Modeling a Mailman's Problem
as
Finding an Eulerian Closed Circuit

Guo Michael Liu
Maurice Fabien
Wen Wang
Math 381
February 21, 2010

Abstract

Delivery and routing problems have been of great importance in our society and to the economy. Consider a mailman, who has a predetermined delivering route; but is this route the most efficient one? We will answer this question by modeling our mailman's problem as finding an Eulerian Closed Circuit and its associated optimal traveling time of the route, and then compare our optimal time to the actual time our mailman spends in delivering all his mails [3]. The algorithm we use to find the Eulerian Circuit is the powerful algorithm that solves the Directed Chinese Postman Problem by Dr. Harold Thimbleby, the founder of the Future Interaction Technology Lab at Swansea University in Wales and former director of the University College London's Interaction Center [5].

Problem Description

A mailman has a delivery route predetermined by the post office. However, it might not be the most efficient route. A friend of ours is a mailman working for the Unites States Postal Services in the rural area of the city of Yakima for over six years. We are trying to help him find a route that costs him the least amount of time to deliver all the mails. Starting from the post office, our mailman has to travel to all the mailboxes in his delivering route and end up back at the post office. This is a graph theoretic problem: all roads can be modeled as directed edges; and road junctions are vertices. Hence, the graph that shows our mailman's delivering routes is a multigraph, a graph that can have more than one edge between a pair of vertices [3]. Since all the roads our mailman travels have two-way traffic, so all the vertices connecting the roads have even degree. The degree of a vertex is defined as the number of edges through that vertex [3].

Therefore, our delivering route must have an Eulerian Closed Circuit by Euler's theorem. An Eulerian Closed Circuit is defined as a path that starts at a given vertex, traverses each edge of the graph exactly once, and returns to the starting point [3]. Our mailman probably wants the shortest tour with fewest repeated street visits, so our goal is to find the Eulerian Closed Circuit of his route and compute the optimal traveling time, which includes the time to deliver all the mails.

Mathematical Model

After talking to our friend, we were able to acquire his actual delivery route and the average delivery time he spends on each side of the roads. We then made a simplified graph based on the information our friend provided. Each vertex is numbered so that the notation is clearer. The number on each edge represents the delivery time on that side of the road. Notice that Vertex 0 is the post office, the origin and final destination for our mailman; other vertices are intersections of roads; the edges between vertex 0 and vertex 1 represent the virtual connection between the post office and the first intersection of the delivery route.

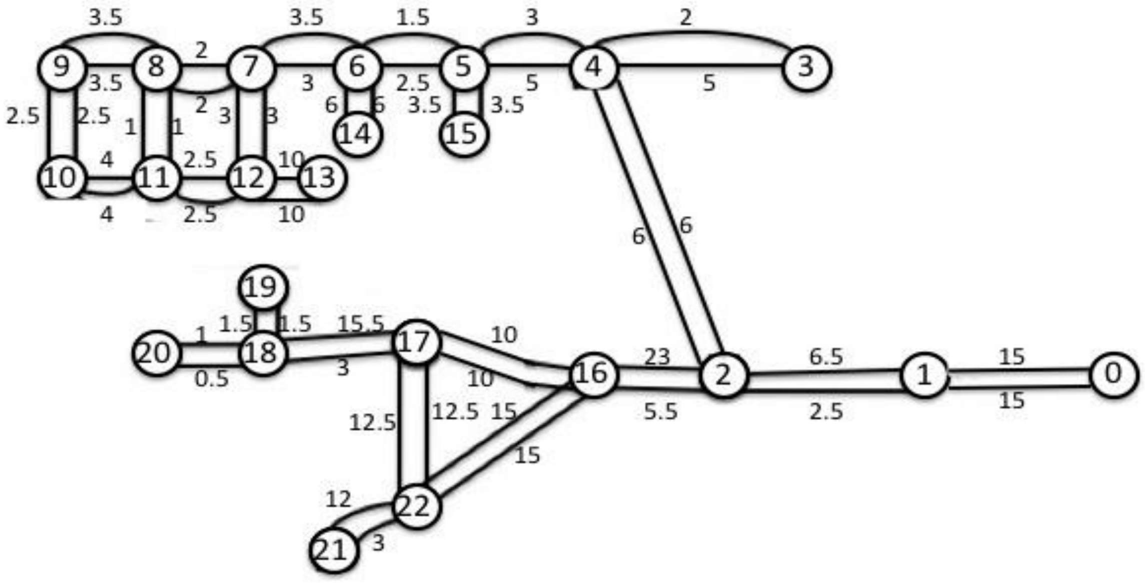

Figure 1: Simplified Delivering Route

The nodes represent the intersections and the edges represent the actual roads. Imagine the mailboxes standing on both sides of the roads between intersections are now standing on the corresponding edges between the nodes, and our friend has to travel to each edge once and the time he spends on each side of the roads is showed on the corresponding edges. Since the number of mailbox on each side of a road may be different, which means the time he spends on each side of the same road is different; so the amount of time showed on each edge between the same two nodes may be different.

Solution to the Mathematical Problem

To find the Eulerian Closed Circuit of our delivering route, we use an algorithm by Dr. Harold Thimbleby, which solves the Directed Chinese Postman Problem (CPP) [5]. A CPP does not necessarily have an Eulerian Closed Circuit; it may contain vertices that have odd degree. However, the goal of the CPP algorithm is the same as ours: to find the optimal tour for a postman. Since an Eulerian circuit, if exists, is the optimal solution to any CPP, and we already know our U.S. postman problem has an Eulerian circuit; so our problem is just a special case of the CPP. Therefore, we can use the CPP algorithm to solve our problem.

The mathematical presentation and the Java executable codes of the CPP algorithm are presented in Dr. Thimbleby's paper [5]. A full body of the Java codes is attached at the end of this paper. The idea of the algorithm is to minimize the times of adding edges between odd degree vertices so that the tour will become an Eulerian circuit, which is unnecessary for our case since our graph is already an Eulerian circuit. For the java codes, Dr. Thimbleby first constructs the CPP graph using his "addArc(String, int, int, double)" method from the CPP class. The first parameter of the "addArc" method is the name of the edge, we name all our edges "e"; the second parameter is the starting vertex number of the edge; the third parameter is the ending vertex number of the edge; the last parameter is the time cost of travelling through the edge. We use the same techniques to create our graph in Java, which is represented in figure 1; here is a piece of the codes we used to create our graph:

```
G.addArc("e", 0, 1, 15).addArc("e", 1, 0, 15).addArc("e", 1, 2, 6.5)……;
```

Then Dr. Thimbleby incorporates the Floyd-Warshall algorithm to find and record the shortest path in his "leastCostPaths()" method. The Floyd-Warshall algorithm is a graph analysis algorithm for finding the shortest paths in a weighted, directed graph; a single execution can find the shortest paths between all pairs of vertices [1]. He also uses the Cycle Canceling and Greedy algorithm to find the optimal solution in his "findFeasible()" and "improvements()" methods. The cycle-canceling algorithm is one of the earliest algorithms to solve the minimum-cost flow problem. This algorithm maintains a feasible solution x in the network G and proceeds by augmenting flows along negative-cost directed cycles in the residual network G(x) and thereby canceling them [4]. A greedy algorithm is any algorithm that follows the problem solving metaheuristic of making the locally optimal choice at each stage with the hope of finding the global optimum [2]. For our problem, we basically run the same codes Dr. Thimbleby used in his paper to find the Eulerian circuit and the time cost.

Results

After running the CPP algorithm, we obtained the Eulerian circuit for our route in the order of vertex numbers: 0-1-2-4-3-4-5-6-7-8-9-10-9-8-11-10-11-12-11-8-7-12-13-12-7-6-14-6-5-15-5-4-2-16-17-18-19-18-20-18-17-22-17-16-22-21-22-16-2-1-0, a visual representation is shown below in Figure 2. The optimal time to finish the route is 296 minutes (about 5 hours), which is the delivery time only; it does not include the time our friend stuck in traffic or his break time. Here is a piece of the actual output from the CPP algorithm:

```
Take arc e from 0 to 1
Take arc e from 1 to 2
          ...
Take arc e from 2 to 1
Take arc e from 1 to 0
Cost = 296.0
```

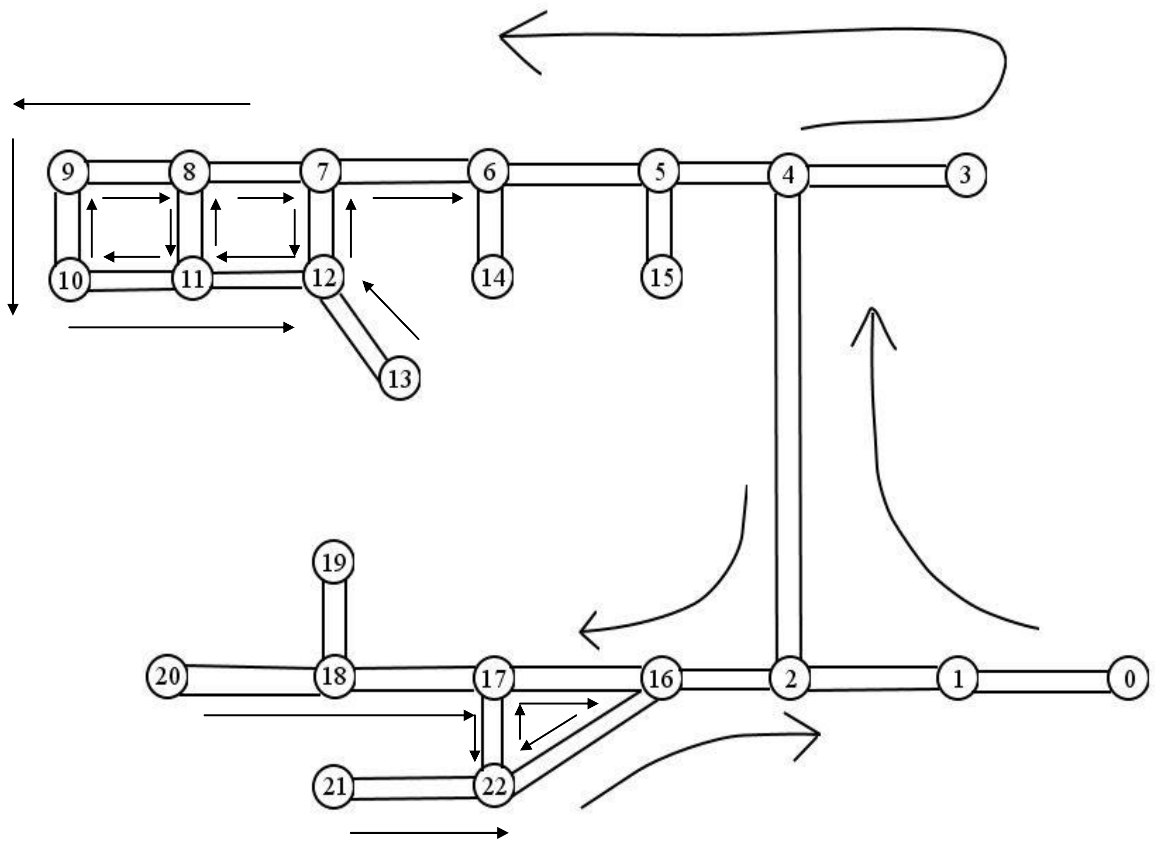

Figure 2: Visual Presentation of the Optimal Route found by CPP algorithm

For comparison, here is our mailman's actual route in the order of vertex numbers: 0-1-2-16-22-21-22-17-18-19-18-20-18-17-22-16-17-16-2-4-3-4-5-15-5-6-14-6-7-8-9-10-11-8-11-10-9-8-7-12-11-12-13-12-7-6-5-4-2-1-0, it is visually represented in Figure 3 below. Since our friend is also traveling in an Eulerian Closed Circuit, the actual delivery time he spends is therefore same as our optimal time – 296 minutes. The only difference between the actual route and our route is the order of vertices visited. However, if we take into account of his break time and traffic time, our friend will spend up to 328 minutes (about 5.5 hours).

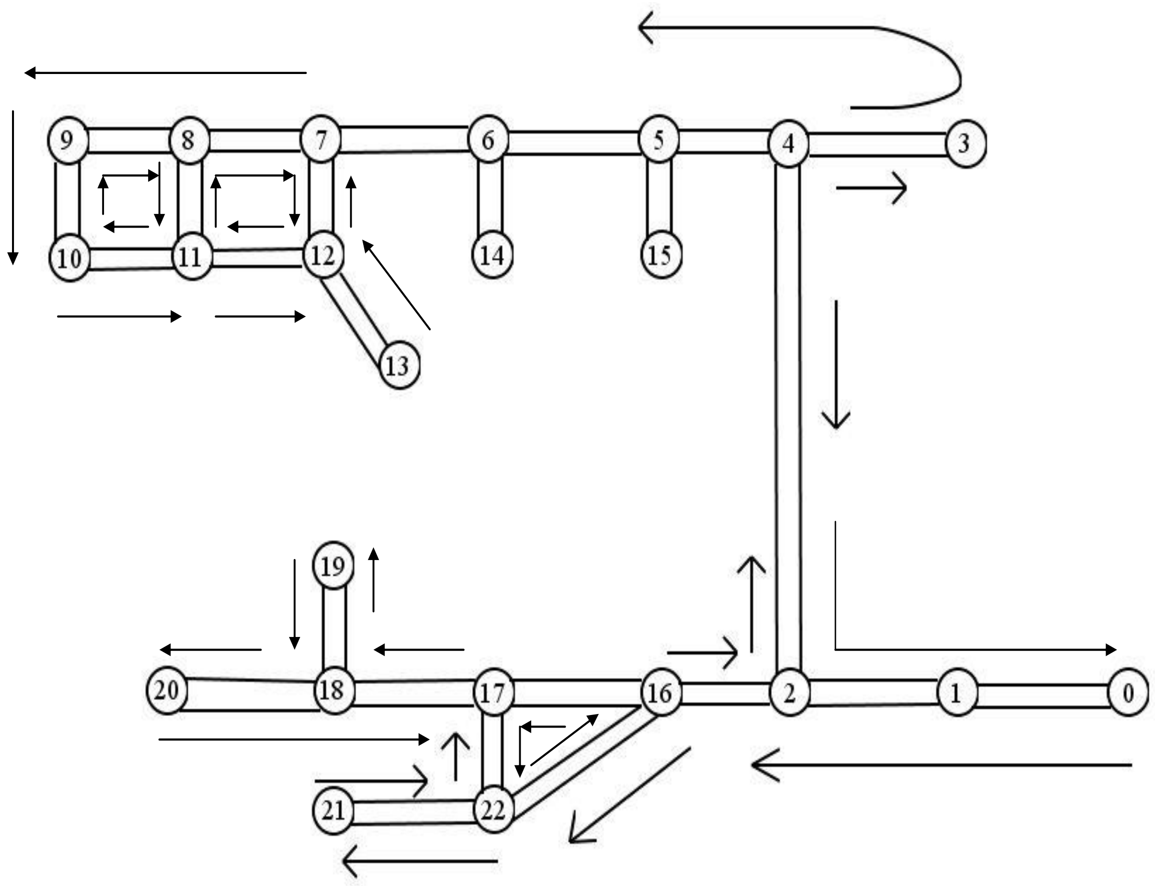

Figure 3: Visual Presentation of the Actual Route

Conclusion

An Eulerian Closed Circuit is the optimal delivery route for our mailman, because it only requires our mailman to travel each edge once; therefore, it minimizes the delivery time. We found the optimal path for our mailman by the CPP algorithm, but one may tend to use Fleury's algorithm to solve a simple Eulerian circuit problem [6]. However, Fleury's algorithm requires a lot of handwork-computation, and it may not be a good choice to solve problems with large amount of data. On the other hand, Dr. Thimbleby's CPP algorithm provides a powerful tool for solving the Chinese Postman Problem and the problems alike.

References

1. Weisstein, Eric. "Floyd-Warshall Algorithm", *Wolfram MathWorld*. Retrieved 13, November 2009.

2. Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 2001, Chapter 16 "Greedy Algorithms".

3. Perkins, Patrick. "Eulerian Closed Circuit", "multigraph", "degree of a vertex", *Math 381 Course Notes*, January, 2010.

4. Sokkalinaam, P.T., "*New Polynomial-time cycle-canceling algorithm for minimum-cost flows*", 2000, vol. 36, pp. 55-63, John Wiley & Sons, New York, NY

5. Thimbleby, Harold. "*The Directed Chinese Postman Problem*"

6. Skiena, S. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica.* Reading, MA: Addison-Wesley, 1990.

Appendix

Here is the full body of the CPP algorithm with necessary modification in order to produce our graph.

```
import java.io.*;

import java.util.*;

public class cppMain {

   public static void main(String[] args)
   {

   CPP G = new CPP(23); // create a graph of 23 vertices
```

```
        // add the arcs for the graph

        G.addArc("e", 0, 1, 15).addArc("e", 1, 0, 15).addArc("e", 1, 2, 6.5).addArc("e", 2,
1, 2.5).addArc("e", 2, 4, 6).addArc("e", 4, 2, 6)
                .addArc("e", 4, 3, 5).addArc("e", 3, 4, 2).addArc("e", 4, 5, 3).addArc("e",
5, 4, 5).addArc("e", 5, 15, 3.5).addArc("e", 15, 5, 3.5)
                .addArc("e", 5, 6, 1.5).addArc("e", 6, 5, 2.5).addArc("e", 6, 14,
6).addArc("e", 14, 6, 6).addArc("e", 6, 7, 3.5).addArc("e", 7, 6, 3)
                .addArc("e", 7, 8, 2).addArc("e", 8, 7, 2).addArc("e", 7, 12, 3).addArc("e",
12, 7, 3).addArc("e", 8, 11, 1).addArc("e", 11, 8, 1)
                .addArc("e", 8, 9, 3.5).addArc("e", 9, 8, 3.5).addArc("e", 9, 10,
2.5).addArc("e", 10, 9, 2.5).addArc("e", 10, 11, 4).addArc("e", 11, 10, 4)
                .addArc("e", 11, 12, 2.5).addArc("e", 12, 11, 2.5).addArc("e", 12, 13,
10).addArc("e", 13, 12, 10).addArc("e", 2, 16, 5.5).addArc("e", 16, 2, 23)
                .addArc("e", 16, 17, 10).addArc("e", 17, 16, 10).addArc("e", 16, 22,
15).addArc("e", 22, 16, 15).addArc("e", 17, 22, 12.5).addArc("e", 22, 17, 12.5)
                .addArc("e", 17, 18, 3).addArc("e", 18, 17, 15.5).addArc("e", 18, 19,
2.5).addArc("e", 19, 18, 2.5).addArc("e", 18, 20, 0.5).addArc("e", 20, 18, 1)
        .addArc("e", 21, 22, 12).addArc("e", 22, 21, 3);

        G.solve(); // find the CPT

        G.printCPT(0); // print it, starting from vertex 0

        System.out.println("Cost = "+G.cost());
        }
}

public class CPP
{
        int N; // number of vertices
        int delta[]; // deltas of vertices
        int neg[], pos[]; // unbalanced vertices
        int arcs[][]; // adjacency matrix, counts arcs between vertices
        Vector label[][]; // vectors of labels of arcs (for each vertex pair)
        int f[][]; // repeated arcs in CPT
        double c[][]; // costs of cheapest arcs or paths
        String cheapestLabel[][]; // labels of cheapest arcs
        boolean defined[][]; // whether path cost is defined between vertices
        int path[][]; // spanning tree of the graph
        float basicCost; // total cost of traversing each arc once

        void solve()
        {
                leastCostPaths();
                checkValid();
                findUnbalanced();
                findFeasible();
                while( improvements() );
        }     // allocate array memory, and instantiate graph object

        CPP(int vertices)
        {
                if( (N = vertices) <= 0) throw new Error("Graph is empty");
                delta = new int[N];
                defined = new boolean[N][N];
                label = new Vector[N][N];
                c = new double[N][N];
                f = new int[N][N];
                arcs = new int[N][N];
                cheapestLabel = new String[N][N];
                path = new int[N][N];
                basicCost = 0;
        }

        CPP addArc(String lab, int u, int v, double cost)
        {
                if( !defined[u][v] ) label[u][v] = new Vector();
                label[u][v].addElement(lab);
                basicCost += cost;
                if( !defined[u][v] || c[u][v] > cost )
                {
                        c[u][v] = cost;
                        cheapestLabel[u][v] = lab;
                        defined[u][v] = true;
                        path[u][v] = v;
                }
                arcs[u][v]++;
                delta[u]++;
                delta[v]--;
                return this;
        }

        void leastCostPaths()
        {
                for( int k = 0; k < N; k++ )
                        for( int i = 0; i < N; i++ )
                                if( defined[i][k] )
                                        for( int j = 0; j < N; j++ )
                                                if( defined[k][j]
                                                        && (!defined[i][j] || c[i][j] >
c[i][k]+c[k][j]) )
```

```
                                        {       path[i][j] = path[i][k];
                                                c[i][j] = c[i][k]+c[k][j];
                                                defined[i][j] = true;
                                                if( i == j && c[i][j] < 0) return; //
stop on negative cycle
                                        }
        }

        void checkValid()
        {       for( int i = 0; i < N; i++ )
                {       for( int j = 0; j < N; j++ )
                                if( !defined[i][j] ) throw new Error("Graph is not strongly
connected");
                        if( c[i][i] < 0) throw new Error("Graph has a negative cycle");
                }
        }

        void findUnbalanced()
        { int nn = 0 , np = 0 ; // number of vertices of negative/positive delta

                for( int i = 0; i < N; i++ )
                        if( delta[i] < 0) nn++;
                        else if( delta[i] > 0) np++;

                        neg = new int[nn];
                        pos = new int[np];
                        nn = np = 0 ;
                for( int i = 0; i < N; i++ ) // initialise sets
                        if( delta[i] < 0) neg[nn++] = i;
                        else if( delta[i] > 0) pos[np++] = i;
        }

        void findFeasible()
        { // delete next 3 lines to be faster, but non-reentrant
                int delta[] = new int[N];
                for( int i = 0; i < N; i++ )
                        delta[i] = this.delta[i];

                for( int u = 0; u < neg.length; u++ )
                { int i = neg[u];
                        for( int v = 0; v < pos.length; v++ )
                        { int j = pos[v];
                                f[i][j] = -delta[i] < delta[j]? -delta[i]: delta[j];
                                delta[i] += f[i][j];
                                delta[j] -= f[i][j];
                        }
```

```
		}
	}

	boolean improvements()
	{ CPP residual = new CPP(N);
		for( int u = 0; u < neg.length; u++ )

		{ int i = neg[u];
			for( int v = 0; v < pos.length; v++ )
			{ int j = pos[v];
				residual.addArc(null, i, j, c[i][j]);
				if( f[i][j] != 0) residual.addArc(null, j, i, -c[i][j]);
			}
		}
		residual.leastCostPaths(); // find a negative cycle
		for( int i = 0; i < N; i++ )
			if( residual.c[i][i] < 0) // cancel the cycle (if any)
			{ int k = 0 , u, v;
				boolean kunset = true;
				u = i; do // find k to cancel
				{ v = residual.path[u][i];
					if( residual.c[u][v] < 0&& (kunset || k > f[v][u]) )
					{ k = f[v][u];
						kunset = false;
					}
				} while( (u = v) != i );
				u = i; do // cancel k along the cycle
				{ v = residual.path[u][i];
					if( residual.c[u][v] < 0) f[v][u] -= k;
					else f[u][v] += k;
				} while( (u = v) != i );
				return true; // have another go
			}
		return false; // no improvements found
	}

	float cost()
	{ return basicCost+phi();
	}
	float phi()
	{ float phi = 0;
		for( int i = 0; i < N; i++ )
			for( int j = 0; j < N; j++ )
				phi += c[i][j]*f[i][j];
		return phi;
	}
```

```
        static final int NONE = -1; // anything < 0
        int findPath(int from, int f[][]) // find a path between unbalanced vertices
        { for( int i = 0; i < N; i++ )
                        if( f[from][i] > 0) return i;
                return NONE;
        }
        void printCPT(int startVertex)
        { int v = startVertex;
                // delete next 7 lines to be faster, but non-reentrant
                int arcs[][] = new int[N][N];
                int f[][] = new int[N][N];
                for( int i = 0; i < N; i++ )
                        for( int j = 0; j < N; j++ )
                        { arcs[i][j] = this.arcs[i][j];
                                f[i][j] = this.f[i][j];
                        }
                while( true )
                { int u = v;
                        if( (v = findPath(u, f)) != NONE )
                        { f[u][v]--; // remove path
                                for( int p; u != v; u = p ) // break down path into its arcs
                                { p = path[u][v];
                                        System.out.println("Take arc "+cheapestLabel[u][p]
                                                +" from "+u+" to "+p);
                                }
                        }
                        else
                        { int bridgeVertex = path[u][startVertex];
                                if( arcs[u][bridgeVertex] == 0)
                                        break; // finished if bridge already used
                                v = bridgeVertex;
                                for( int i = 0; i < N; i++ ) // find an unused arc, using bridge
    last
                                        if( i != bridgeVertex && arcs[u][i] > 0)
                                        { v = i;
                                                break;
                                        }
                                arcs[u][v]--; // decrement count of parallel arcs
                                System.out.println("Take arc
    "+label[u][v].elementAt(arcs[u][v])
                                        +" from "+u+" to "+v); // use each arc label in turn
                        }
                }
        }
    }
```

Work Contribution:

Michael – my contribution for the project includes data gathering; compiling the first and final draft; making the power point presentation; author of the “math model”, “solution”, which includes finding the CPP algorithm; and “result” part of this paper.

Wen – I have several tasks as parts of the project. For the proposal at the beginning of this quarter, after discussing with my group members, I wrote a paragraph describing how we were going to solve our problem, and list several algorithms we might use. We had one or two group meetings per week after the proposal, working on the first draft. My job was to drawing graphs, finding references, writing the conclusion part of the article. Collaborating with Michael and Maurice, we finished the first draft. Our project needs a lot of works to do, and teamwork is important. It is fun to work together with people and get our project done.

Maurice – I lead and conducted our project presentation, answered questions from Professor Perkins (Eulierian circuit exits iff every node is even) and students. Also I drew (with node details and direction of flow) original route graph and the result route using LaTeXdraw. These graphs were used in the final draft of the project as well as in the presentation. I made the suggestion that we make a comparison between the route we found and the original route. Additionally I wrote the problem statement as well as the abstract and searched for numerical solution methods.