Underground Tunnels

Jane Hwang

Elaine Lee

Zhi Wei Leong

Department of Mathematics

Carnegie Mellon University

Pittsburgh, PA 15213

Table of Contents

Background

Every year, Pittsburgh winters have Carnegie Mellon students and faculty griping about the cold, snow, and slippery roads. We believe that in order to improve the overall campus experience, it will be beneficial for the school to have underground tunnels linking campus buildings together to provide shelter during poor weather conditions. Other universities in the United States have pedestrian tunnel systems implemented for this specific purpose. Thus we believe a pedestrian tunnel system would be justifiable and viable for the Carnegie Mellon campus community.

Objective

To begin investigating how to build a pedestrian tunnel system at Carnegie Mellon, it is important to first determine the concrete goals that the tunnel system must accomplish, as well as the limitations associated with constructing it. The goal of our project is to find the optimal distribution of tunnels among the campus buildings such that minimizes the construction fee while keeping the following constraints.

Constraints

The pedestrian tunnel system must satisfy the following construction considerations:

- Every major campus building must be reachable from any other major campus building via a series of pedestrian tunnels and/or connected buildings.
- The size of each tunnel must accommodate the volume of human traffic traveling through it at any given time.
- The locations of the tunnels somehow reflect the preferences of the Carnegie Mellon campus community.

Methodology

To find the optimal way of distributing tunnels among the campus buildings, a strategic approach to the problem is required. The general methodology to this optimization problem is summarized in Figure 1. First, the quantitative data for the traffic flow was collected. After obtaining the data for the traffic flow, the Kruskal's algorithm was applied to the graph to produce the initial input for the flipping algorithm. Then, the optimal solution was obtained by applying the flipping algorithm.

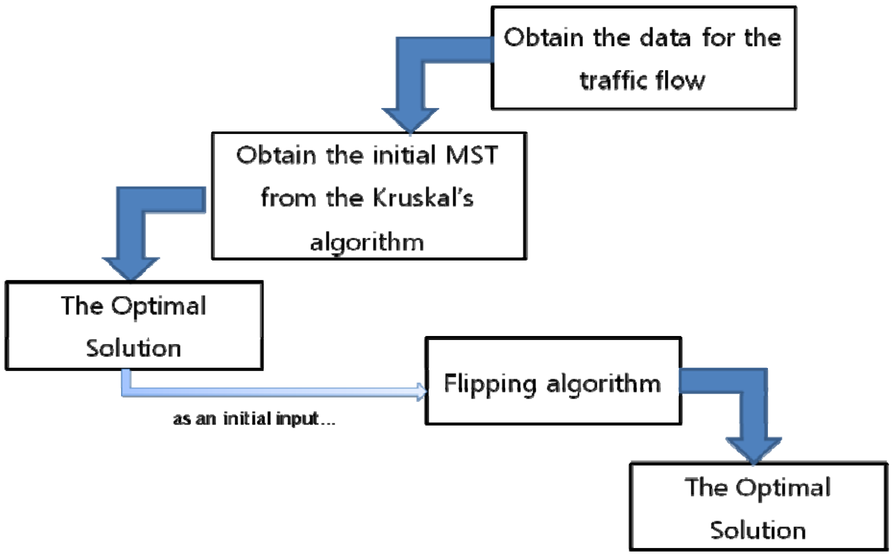

Obtain the data for the traffic flow
Obtain the initial MST from the Kruskal's algorithm
The Optimal Solution
Flipping algorithm
as an initial input...
The Optimal Solution

Figure 1. the schematic diagram of the overall approach to the problem

Modeling as a Minimum Spanning Tree

Based on the objectives that were established, it resembles a minimum spanning tree problem, a very well-known problem in operations research and computer science.

Modeling the construction problem as a minimum spanning tree problem, the campus map is represented by the graph such that:

- A vertex represents one building.
- If two or more buildings are already connected, they are treated as one vertex (e.g., Doherty Hall, Wean Hall, and Newell-Simon Hall)
- An edge is a possible tunnel connecting two buildings.

- A weight for the edge will be the cost associated with building a tunnel along that edge.

Locations

The major campus buildings selected were buildings that the Carnegie Mellon campus community frequents regularly. Most of the buildings on campus were selected, with the exception of facilities management buildings and satellite buildings for academic departments (like math and humanities). Some buildings were grouped together and treated as one location, such as the majority of the Greek quadrangle and the dormitories on the Hill on Margaret Morrison St.

The tunnel locations were selected based on personal experiences and observation about pedestrian traffic patterns. The selection is by no means adequate for the study, because it is biased, but it is adequate as a starting point for this study. Later, campus community input was gathered regarding tunnel locations.

Weight

Once tunnel locations and edges were determined, weights needed to be assigned. The weight should be directly proportional to the volume of the tunnel. Since the volume was proportional to the length of the tunnel as well as the expected traffic flow, the weight of each edge should be proportional to the product of the length of the edge and the expected traffic flow. In other words, a longer tunnel costs more to build than a shorter tunnel; a wider tunnel (heavier traffic) costs more to build than a narrower tunnel (less traffic). Additionally, the minimum spanning tree algorithm should favor tunnel locations according to the campus community's preferences. In order to do this, the weight of an edge should somehow be reduced if the edge is highly in demand. Of course, the cost guidelines governing tunnel volume still apply. These considerations have taken into account the costs of building a tunnel while attempting to aid the minimum spanning tree algorithm in finding a socially optimal solution. The following function for assigning a weight to each edge was developed:

$$\text{Weight} = \begin{cases} \dfrac{\text{Length}}{\text{flow}^2} \times 100, & 0 \leq \text{flow} < 30 \\ \dfrac{\text{Length}}{\text{flow}} \times 100, & 30 \leq \text{flow} < 60 \\ \dfrac{\text{Length}}{\sqrt{\text{flow}}} \times 100, & 60 \leq \text{flow} < 100 \end{cases}$$

The function for assigning weight is piecewise to closely resemble the reality of scaling construction projects – size is only increased in discrete increments rather than in continuous increments because of standardized parts. Similarly, the cost increases in discrete increments as well. The Weight function is broken down by flow through the edge. Each piece of the Weight function reflects the guidelines proposed: Weight is directly proportional to length and yet inversely proportional to flow so the minimum spanning tree algorithm can take the campus community's preferences into consideration. Weight is piecewise to account for increasing the width of the tunnel based on traffic flow. It should be noted that there is no "start-up" cost in the weight function. Since the allocation of tunnels is based on a minimum spanning tree and the number of vertices is kept constant throughout the analysis, the number of edges representing the tunnels built is a constant (number of vertices – 1) regardless of the choice of tunnels. Hence, the start-up cost is constant and thus disregarded.

Obtaining Data Regarding Traffic Flow

Campus Map

An estimation of traffic flow throughout campus was required for the Weight function. Initially, data was obtained from Carnegie Mellon's Office of Institutional Research & Analysis (Figure 2). The data is in the form of a campus map with pedestrian paths marked in varying degrees of thickness according to popularity. Thicker lines imply a heavily used path. While the data provides some idea of traffic flow, it is inadequate in terms of providing quantitative information. Thus, another approach was needed to obtain quantitative estimates of traffic flow.

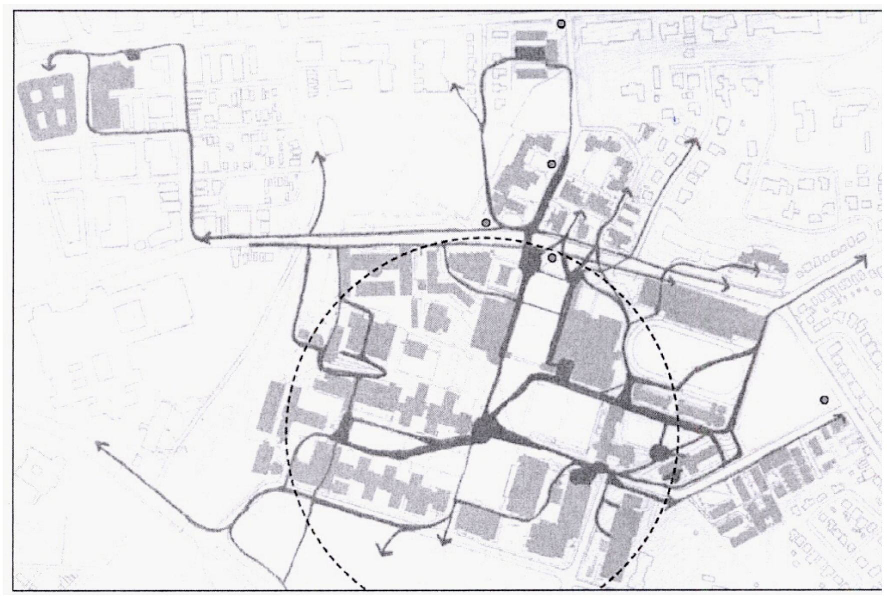
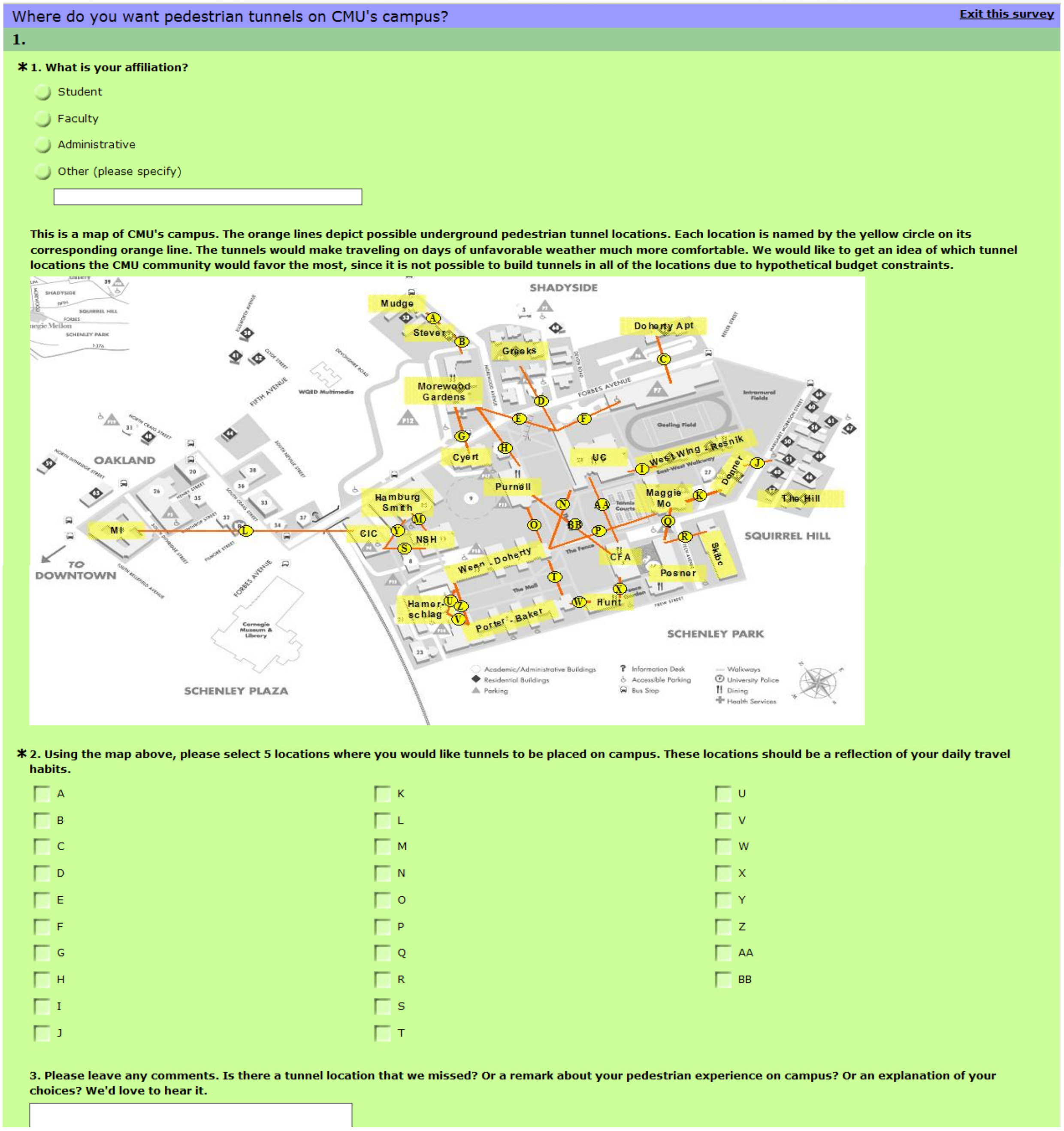


Figure 2. The graphical representation of the traffic flow on the campus[1]

Online Survey

An online survey was used to gather input from the campus community about pedestrian traffic patterns. The overall layout of the questionnaire is shown in Figure 3. The survey asks for the affiliation of the survey taker, the survey taker's five tunnel location preferences among the tunnel locations listed, and comments. It is assumed that the tunnels selected by the survey takers are accurate reflections of a typical daily walking route. The survey was posted to cmu.misc.market, an online bulletin board on the internal Carnegie Mellon network that is heavily viewed by all Carnegie Mellon members. 100 responses were collected. The votes for each edge were tabulated and the result is presented in Appendix A. Additionally, suggestions for additional edges were tallied. The flow of an edge is the number of votes received by that edge. The figures were unaltered and directly applied to the Cost function. The revised tunnel location map based on survey takers' input can be viewed in Appendix B along with the graph representing the original map used in the survey. After the survey, one more vertex (Oakland Apartment) and 11 more edges were added to the graph. Thus, the graph contains 24 vertices and 39 edges in total.



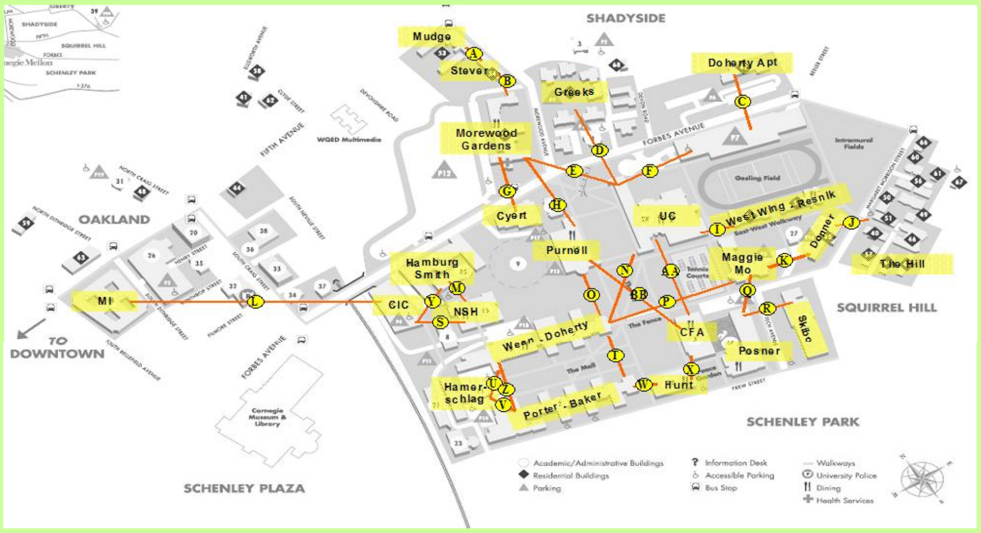
Where do you want pedestrian tunnels on CMU's campus? Exit this survey

1.

*** 1. What is your affiliation?**

- Student
- Faculty
- Administrative
- Other (please specify)

This is a map of CMU's campus. The orange lines depict possible underground pedestrian tunnel locations. Each location is named by the yellow circle on its corresponding orange line. The tunnels would make traveling on days of unfavorable weather much more comfortable. We would like to get an idea of which tunnel locations the CMU community would favor the most, since it is not possible to build tunnels in all of the locations due to hypothetical budget constraints.



*** 2. Using the map above, please select 5 locations where you would like tunnels to be placed on campus. These locations should be a reflection of your daily travel habits.**

☐ A	☐ K	☐ U
☐ B	☐ L	☐ V
☐ C	☐ M	☐ W
☐ D	☐ N	☐ X
☐ E	☐ O	☐ Y
☐ F	☐ P	☐ Z
☐ G	☐ Q	☐ AA
☐ H	☐ R	☐ BB
☐ I	☐ S	
☐ J	☐ T	

3. Please leave any comments. Is there a tunnel location that we missed? Or a remark about your pedestrian experience on campus? Or an explanation of your choices? We'd love to hear it.

Figure 3. A screenshot of the online survey questionnaire

Minimum Spanning Tree Algorithm

Kruskal's Algorithm

With the campus modeled as a graph whose edges are the possible tunnel locations with the associated weight, the minimum spanning tree algorithm can be applied. Kruskal's algorithm is a greedy algorithm for finding the minimum spanning tree for a connected, weighted graph[2].

The Kruskal's algorithm finds the minimum spanning tree in the following way[2]:

- For a given graph, the set containing all the edges in the graph is created.
- Edges are quick-sorted based on their weight.
- Starting from the edge with the smallest weight, the edge is added to the graph if and only if the addition of it does not create a cycle in the graph. Otherwise, the edge is discarded.
- The algorithm stops when there is no edge left in the set or if a minimum spanning tree has been created (in this specific case, the algorithm stops after 23 edges have been added).
- For the cycle detection, the union-find algorithm is used.

In this research, Kruskal's algorithm was applied with two different sets of edge weights to validate our choice of the Weight function. The resulting minimum spanning tree, in which the weight of each edge was calculated from the proposed weight function, was also used as the initial input for the flipping algorithm.

Flipping Algorithm

The Kruskal's algorithm is the easiest, quickest algorithm to create the minimum spanning tree with "static" edge weights; i.e., the weight of each edge stays the same throughout the iteration. However, in this project, the traffic flows within the tunnels changes based on the choice of edges; traffic has to be redirected from tunnels that were not chosen. Therefore, we needed another algorithm that can reflect the dynamically changing edge weights: the flipping algorithm. The flipping algorithm modifies a spanning tree by viewing combinations of edge insertions and deletions until the cost of the resulting spanning tree reaches a minimum. In every iteration of the flipping algorithm, the flow of each edge was recalculated and the sum of weights of all edges (i.e., the cost of the graph) was compared. The flow chart for the flipping algorithm is shown in Figure 4.

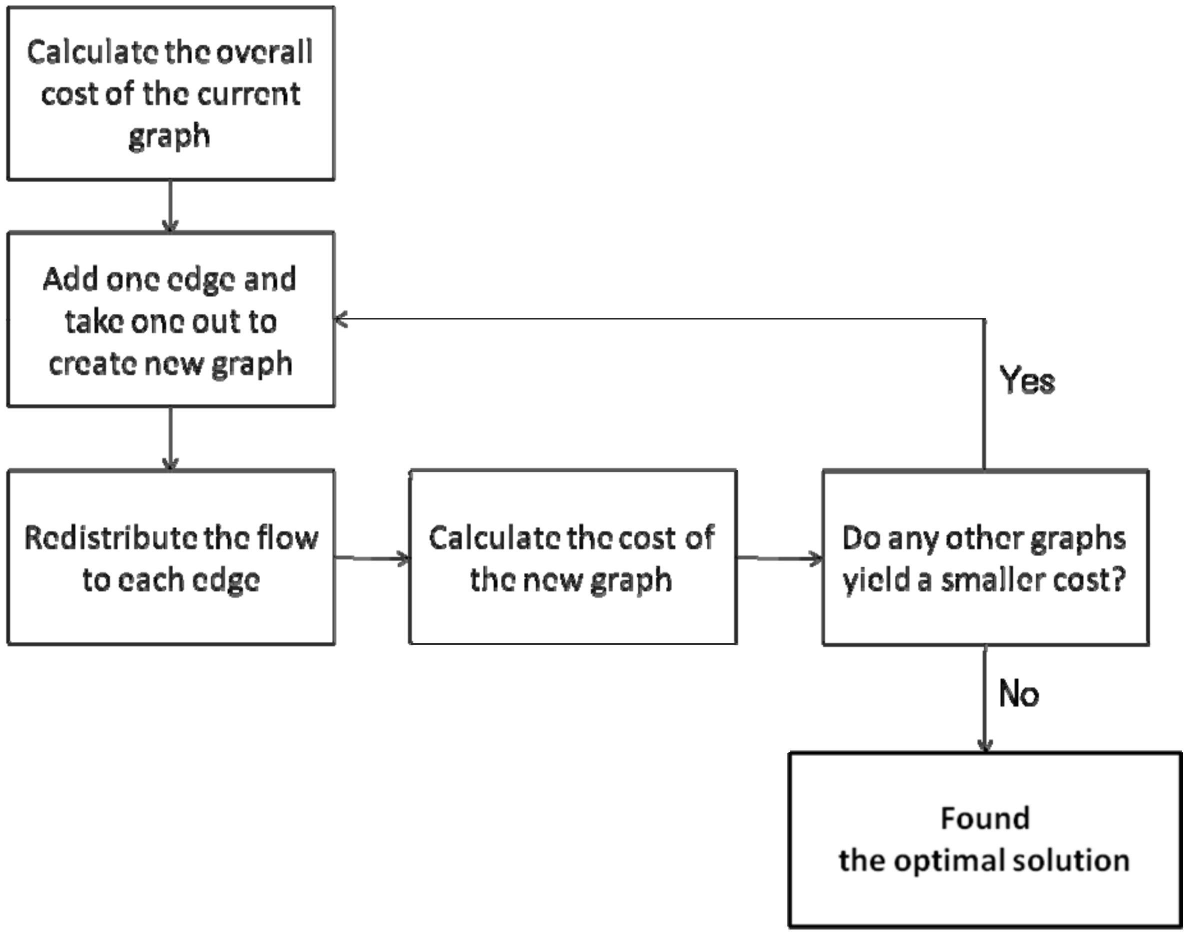

Calculate the overall cost of the current graph
Add one edge and take one out to create new graph
Redistribute the flow to each edge
Calculate the cost of the new graph
Do any other graphs yield a smaller cost?
Yes
No
Found the optimal solution

Figure 4. The schematic diagram of the flipping algorithm

As previously mentioned, the minimum spanning tree generated by Kruskal's algorithm using the graph with the Weight function as edge weights was a basis for the initial spanning tree for the flipping algorithm. Though the connectivity of the initial graph stays the same as the result from Kruskal's algorithm, the flow of each edge was re-calculated. Then, an edge is added to the spanning tree. The cycle detection method of the flipping algorithm finds the cycle created by the addition of this edge; the cycle detection algorithm was a slight modification of the depth first search algorithm. Then it removes an edge in the cycle and redistributes its flow to all the other edges in the cycle. In other words, the flow of each edge in the cycle increases by the flow of the removed edge. The cost of the resulting spanning tree is calculated. The cost of the newly generated graph is compared with the cost of the original graph, and the graph with the smaller cost is chosen. The algorithm is repeated until none of alternative graphs yields a smaller cost than the current graph. Then, the optimal solution is found and the algorithm stops.

Results and Discussion

The validity of the weight function was determined by how well it reflected the community members' preference in the results, while taking into consideration the cost of construction. To test this, Kruskal's algorithm was applied to the same graph with different weights. These results are shown in Figure C-1 and Figure C-2 in Appendix C.

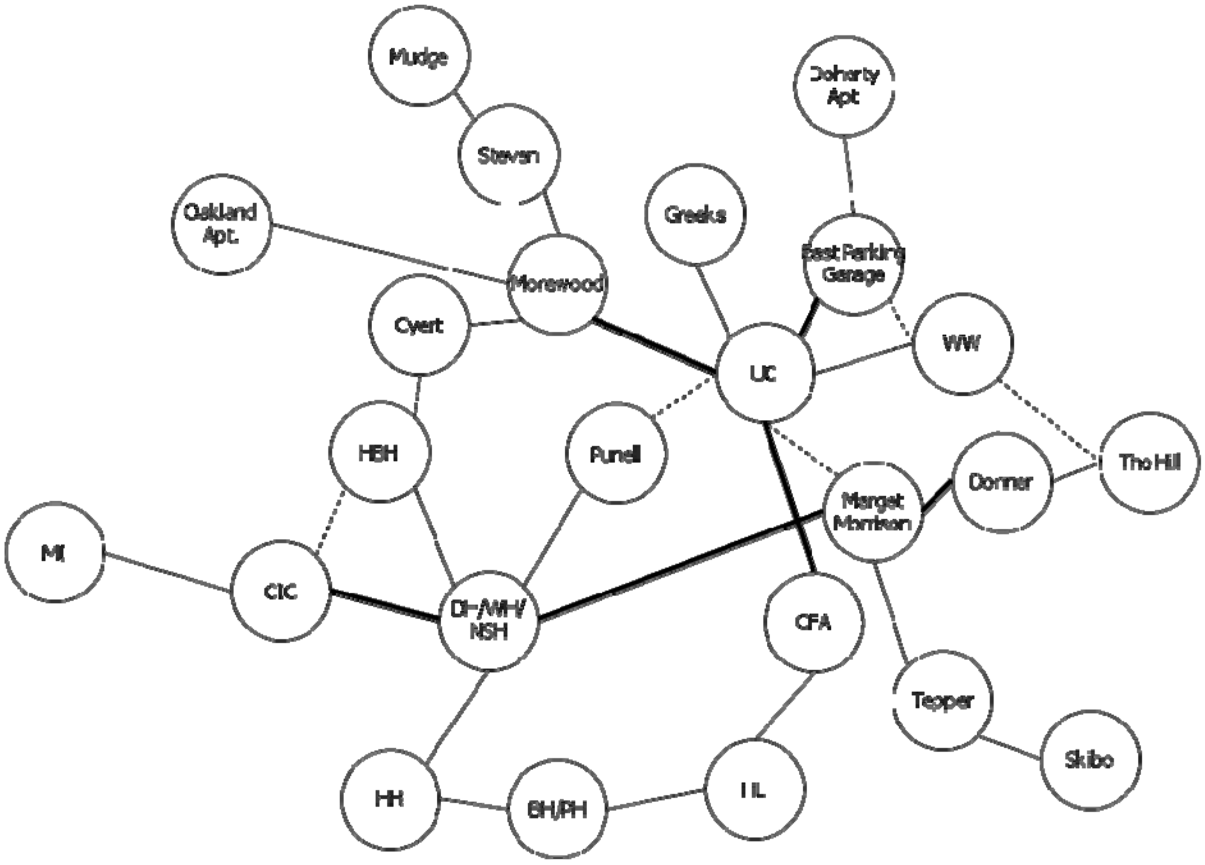


Figure 5. Comparison of the resulting MST from Kruskal's algorithm using different weights. *The bolded line represents the edge which was added when the Weight function was used while the dashed line represents the edge removed when the Weight function was used.*

As shown in Figure 5, the resulting MST, where each edge weight was calculated using the Weight function, contains the longer, but more demanded edges compared to the resulting MST in which only distance was considered. For example, Kruskal's algorithm chose Edge s (the edge between CIC and DH/WH/NSH) over Edge y (the edge between HBH and CIC). The edge y has the distance of 2 and the flow of 3 while the edge s has the distance of 3 and the flow of 6. Though Edge s is 1.5 times longer than Edge y, it has twice the demand of Edgy y. Indeed, Kruskal's algorithm chooses edges with higher demand and slightly longer length by imposing the Weight function. Therefore, it is concluded that the Weight function reflects the community members' preference and balances it out with the actual construction fee.

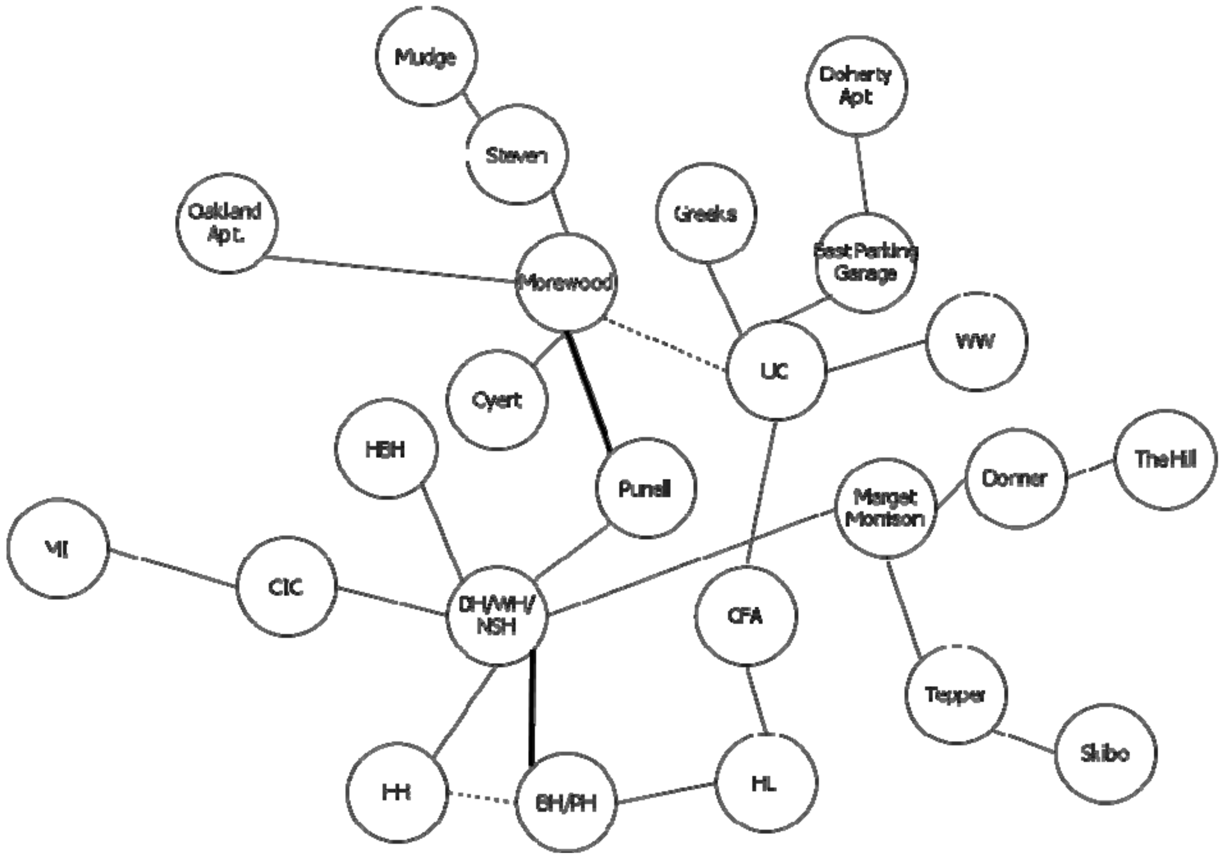


Figure 6. Comparison of the resulting MST from Kruskal's algorithm and the flipping algorithm. *The bolded line represents the edge which was added from the flipping algorithm while the dashed line represents the edge removed from the flipping algorithm.*

After redistributing the flow of traffic, the tree obtained from Kruskal's algorithm is no longer the optimal solution. Thus, using the flipping algorithm, the cost of the overall graph decreased by 10%. The cost of the graph from Kruskal's algorithm was 673.1 while the cost of the graph from the flipping algorithm was 611.9. Hence the flipping algorithm yields a better solution than the Kruskal's algorithm did. Also, in Figure 6, the result from the flipping algorithm is not so much different from the result of Kruskal's algorithm, indicating that Kruskal's algorithm is not a bad approximation to our problem. The resulting minimum spanning tree from the flipping algorithm is shown in Figure C-3 in Appendix C.

While the methodology employed was suitable for the problem, there are many modifications that can be made:

Striving for Social Optimality

As discussed earlier, developing the Weight function was challenging. Though the Weight function we created was successful in balancing out the construction fee and the demand from the community members, the resulting tree is not the most socially optimal. Note that Edge n (DH – UC) was not included in any of the resulting trees despite the fact that it had the highest demand (63%). Hence, it would be beneficial to analyze the effect of the Weight function on the minimum spanning trees produced. This would provide intuition on how to construct a Weight function that yields socially optimal spanning trees.

Steiner Tree

Instead of constructing minimum spanning trees, Steiner trees can be constructed. Steiner trees are modified minimum spanning trees that allow the addition of edges not contained in the graph to be included if it will further minimize the total cost[3]. With respect to the underground tunnel system, certain tunnel structures can be allowed such as having main tunnels with secondary branches to reach the buildings.

Real-World Solution

Another possible next step for this investigation is to use real costs associated with building tunnels. In this investigation, numerical values that sufficiently convey the relative costs of edges were used. With some research into construction costs, the investigation can be made less theoretical and more realistic. The results would be more directly relevant and presentable to interested groups, such as the Carnegie Mellon administration.

Reference

1. Carnegie Mellon Campus Plan, May 20, 2002.
2. Wikipedia – Kruskal's Algorithm, http://en.wikipedia.org/wiki/Kruskal%27s_algorithm
3. Wikipedia – Steiner Tree, http://en.wikipedia.org/wiki/Steiner_tree

Appendix

Appendix A – the survey result

Edge			Distance	Flow	Cost
	Starting	Ending			
a	Mudge	Steven	2	4	12.5
b	Steven	Morewood	2	6	5.56
c	Doherty Apt	East Parking	4	7	8.16
d	Greeks	UC	5	8	7.81
e	Morewood	UC	6	40	15
f	UC	East Parking	5	11	4.13
g	Morewood	Cyert	3	16	1.17
h	Morewood	Purnell	5	32	15.63
i	UC	WW/Resnik	2	8	3.13
j	The Hill	Donner	2	7	4.08
k	Donner	Marget Morrison	3	10	3
l	MI	CIC	15	29	1.78
m	Hamburg	NSH	2	7	4.08
n	Doherty Hall	UC	7	63	88.19
o	Purnell	Doherty Hall	3	35	8.57
p	Doherty Hall	Marget Morrison	7	32	21.88
q	Marget Morrison	Posner	2	3	22.22
r	Posner	Skibo Gym	3	4	18.75
s	CIC	Doherty Hall	3	6	8.33
t	Doherty Hall	Baker Hall	3	61	38.41
u	Hamerschlag	Doherty Hall	2	13	1.18
v	Hamerschlag	Baker Hall	2	10	2
w	Baker Hall	Hunt	2	15	0.89
x	Hunt	CFA	2	6	5.56
y	CIC	Hamburg	2	3	22.22
aa	UC	CFA	5	20	1.25
bb	Purnell	CFA	7	8	10.94
cc	Purnell	UC	3	2	75
dd	Hamburg	UC	11	1	1100

ee	CFA/Hunt	Doherty Hall	4	1	400
ff	Morewood	Hamburg	7	1	700
gg	UC	Hunt	8	1	800
hh	Off-campus	Morewood	14	2	350
ii	Marget Morrison	UC	2	2	50
jj	Morewood	Doherty Hall	10	1	1000
kk	Morewood	Baker Hall	14	1	1400
ll	Cyert	Hamburg	2	1	200
mm	The Hill	WW/Resnik	2	1	200
nn	WW/Resnik	East Parking	4	1	400

Appendix B – the graph before and after the survey

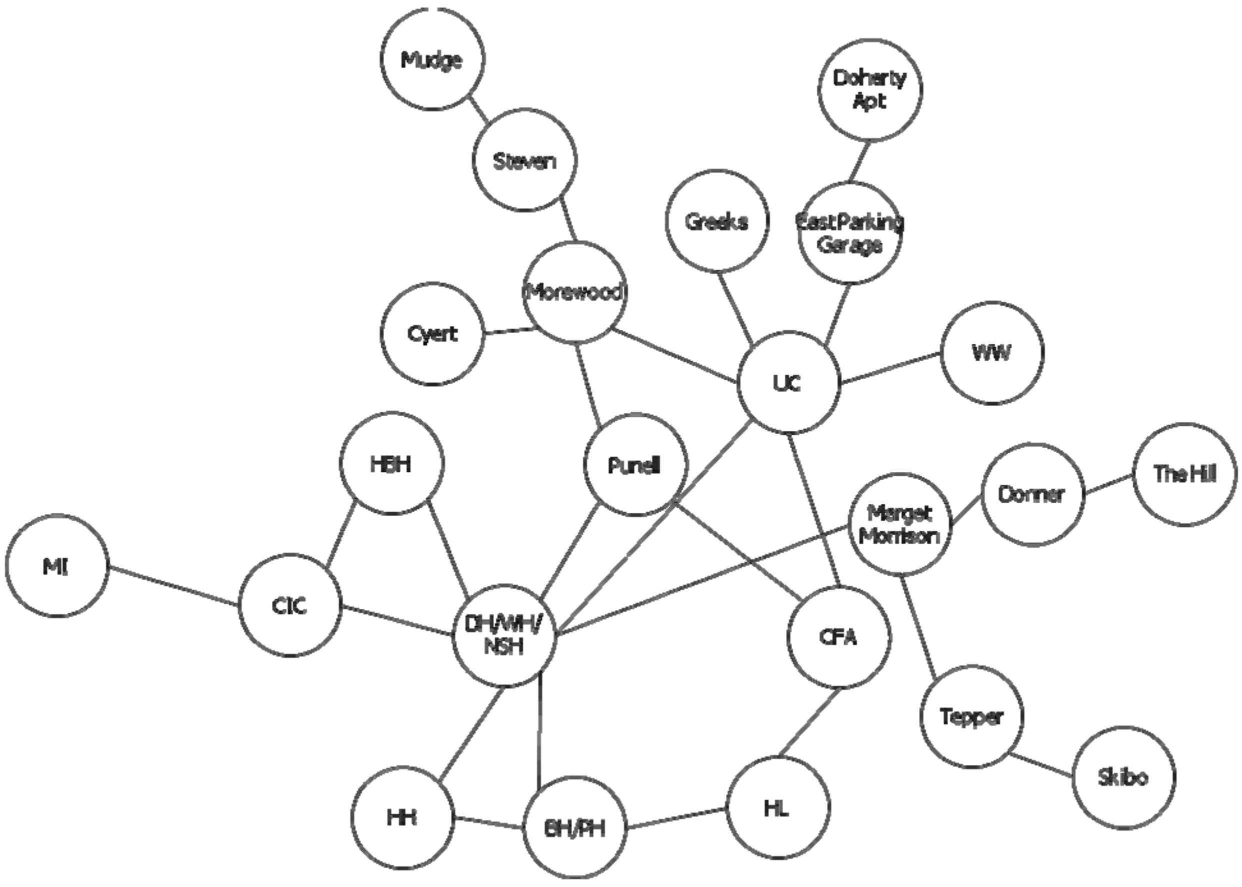


Figure B-1. The graph representing the original map used in the survey

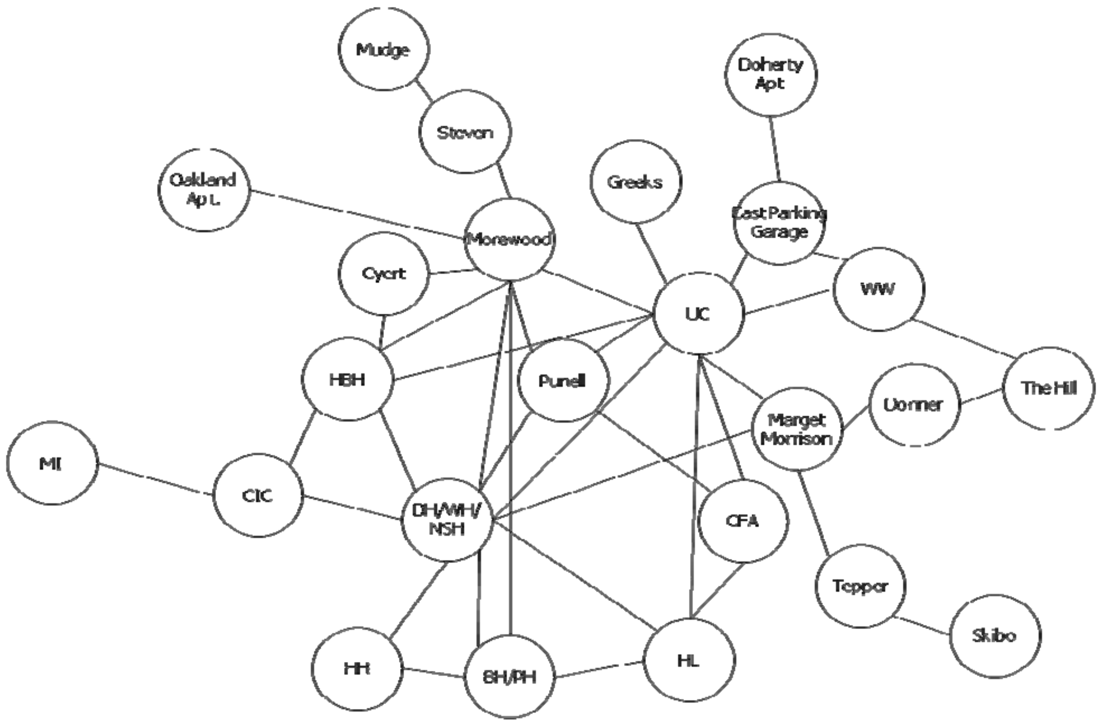


Figure B-2. The modified graph after incorporate the suggestions from the survey takers

Appendix C – the resulting MST from each algorithm

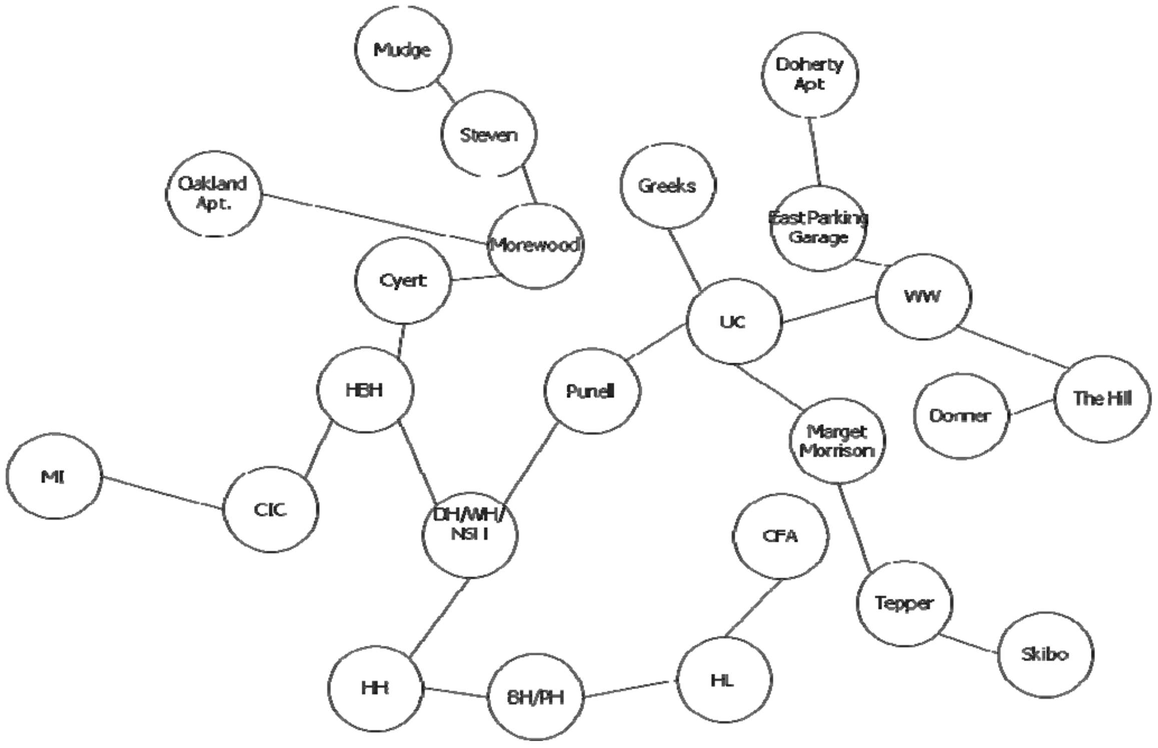


Figure C-1. The solution from Kruskal's Algorithm based solely on distances

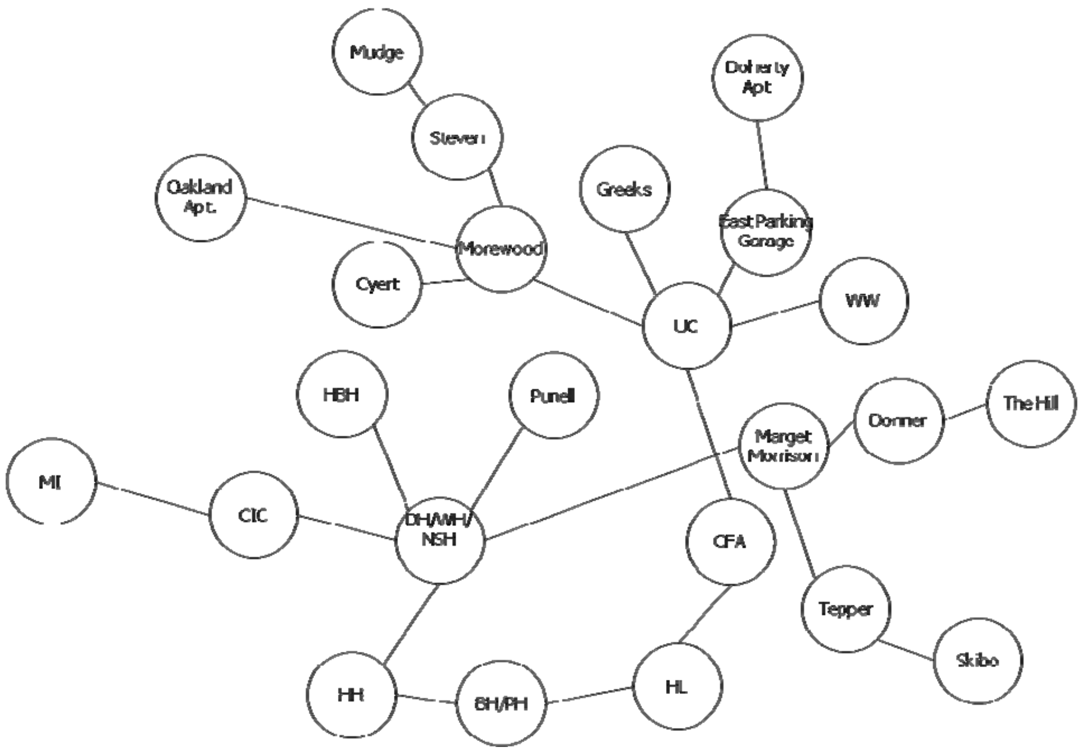


Figure C-2. The solution from Kruskal's Algorithm based on the Cost function

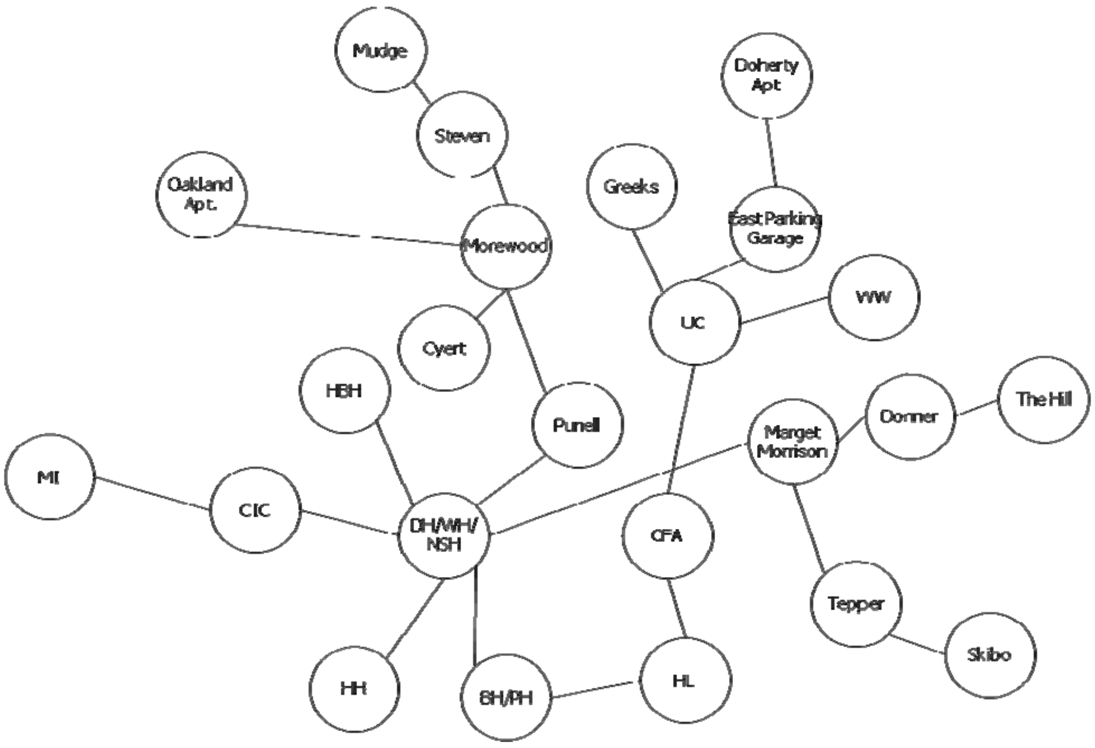


Figure C-3. The optimal solution from the Flipping Algorithm

Appendix D – the code for Kruskal's algorithm

```
import java.util.*;

public class Tunnels {
        public static void main(String[] args) {
                public static final ArrayList<Edge> allEdges = new ArrayList<Edge>();

                Edge a = new Edge(, 19, 20);
                Edge b = new Edge(, 18, 19);
                Edge c = new Edge(, 22, 23);
                Edge d = new Edge(, 15, 21);
                Edge e = new Edge(, 15, 18);
                Edge f = new Edge(, 15, 23);
                Edge g = new Edge(, 17, 18);
                Edge h = new Edge(, 16, 18);
                Edge i = new Edge(, 14, 15);
                Edge j = new Edge(, 12, 13);
                Edge k = new Edge(, 11, 12);
                Edge l = new Edge(, 1, 2);
                Edge m = new Edge(, 3, 4);
                Edge n = new Edge(, 3, 15);
                Edge o = new Edge(, 3, 16);
                Edge p = new Edge(, 3, 11);
                Edge q = new Edge(, 9, 11);
                Edge r = new Edge(, 9, 10);
                Edge s = new Edge(, 2, 3);
                Edge t = new Edge(, 3, 6);
                Edge u = new Edge(, 5, 3);
                Edge v = new Edge(, 5, 6);
                Edge w = new Edge(, 7, 6);
                Edge x = new Edge(, 7, 8);
                Edge y = new Edge(, 2, 4);
                Edge aa = new Edge(, 8, 15);
                Edge bb = new Edge(, 8, 16);
                Edge cc = new Edge(, 15, 16);
                Edge dd = new Edge(, 4, 15);
                Edge ee = new Edge(, 3, 7);
                Edge ff = new Edge(, 4, 18);
                Edge gg = new Edge(, 7, 15);
                Edge hh = new Edge(, 24, 18);
                Edge ii = new Edge(, 11, 15);
                Edge jj = new Edge(, 18, 3);
                Edge kk = new Edge(, 6, 18);
                Edge ll = new Edge(, 4, 17);
                Edge mm = new Edge(, 13, 14);
                Edge nn = new Edge(, 14, 23);

                allEdges.add(a);
                allEdges.add(b);
                allEdges.add(c);
                allEdges.add(d);
                allEdges.add(e);
                allEdges.add(f);
                allEdges.add(g);
                allEdges.add(h);
                allEdges.add(i);
                allEdges.add(j);
                allEdges.add(k);
                allEdges.add(l);
                allEdges.add(m);
                allEdges.add(n);
                allEdges.add(o);
                allEdges.add(p);
                allEdges.add(q);
                allEdges.add(r);
                allEdges.add(s);
                allEdges.add(t);
                allEdges.add(u);
                allEdges.add(v);
                allEdges.add(w);
                allEdges.add(x);
                allEdges.add(y);
                allEdges.add(aa);
                allEdges.add(bb);
                allEdges.add(cc);
                allEdges.add(dd);
                allEdges.add(ee);
                allEdges.add(ff);
                allEdges.add(gg);
                allEdges.add(hh);
                allEdges.add(ii);
                allEdges.add(jj);
                allEdges.add(kk);
                allEdges.add(ll);
                allEdges.add(mm);
                allEdges.add(nn);

                public static final int numEdges = allEdges.size();
          }

          static HashMap map;

          static ArrayList<Edge> kruskal(ArrayList<Edge> edgelist, int vertexCount) {
                ArrayList<Edge> ordered = quickSort(edgelist); // order the edges
                int size = ordered.size(); // no. of edges
                map = new HashMap();
                ArrayList<Edge> min = new ArrayList<Edge>();
                int count = vertexCount; // no. of vertices
                for (int i = 0; i < size; i++) {
                        if (count < 2) break;           // (no. of vertices-1) has been inserted
                        Edge e = ordered.get(i); // get min edge
                        int[] vertices = e.findVertices();
                        if (find(vertices[0]) != find(vertices[1])) { // not a cycle
                                uni(vertices[0], vertices[1]);
                                min.add(e); // insert edge in MST
                                count--;
                        }
                }
                if (count > 1) //no spanning tree
                        return null;
                return min;
          }

          static ArrayList<Edge> quickSort(ArrayList<Edge> edgelist) {
                int length = edgelist.size();
                ArrayList<Edge> fullList = new ArrayList<Edge>();
                if (length > 2) {
                        int pivot = length/2;
                        int pivotW = edgelist.get(pivot).findWeight();
                        ArrayList<Edge> lessList = new ArrayList<Edge>();
                        ArrayList<Edge> moreList = new ArrayList<Edge>();
                        for (int i = 0; i < length; i++) {
                                if (i == pivot)         continue;
                                Edge iEdge = edgelist.get(i);
                                if (iEdge.findWeight() < pivotW)     lessList.add(iEdge);
                                if (iEdge.findWeight() >= pivotW)   moreList.add(iEdge);
                        }
                        lessList = quickSort(lessList); //recurse on edges < pivot
                        moreList = quickSort(moreList); //recurse on edge >= pivot
                        fullList.addAll(lessList);
                        fullList.add(edgelist.get(pivot));
                        fullList.addAll(moreList);
                        return fullList;
                }
                if (length == 2) {
                        if (edgelist.get(0).findWeight() <= edgelist.get(1).findWeight())
                                return edgelist;
                        ArrayList<Edge> newList = new ArrayList<Edge>();
                        newList.add(edgelist.get(1));
                        newList.add(edgelist.get(0));
                        return newList;
                }
                return edgelist;
          }

          static int find(int vertex) {
                Node nodeV = (Node)(map.get(vertex));
                if (nodeV == null) {
                        map.put(vertex, new Node());
                        return vertex;
                }
                if (nodeV.getPred() == -1 || nodeV.getPred() == vertex)
                        return vertex;
                int root = find(nodeV.getPred());
                nodeV.changePred(root); //path compression
```

```
                return root;
        }

        static void uni(int vertexA, int vertexB) {
                int aParent = find(vertexA);
                int bParent = find(vertexB);
                Node aParentN = (Node)(map.get(aParent));
                Node bParentN = (Node)(map.get(bParent));
                int aParentSize = aParentN.getSize();
                int bParentSize = bParentN.getSize();
                if (aParentSize < bParentSize) //set a is smaller than set b
                        aParentN.changePred(bParent);
                else if (bParentSize < aParentSize) //set b is smaller than set a
                        bParentN.changePred(aParent);
                else {
                        aParentN.changePred(bParent);
                        bParentSize++;
                        bParentN.changeSize(bParentSize);
                }
        }

        static void clear() {
                map = new HashMap();
        }
        static BoolMap toBoolean(ArrayList<Edge> edgelist){
                // initialize
                BoolMap mapAsBool = new BoolMap();

                for(int i = 0; i < numEdges; i++){
                        if(edgelist.contains(allEdges(i)){
                                mapAsBool.update(i);
                        }
                }
        }

private static class Edge {
            double length;
            double flow;
            int vertex1;
            int vertex2;
            double cost = 0;

            Edge(int length, int flow, int vertex1, int vertex2) {
                  this.length = (double)length;
                  this.flow = (double)flow;
                  this.vertex1 = vertex1;
                  this.vertex2 = vertex2;
            }

            double findCost() {
                  if (this.flow < 30)
                        this.cost = 100*this.length/((this.flow)*(this.flow));
                  else if (this.flow < 60)
```

```
				this.cost = 100*this.length/this.flow;
			else this.cost = 100*this.length/Math.sqrt(this.flow);
			return this.cost;
		}

		void changeFlow(double newFlow) {	this.flow = newFlow;		}

		double findFlow() {		return this.flow;		}

		int[] findVertices() {
			int[] vertices = new int[2];
			vertices[0] = this.vertex1;
			vertices[1] = this.vertex2;
			return vertices;
		}

		public String toString() {
			return ("["+cost+" "+vertex1+" "+vertex2+"]");
		}
	}

	private static class Node {
		private int pred = -1;
		private int size = 1;

		public Node() {}

		public int getPred() {	return this.pred;		}

		public void changePred(int pred) {	this.pred = pred;	}

		public int getSize() {	return this.size;		}

		public void changeSize(int size) {		this.size = size;		}
	}

	private static class BoolMap{
		int[] map = new int[numEdges];

		public BoolMap(){
			for(int i = 0; i < numEdges; i++){
				map[i] = 0;
			}
		}

		public void update(int index){	map[index] = 1;		}

		public int getValue(int index){	return map[index];	}

		public boolean equals(Object map2){
			BoolMap boolMap2 = (BoolMap) map2;

			for (int i = 0; i < numEdges; i++){
```

```
                        if(this.getValue(i) != boolMap2.getValue(i))
                              return false;
                  }
                  return true;
            }
      }
}
```

Appendix D – the code for the flipping algorithm

```
import java.util.*;

public class FlippingAlgorithm1 {

        private static int numEdges = 39;
        private static int numNodes = 24;
        private static ArrayList<Edge> allEdgesInMap = new ArrayList<Edge>();
        private static HashSet<Tree> alreadySeenTree = new HashSet<Tree>();
        private static double[] flowList = new double[numEdges+1];

        public static void main(String[] args) {
                Edge a = new Edge(1, 2, 4, 19, 20);
                Edge b = new Edge(2, 2, 6, 18, 19);
                Edge c = new Edge(3, 4, 7, 22, 23);
                Edge d = new Edge(4, 5, 8, 15, 21);
                Edge e = new Edge(5, 6, 40, 15, 18);
                Edge f = new Edge(6, 5, 11, 15, 23);
                Edge g = new Edge(7, 3, 16, 17, 18);
                Edge h = new Edge(8, 5, 32, 16, 18);
                Edge i = new Edge(9, 2, 8, 14, 15);
                Edge j = new Edge(10, 2, 7, 12, 13);
                Edge k = new Edge(11, 3, 10, 11, 12);
                Edge l = new Edge(12, 15, 29, 1, 2);
                Edge m = new Edge(13, 2, 7, 3, 4);
                Edge n = new Edge(14, 7, 63, 3, 15);
                Edge o = new Edge(15, 3, 35, 3, 16);
                Edge p = new Edge(16, 7, 32, 3, 11);
                Edge q = new Edge(17, 2, 3, 9, 11);
                Edge r = new Edge(18, 3, 4, 9, 10);
                Edge s = new Edge(19, 3, 6, 2, 3);
                Edge t = new Edge(20, 3, 61, 3, 6);
                Edge u = new Edge(21, 2, 13, 5, 3);
                Edge v = new Edge(22, 2, 10, 5, 6);
                Edge w = new Edge(23, 2, 15, 7, 6);
                Edge x = new Edge(24, 2, 6, 7, 8);
                Edge y = new Edge(25, 2, 3, 2, 4);
                Edge aa = new Edge(26, 5, 20, 8, 15);
                Edge bb = new Edge(27, 7, 8, 8, 16);
                Edge cc = new Edge(28, 3, 2, 15, 16);
                Edge dd = new Edge(29, 11, 1, 4, 15);
                Edge ee = new Edge(30, 4, 1, 3, 7);
                Edge ff = new Edge(31, 7, 1, 4, 18);
                Edge gg = new Edge(32, 8, 1, 7, 15);
                Edge hh = new Edge(33, 14, 2, 24, 18);
                Edge ii = new Edge(34, 2, 2, 11, 15);
                Edge jj = new Edge(35, 10, 1, 18, 3);
                Edge kk = new Edge(36, 14, 1, 6, 18);
                Edge ll = new Edge(37, 2, 1, 4, 17);
                Edge mm = new Edge(38, 2, 1, 13, 14);
                Edge nn = new Edge(39, 4, 1, 14, 23);

                allEdgesInMap.add(a);
                allEdgesInMap.add(a);
                allEdgesInMap.add(b);
                allEdgesInMap.add(c);
                allEdgesInMap.add(d);
                allEdgesInMap.add(e);
                allEdgesInMap.add(f);
                allEdgesInMap.add(g);
                allEdgesInMap.add(h);
                allEdgesInMap.add(i);
                allEdgesInMap.add(j);
                allEdgesInMap.add(k);
                allEdgesInMap.add(l);
                allEdgesInMap.add(m);
                allEdgesInMap.add(n);
                allEdgesInMap.add(o);
                allEdgesInMap.add(p);
                allEdgesInMap.add(q);
                allEdgesInMap.add(r);
                allEdgesInMap.add(s);
                allEdgesInMap.add(t);
                allEdgesInMap.add(u);
                allEdgesInMap.add(v);
                allEdgesInMap.add(w);
                allEdgesInMap.add(x);
                allEdgesInMap.add(y);
                allEdgesInMap.add(aa);
                allEdgesInMap.add(bb);
                allEdgesInMap.add(cc);
                allEdgesInMap.add(dd);
                allEdgesInMap.add(ee);
                allEdgesInMap.add(ff);
                allEdgesInMap.add(gg);
                allEdgesInMap.add(hh);
                allEdgesInMap.add(ii);
                allEdgesInMap.add(jj);
                allEdgesInMap.add(kk);
                allEdgesInMap.add(ll);
                allEdgesInMap.add(mm);
                allEdgesInMap.add(nn);
                for (int index = 1; index < 40; index++)
                      flowList[index] = allEdgesInMap.get(index).getFlow();
```

```
ArrayList kruskal = new ArrayList();
kruskal.add(l);
kruskal.add(s);
kruskal.add(m);
kruskal.add(u);
kruskal.add(v);
kruskal.add(w);
kruskal.add(x);
kruskal.add(q);
kruskal.add(r);
kruskal.add(k);
kruskal.add(j);
kruskal.add(p);
kruskal.add(o);
kruskal.add(aa);
kruskal.add(i);
kruskal.add(hh);
kruskal.add(a);
kruskal.add(b);
kruskal.add(g);
kruskal.add(e);
kruskal.add(d);
kruskal.add(f);
kruskal.add(c);


Tree kruskTr = new Tree(kruskal);
System.out.println(kruskTr.toString());
System.out.println(kruskTr.getOverallCost());
double bestCost = redist(kruskTr);
System.out.println(kruskTr.toString());
System.out.println(bestCost);
Tree best = new Tree(kruskTr);
boolean[] all = new boolean[40];
all = kruskTr.getEdgeList();
for (int index = 1; index < all.length; index++) {
        if (!all[index]) {
                Edge newEdge = allEdgesInMap.get(index);
                ArrayList<Edge> cycleList = kruskTr.traceCycle(newEdge);
                kruskTr.addEdge(newEdge);
                for (int index2 = 0; index2 < cycleList.size(); index2++) {
                        kruskTr.deleteEdge(cycleList.get(index2));
                        double currentCost = redist(kruskTr);
                        if (currentCost < bestCost) {
                                best = new Tree(kruskTr);
                                bestCost = currentCost;
                        }
                        kruskTr.addEdge(cycleList.get(index2));
                }
                kruskTr.deleteEdge(newEdge);
                if (!best.equals(kruskTr)) {
                        kruskTr = new Tree(best);
                        all = kruskTr.getEdgeList();
                    }
                }
            }
            System.out.println(best.toString());
            System.out.println(redist(best));
    }

    public static double redist(Tree tree) {
            boolean[] all = tree.getEdgeList();
            for (int index = 1; index < all.length; index++) {
                    if (all[index])
                            allEdgesInMap.get(index).changeFlow(flowList[index]);
            }
            for (int index = 1; index < all.length; index++) {
                    if (!all[index]) {
                            ArrayList<Edge> cycleList =
                            tree.traceCycle(allEdgesInMap.get(index));
                            for (int index2 = 0; index2 < cycleList.size(); index2++) {
                                    Edge e = cycleList.get(index2);
                                    double val = e.getFlow();
                                    e.changeFlow(flowList[index] + val);
                            }
                    }
            }
            return tree.getOverallCost();
    }

    private static class Tree {
            private double totalCost = 0.0;
            private boolean[] edgeList = new boolean[numEdges+1];
            private double[] updatedEdgeWeight = new double[numEdges+1];
            private ArrayList<Edge> allEdges = new ArrayList<Edge>();
            private ArrayList<Edge>[] nodeList = new ArrayList[numNodes+1];

            public Tree(ArrayList<Edge> listEdge) {
                    for(int i=0; i<this.nodeList.length; i++)
                            this.nodeList[i] = new ArrayList();
                    for(int i=0; i<listEdge.size(); i++)
                            addEdge(listEdge.get(i));

            }

            public Tree(Tree tree) {
                    for(int i=0; i<this.nodeList.length; i++)
                            this.nodeList[i] = new ArrayList();
                    for(int i=0; i<tree.getAllEdges().size(); i++)
                            addEdge(tree.getAllEdges().get(i));
            }

            public boolean[] getEdgeList() {     return this.edgeList;  }
            public ArrayList<Edge> getAllEdges() {     return this.allEdges;  }
            public double[] getEdgeWeight() { return this.updatedEdgeWeight;}
```

```
public boolean addEdge(Edge e) {
        if(contains(e))
                return false;
        this.edgeList[e.getIndex()] = true;
        this.allEdges.add(e);
        this.nodeList[e.getVertex1()].add(e);
        this.nodeList[e.getVertex2()].add(e);
        this.updatedEdgeWeight[e.getIndex()] = e.getCost();
        return true;
}

public boolean deleteEdge(Edge e) {
        if(!contains(e))
                return false;

        this.edgeList[e.getIndex()] = false;
        this.updatedEdgeWeight[e.getIndex()] =
        allEdgesInMap.get(e.getIndex()).getCost();
        this.nodeList[e.getVertex1()].remove(e);
        this.nodeList[e.getVertex2()].remove(e);
        this.allEdges.remove(e);
        return true;
}

public double getOverallCost() {
        this.totalCost = 0.0;
        for(int i=0; i<this.allEdges.size(); i++)
                this.totalCost += this.allEdges.get(i).getCost();
        return this.totalCost;
}

public boolean equals(Object o) {
        Tree t = (Tree)o;
        boolean[] tBoolean = t.getEdgeList();
        for(int i=0; i<edgeList.length; i++) {
                Boolean thisEdgeBoolean = new Boolean(edgeList[i]);
                Boolean thatEdgeBoolean = new Boolean(tBoolean[i]);
                if(!thisEdgeBoolean.equals(thatEdgeBoolean))
                        return false;
        }
        return true;
}

public boolean contains(Object o) {
        Edge e = (Edge)o;
        return edgeList[e.getIndex()];
}

public String toString() {
        String str = "";
        for (int i=0; i<allEdges.size(); i++)
                str += allEdges.get(i).toString() + " ";
```

```
			return str;
		}

		public ArrayList<Edge> traceCycle(Object o) {
			Edge e = (Edge) o;
			int vertex1 = e.getVertex1();
			int vertex2 = e.getVertex2();
			ArrayList<Edge> outgoing = nodeList[vertex1];
			ArrayList<Edge> returned = new ArrayList<Edge>();
			for (int i = 0; i < outgoing.size(); i++) {
				Edge k = outgoing.get(i);
				ArrayList<Edge> current = new ArrayList<Edge>();
				current.add(k);
				if (k.getVertex1() == vertex1)
					returned = tracer(k.getVertex2(), vertex2, k, current);
				else
					returned = tracer(k.getVertex1(), vertex2, k, current);
				if (returned.size() != 0)
					return returned;
			}
			return returned;

		}

		public ArrayList<Edge> tracer(int vertexNow, int vertexEnd, Object o,
		ArrayList<Edge> current) {
			ArrayList<Edge> outgoing = nodeList[vertexNow];
			outgoing.remove((Edge)o);
			if (outgoing.size() == 0) {
				current.remove((Edge)o);
				return current;
			}
			for (int i = 0; i < outgoing.size(); i++) {
				Edge k = outgoing.get(i);
				current.add(k);
				if (k.getVertex1() == vertexNow && k.getVertex2() ==
vertexEnd)
					return current;
				else if (k.getVertex1()==vertexEnd &&
k.getVertex2()==vertexNow)
					return current;
				else {
					if (k.getVertex1()==vertexNow) {
						ArrayList<Edge> returned =
tracer(k.getVertex2(), vertexEnd, k, current);
						if (returned.size() != current.size())
							return returned;
					}
					else {
						ArrayList<Edge> returned =
tracer(k.getVertex1(), vertexEnd, k, current);
						if (returned.size() != current.size())
							return returned;
```

```
                        }
                    }
                }
                current.remove((Edge)o);
                return current;
            }
        }


        private static class Edge {
            private int index, vertex1, vertex2;
            private double flow, length, cost;

            public Edge(int index, double length, double flow, int vertex1, int vertex2) {
                this.index = index;
                this.vertex1 = vertex1;             //since the index starts 0 in array
                this.vertex2 = vertex2;             //since the index starts 0 in array
                this.flow = flow;
                this.length = length;
                this.findCost();
            }

            public int getIndex() {        return index;  }

            public int getVertex1() {      return vertex1;        }

            public int getVertex2() {      return vertex2;        }

            public double getFlow() {      return flow;   }

            public double getLength() { return length; }

            public double getCost()        {       return cost;   }

            void findCost()
            {
                if (this.flow < 30)
                    this.cost = 100*this.length/((this.flow)*(this.flow));
                else if (this.flow < 60)
                    this.cost = 100*this.length/this.flow;
                else this.cost = 100*this.length/Math.sqrt(this.flow);
            }

            public void changeFlow(double newFlow)
            {
                this.flow = newFlow;
                this.findCost();               // recalculate the cost of the edge
            }

            public boolean equalsTo(Object o)
            {
                Edge e = (Edge)o;
                if(vertex1 == e.getVertex1() && vertex2 == e.getVertex2())
```

```
                        return true;
                else if(vertex1 == e.getVertex2() && vertex2 == e.getVertex1())
                        return true;

                return false;
        }
        public String toString() { return ("["+cost+" "+vertex1+" "+vertex2+"]"); }
    }
}
```