, 0, 0, 1329, 919, 0],
     [0, 0, 0, 0, 0, 522, 285, 525, 1329, 0, 0, 400],
     [0, 0, 0, 0, 0, 834, 525, 213, 919, 0, 0, 427],
     [0, 0, 0, 0, 0, 0, 0, 0, 0, 400, 427, 0]]
```

Next, we created the variables described above to help store various information.

```
n_vert = 12

selected_node = np.zeros(n_vert)

no_edge = 0

selected_node[0] = True

total_weight=0
```

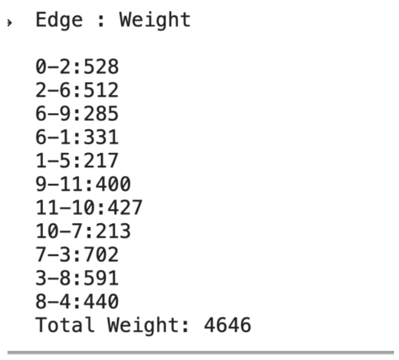
Then, we ran our loop, as described above.

```
print("Edge : Weight\n")
while (no_edge < n_vert - 1):

    minimum = INF
    vert1 = 0
    vert2 = 0
    for v1 in range(n_vert):
        if selected_node[v1]:
            for v2 in range(n_vert):
                if ((not selected_node[v2]) and G[v1][v2]):
                    # not in selected and there is an edge
                    if minimum > G[v1][v2]:
                        minimum = G[v1][v2]
                        vert1 = v1
                        vert2 = v2
    print(str(vert1) + "-" + str(vert2) + ":" + str(G[vert1][vert2]))
    total_weight += G[vert1][vert2]
    selected_node[vert2] = True
    no_edge += 1

print("Total Weight: " + str(total_weight))
```

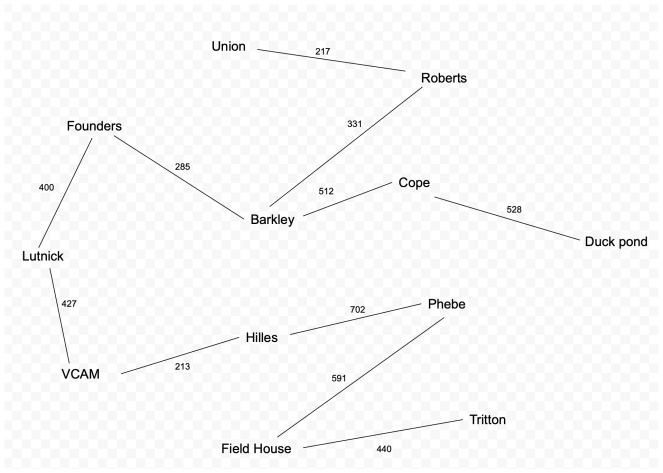
We got the following output, showing which edges were selected and the total weight.



```
Edge : Weight

0-2:528
2-6:512
6-9:285
6-1:331
1-5:217
9-11:400
11-10:427
10-7:213
7-3:702
3-8:591
8-4:440
Total Weight: 4646
```

This can be then interpreted as a graph, which we have drawn below. Note: “0-2” indicates that we have chosen the edge between nodes 0 and 2. Going back to our node-order, we know this is Duck Pond and Cope.



Appendix

Link to code (shared as well):
https://colab.research.google.com/drive/1nQDKyqIcrczj8Ujj9hc7tAgvpcofWp8m#scrollTo=wGMi-EOElKXR

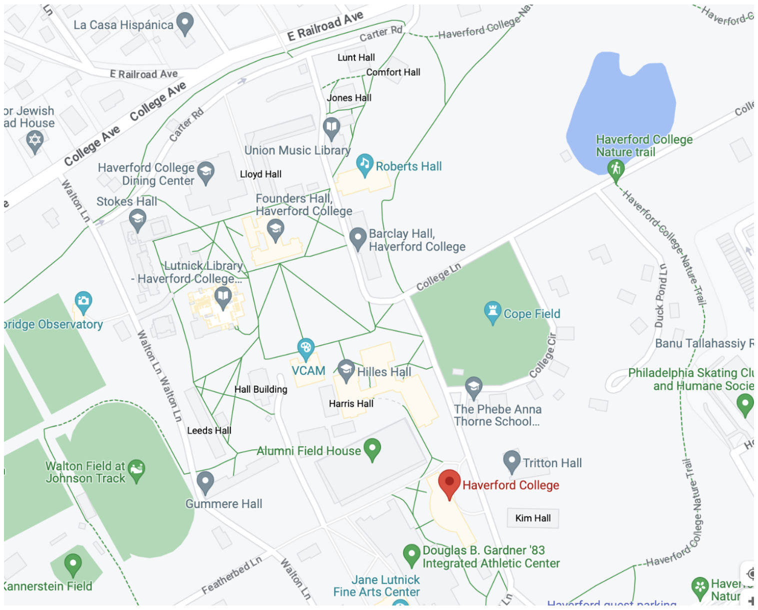

La Casa Hispánica
E Railroad Ave
Carter Rd
Haverford College Nature trail
Lunt Hall
Comfort Hall
Jones Hall
College Ave
Union Music Library
Roberts Hall
Haverford College Dining Center
Lloyd Hall
Stokes Hall
Founders Hall, Haverford College
Barclay Hall, Haverford College
Lutnick Library - Haverford College...
College Ln
Cope Field
Walton Ln
VCAM
Hilles Hall
Hall Building
Harris Hall
The Phebe Anna Thorne School...
Leeds Hall
Alumni Field House
Walton Field at Johnson Track
Gummere Hall
Tritton Hall
Haverford College
Kim Hall
Douglas B. Gardner '83 Integrated Athletic Center
Jane Lutnick Fine Arts Center
Featherbed Ln
Duck Pond Ln
Banu Tallahassiy
Philadelphia Skating Club and Humane Society

Image 1: Haverford College Campus Map

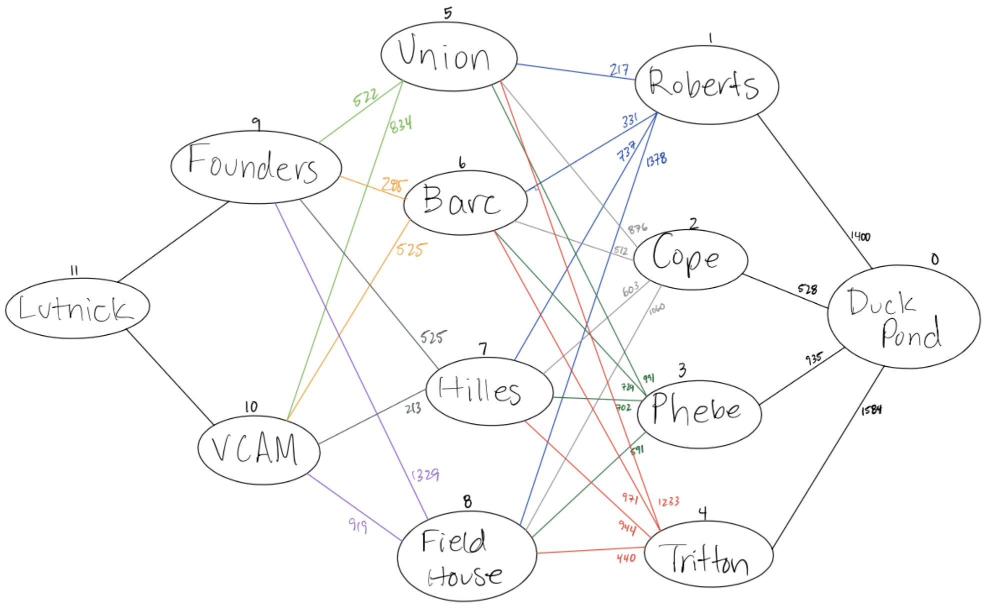


Image 2: Example drawn network flow developed from map information