# OAR Lib: An Open Source Arc Routing Library

Oliver Lum

Department of Applied Mathematics and Scientific Computation (AMSC), University of Maryland
oliver@math.umd.edu

Bruce Golden

Robert H. Smith School of Business, University of Maryland
bgolden@rhsmith.umd.edu

September 2013

**Abstract**

We provide a flexible graph architecture, and an implementation of solvers to some of the most ubiquitous problems in the field of Arc Routing, (The Chinese Postman Problem on undirected, directed, mixed, and windy graphs, as well as the Rural Postman Problem on directed and windy graphs). The project is completed in java, is open source, and is hosted at `https://github.com/Olibear/ArcRoutingLibrary`.

## 1 Introduction

Broadly speaking, in the realm of vehicle routing, there are two classes of problems; node routing problems, and arc routing problems. In the former, the goal is to visit some (sub)set of nodes in a graph while minimizing some cost (or maximizing some reward) function. Likewise, in the latter, we seek to optimize some objective function, but this time the requirement is that a (sub)set of edges gets traversed. For example, the well-known Traveling Salesperson Problem is a node routing problem that requires the construction of a cycle of minimal cost that visits every node in the graph, (with costs associated with the traversal of each edge). Meanwhile, the analogous problem in arc-routing is the Chinese Postman Problem (CPP), where a candidate cycle must traverse every edge in the graph.

Unfortunately, the vast majority of outstanding network optimization problems have been shown to be outside of P, (the class of provably polynomial-time solvable problems), which means that it is unlikely that they can be solved to optimality in a computationally tractable manner. Still, since such problems are nearly ubiquitous in industry, (with transportation / infrastructure networks, server topologies, and social networks all benefitting from advances in the field), there exists a vast literature devoted primarily to devising efficient (meta)heuristics that aim to get close to the optimum without prohibitive computational effort. The virtues of a particular heuristic are usually presented via an analysis of two factors: speed, and proximity to optimality. Of course, in order to make meaningful comparisons, researchers typically present and solve benchmark instances that showcase their algorithm's performance relative to that of an established alternative.

However, there is another element of variability that is less frequently accounted for. Namely, differences in implementation of the same algorithm can be responsible for discrepancies in results. Given that many of these heuristics proceed by decomposing the more complicated problem into a series of instances of easier problems, (and solving these simpler ones to optimality using a known algorithm), it is especially important to have standardization with respect to these easier problems so that performance can be attributed solely to the merits of the heuristics themselves.

## 2 Approach

To that end, we have created an open source code library that provides exactly such functionality. More specifically, this library will feature solvers for the following problems: the CPP on a directed graph (DCPP), the CPP on an undirected graph (UCPP), the CPP on a mixed graph (MCPP), the CPP on an undirected graph with directionally asymmetric costs (WPP for Windy Postman Problem), and the Rural Postman Problem on directed and windy graph (where not all arcs are required to be traversed in the solution). For each of these problems, if it is not possible to efficiently solve it to optimality, then we present two solvers; one of the more well-known heuristics, and one that is closer to the state-of-the-art. Obviously, if a problem is solvable in polynomial time, we implement the exact algorithm precisely. For the details of each specific algorithm beyond what follows, consult references (15) for the Directed CPP, (4) for the Undirected CPP, (9) and (17) for the Mixed CPP, (16) and (10) for the WPP, (2) for the Directed RPP, and (1) for both Windy RPP heuristics.

### 2.1 Definitions

A graph G is defined as a double $(V, L)$ where $V$ is a set of vertices (also referred to here as nodes), and $L$ a multiset of links. Typically, vertices are indexed naively (i.e. 1,2,3, etc.) while a link is represented as an ordered pair $(i, j)$ where both $i$ and $j$ are members of the vertex set, (we do not consider hypergraphs here). We call a link an edge if it is undirected (that is, it can be traversed from $i$ to $j$, and from $j$ to $i$), and an arc if it is directed (i.e. it can only be traversed from $i$ to $j$). In the case of arcs, the first element of the ordered pair is referred to as the *tail*, while the second is referred to as the *head*. For clarity's sake, an undirected graph is one in which all elements of the link set are undirected, and is usually represented as $(V, E)$. Accordingly, a directed graph has only arcs (and is represented $(V, A)$. We call a graph mixed if it is allowed to have links of both types, and use the representation $(V, E, A)$. Finally, we refer to a graph as being windy if it is undirected, *and* has asymmetric traversal costs, (i.e. it is not necessarily the case that the cost of going from vertex $i$ to vertex $j$ = the cost of going from vertex $j$ to vertex $i$).

It is important to notice that a windy graph is, for all practical purposes, capable of modelling every other type of graph in the following manner: for undirected graphs, simply set $c_{ij} = c_{ji}$ for every edge; for directed graphs, set $c_{ij}^W = c_{ij}^A$ and $c_{ji}^W = N$ where $N$ is some prohibitively high value (certainly $>> \sum_{a \in A} a_{ij}$ ); for mixed graphs, it follows directly. Thus, any *reasonably good* method applicable to the windy cases of the problems we are investigating is immediately applicable to all other types of graphs as well. We have to caveat *reasonably good* because otherwise the method could theoretically traverse edges representing arcs in the wrong direction, but this would naively be able to be improved by replacing these infeasible moves with shortest paths given our assumption about the prohibitively high costs.

A graph has the property of being *strongly connected* if it's possible to reach any vertex from any other vertex, (more precisely, for any pair of vertices $i$ and $j$, it is possible to construct an ordered list of links $(i_0, j_0), (j_0, j_1), (j_1, j_2)...(j_{k-1}.j_k)$ where $i_0 = i$ and $j_k = j$ that constructs a valid path from vertex $i$ to vertex $j$).

A graph is called *Eulerian* if and only if there exists an Eulerian cycle (a path through the graph that traverses every link in the graph exactly once and returns to its starting vertex). The various criteria for being Eulerian are as follows:

- *Undirected*: A undirected graph is Eulerian $\iff$ every node has even degree.
- *Directed*: A directed graph is Eulerian $\iff$ every node has in-degree = out-degree (a property known as symmetry).
- *Mixed*: A mixed graph is Eulerian $\iff$ every node has even degree, and the graph is balanced, (for any subset $S$ of $V$, |number of arcs from $S$ to $V \setminus S$ - number of arcs from $V \setminus S$ to $S$| $\leq$ number of edges from $S$ to $V \setminus S$). A sufficient condition to be Eulerian for a mixed graph is for every node to have even degree, and in-degree = out-degree.

We call a graph $G_2 = (V_2, L_2)$ an augmentation of the graph $G_1 = (V_1, L_1)$ if $V_1 \subseteq V_2$, $L_1 \subseteq L_2$, and $\forall l_{ij}^2 \in L_2, (\exists\ l_{ij}^1 \in L_1 \quad \& \quad cost(l_{ij}^2) = cost(l_{ij}^1))$. Colloquially, this means that every link in the original graph appears in the augmentation, and that the augmentation only includes copies of links in the original graph.

Finally, we introduce, and distinguish between several notions of *degree* of a vertex. In an undirected graph, the degree of a vertex is simply the number of edges incident on the vertex. In a directed graph, the *in-degree* of a vertex $v$ is the number of arcs $a \in A$ for which $v$ is the head, and the *out-degree* of a vertex $v$ is the number of arcs for which $v$ is the tail. Lastly, for a mixed graph, our definition of in-degree and out-degree remains the same, (i.e. it only takes into account arcs in the graph), while our definition of degree incorporates both edges and arcs, (i.e. pretend as though all arcs were undirected).

The problems we intend to solve all fall under the following two categories:

- *The Chinese Postman Problem:* Given a graph $G$ (either directed, undirected, mixed, or windy) and a set of costs $c_{ij}$ (associated with traversing the link $(i,j)$), the task is to find a closed path that traverses each link (edge or arc) at least once and minimizes the total traversal cost. This problem is solved in two steps: the first step (and the most computationally involved) is that of finding an Eulerian augmentation of the original graph; the second step is finding the Eulerian cycle in this augmented graph. The second step can be performed easily and efficiently in polynomial time (Hierholzer's algorithm) (11).
- *The Rural Postman Problem:* Given a graph $G$ (in our case, directed, or windy), a set of required links $L_R \subseteq L$, and a set of costs $c_{ij}$ (associated with traversing the link $(i,j)$), the task is to find a closed path that traverses each *required* arc at least once and minimizes the total traversal cost. This problem is solved in two steps: the first step is that of finding an Eulerian augmentation of the original graph; the second step is finding the Eulerian cycle in this augmented graph. The second step can be performed easily and efficiently in polynomial time (Hierholzer's algorithm).

Other common notation includes:

- $\delta(v)$ = out-degree − in-degree
- $D^+$ and $D^-$ are the set of vertices with $\delta(v) > 0$ and $\delta(v) < 0$.
- $\mathbb{Z}_+^0$ is the set of non-negative integers, ($\mathbb{N}$).

## 2.2 Common Algorithms

In the course of developing the library, certain graph algorithms are fairly prevalent and useful to have split off and available for all solvers / users of the library to have access to. Currently, these include:

- Floyd-Warshall All-Pairs Shortest Paths Algorithm (2.2.3)
- Dijkstra's Single-Source Shortest Paths Algorithm with Priority Queue Implementation (2.2.4)
- Min-Cost Flow Algorithms (Cycle-Cancelling, and Successive-Shortest Paths now implemented) (2.2.1, 2.2.2)

We detail these algorithms in the following section.

### 2.2.1 The Cycle-Cancelling Min Cost Flow Algorithm

In order to obtain a minimum cost solution to the flow problem, we have several options. Currently, we have included both a cycle-cancelling algorithm, as well as a successive shortest paths algorithm. It is worth noting that our Shortest Successive Paths (SSP) implementation significantly outperforms our cycle-cancelling implementation, so while we review the algorithmic details of both here, only the latter is currently called by our solvers.

The cycle-cancelling algorithm proceeds by forming residual graphs, detecting the presence of negative cycles, and cancelling said cycles by pushing flow around them. Before we may form the residual graph, though, we must first find a feasible solution to the flow problem, (that is, find any acceptable flow that satisfies the demands, irrespective of cost, as in Figure 1). Since the flow problem is usually formulated on capacitated graphs (where edges or arcs can only support a finite amount of traffic), the well-known algorithms all take this into consideration (Ford-Fulkerson is the classical algorithm for solving the min cost flow problem). However, in all of these problems, we are working with uncapacitated graphs, and so we may greedily construct a feasible path in the following manner:

- Select any node with excess in-degree, call it $u$.
- Select any node with excess out-degree, call it $v$.
- Add $\min |\delta(u)|, |\delta(v)|$ shortest paths from $u$ to $v$.

Assuming we have shortest paths (which we shall explain how to find later), if we simply repeat this procedure until all nodes are symmetric, then we shall have constructed a feasible solution to the flow problem.

Now, we must introduce the notion of a residual graph (Figure 2). Given a graph $G$, and a feasible solution $X$ to a flow problem, we may construct its residual graph $G_X$ as follows:

- Every vertex in $G$ is also a vertex of $G_X$.

- For each arc $(i, j)$ with capacity $r_{ij}$ and cost $c_{ij}$ in the original graph $G$, add two arcs to $G_X$: one arc $(i, j)$ with capacity $r_{ij} - f_{ij}$ and cost $c_{ij}$, and one arc $(j, i)$ with capacity $f_{ij}$ and cost $-c_{ij}$. Here, $f_{ij}$ is the amount of flow pushed along the original arc in $X$.

Now, once we have the residual graph, we simply compute shortest paths from a vertex to itself. If this value is negative, we know a negative cycle exists. We then cancel this cycle by pushing as much flow as possible in the opposite direction (this is trivial to find; it is simply the minimum capacity along the cycle). Once no more negative cycles exist, the algorithm terminates, and the directed graph we are left with represents a least cost solution to the flow problem (Figure 3).

### 2.2.2 The Successive Shortest Paths Algorithm

The Successive Shortest Paths Algorithm proceeds by converting the problem into a single-source, single-sink flow problem, and then repeatedly pushing flow along the shortest possible path from the source to the sink. In order to perform this conversion, a single source vertex, and a single sink vertex is added to the network. Then, for each vertex $v_s$ with supply, a zero-cost arc is added from the source vertex to $v_s$. This arc has capacity equal to the supply of $v_s$. Similarly, for each vertex with demand $v_d$, a zero-cost arc is added from $v_d$ to the sink, with capacity equal to the demand of $v_d$.

From here, the process is conceptually simple; we calculate a shortest path from source to sink using Dijkstra's Algorithm for single-source shortest paths, and then push as much flow as possible along that path. Just as in the Cycle-Cancelling Algorithm, we then form the residual graph given our new flow, and then repeat. The algorithm terminates when there are no more paths from source to sink. At this point, we obtain our flow solution by simply removing the source and sink vertex, (as well as all incident arcs). Figures 4 and 5 illustrate one iteration of the process.

The only complication that remains is to resolve a complication that arises to limitations with Dijkstra's Algorithm; namely, it's inability to deal with negative edge costs. Obviously, negative edge costs arise frequently in residual graphs, so this concern is non-trivial. However, we avoid having negative costs by reducing the arc costs at each iteration of the method in the following manner. Consider an arc $a_{ij}$, with original cost $c_{ij}$. Then we set its cost to $c_{ij} + dist(j) - dist(i)$ where $dist(j)$ is the shortest path distance from the source to vertex j.

Since we have to repeatedly call Dijkstra's Algorithm at worst on the order of $O(nB)$ times, where $B$ is an upper bound on the supply of any node, then we can guarantee a complexity of $O(n^2 B \log n + nmB)$

### 2.2.3 The Floyd-Warshall All-Pairs Shortest Paths Algorithm

There are variety of algorithms to calculate shortest paths, but because of our choice of supporting algorithms (particularly the min cost flow algorithm), the Floyd-Warshall algorithm (with path reconstruction) will be one of the ideal ones to implement.

The Floyd-Warshall algorithm proceeds very simply; we construct two $|V| \times |V|$ matrices. The first matrix $D$, will store shortest path distances, where $d_{ij}$ will represent the shortest path cost of getting from vertex $i$ to vertex $j$. The second matrix $P$ will store information about exactly what the path is; $p_{ij}$ will hold the next node in the shortest path from node $i$ to node $j$.

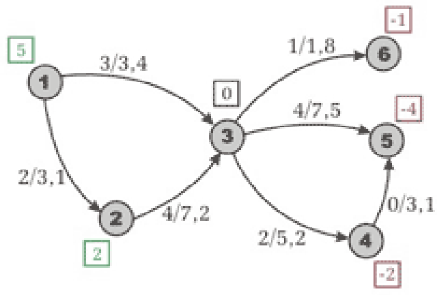
initially, diagonal elements of $D$ are set to 0, and for each edge $(i.j)$ in the graph, $d_{ij}$ is set to $c_{ij}$, (we are assuming this is not a multigraph; otherwise, we take the cheaper of the two). Every other entry in $D$ is set to infinity (some arbitrarily high number that is greater than the sum of the edge costs), while every entry in $P$ is set to some dummy value, ($-1$ will suffice). Finally, we iterate through all vertex triples $(i, j, k)$, and ask whether the $d_{ik} + d_{kj} < d_{ij}$. If so, the new, cheaper cost is stored in $d_{ij}$, and $p_{ij}$ gets set to $k$ to signify that, in the shortest path from node $i$ to node $j$, the next node in the walk is $k$.

Since all we are doing is traversing ordered triples $(i, j, k)$ where each is allowed to be any number from 1 to $n$, then clearly the asymptotic complexity is $O(n^3)$.

### 2.2.4 Dijkstra's Single-Source Shortest Paths Algorithm

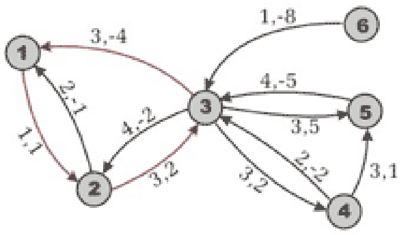
Similarly to the Floyd-Warshall All-Pairs Shortest Paths Algorithm, Dijkstra's algorithm proceeds by maintaining an array of distances and path information that we update by examining links in the graph. However, since we are only concerned with a single vertex as the starting point, this cuts down the complexity of the problem by a factor of roughly $n$.

More precisely, the algorithm begins by initializing all distances to infinity, and then examining each of the neighbors of the starting vertex, and assigning them distances equal to the cost of the edge between them and the starting vertex. Each of these neighbors is then added to a list which contains vertices left to be examined. The algorithm proceeds by choosing the vertex in the list with the cheapest distance



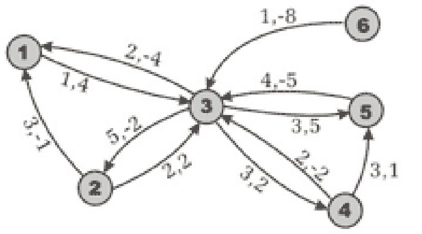
(a)

Figure 1: A flow network on which we may run the Cycle-Cancelling algorithm. For each arc, the numbers shown represent (current flow) / (arc capacity), (cost). Meanwhile, the numbers in green represent the amount of supply that the associated vertex has, and the numbers in red indicate demand. Here we see an initial, non-optimal feasible flow which cycle-cancelling will iteratively improve upon.



(b)

Figure 2: The residual graph induced by the initial flow in the previous figure. Here, we see (highlighted in red) the presence of a negative cycle, around which we are entitled to push 1 flow around (restricted by the arc from vertex 1 to vertex 2).



(c)

Figure 3: The resulting residual graph after 1 flow has been pushed around the negataive cycle in the previous figure.

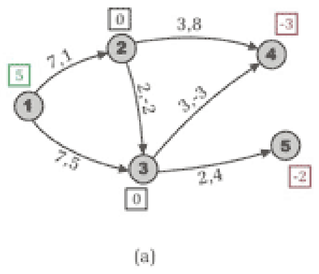


Figure 4: A flow network on which we may run the Successive Shortest Paths algorithm.

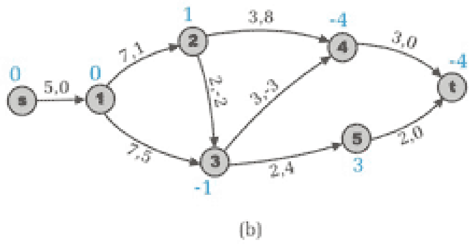


Figure 5: The modified flow network, where each vertex in the original flow network that had positive supply has been connected to a new source vertex, and each vertex in the original flow network that had positive demand has been connected to the new sink vertex. These added connections all have zero cost, and capacity equal to the supply/demand of the node in the original graph, (e.g. the connection to vertex 1 has capacity 5).

associated with it, and then repeats the process of examining its neighbors and updating the distance and path arrays. Each of these neighbors is added to our list, and when we have finished examining all of a vertex's neighbors, it is ejected from our list. The algorithm terminates when all vertices have been exhausted.

Since the algorithm considers each edge once, and adds vertices to a priority queue that requires $O(n \log n)$, then we have that the algorithm is $O(m + n \log n)$. This process is depicted in Figure 6.

## 3 Problem Setting and Algorithms

Armed with the utility functions specified in Section 2, we now describe the solvers implemented in the library for the aforementioned fundamental problems in arc routing.

### 3.1 The Directed Chinese Postman Problem

In light of the notion of an Eulerian graph, (as will be the strategy in general), it suffices to find a least-cost way of augmenting the original graph in order to make it Eulerian. Obviously, on the augmented graph, the Euler cycle (that is guaranteed to exist) is an optimal solution to the CPP. With this in mind, we formulate the problem as an integer program that attempts to minimize the cost of the arcs we are adding.

Problem Statement:

$$\text{minimize} \qquad \sum_{i \text{ or } j \in \{D^+ \cup D^-\}} c_{ij} x_{ij}$$

subject to:

$$\sum_{j \in D^+} x_{ij} = -\delta(i), \forall i \in D^- \qquad (1)$$

$$\sum_{i \in D^-} x_{ij} = \delta(j), \forall j \in D^+ \qquad (2)$$

$$x_{ij} \in \mathbb{Z}^0_+ \qquad (3)$$

Intuitively, the variable $x_{ij}$ represents the number of times we've added a shortest path from node $i$ to node $j$ in the augmented graph (with $c_{ij}$ is the shortest path cost). Thus, the objective function is the total *additional* cost incurred by the augmentation. Meanwhile, constraints (1) and (2) ensure that, once we've added these shortest paths, the graph is completely symmetric.

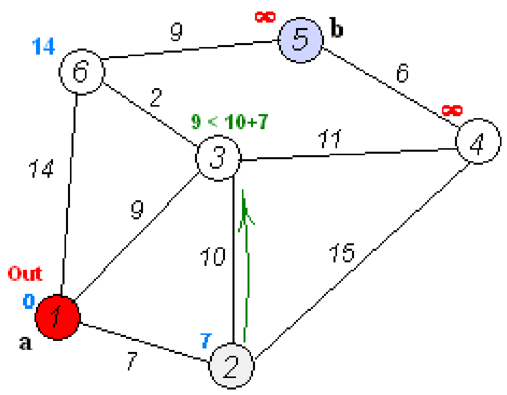


Figure 6: An intermediate step of Dijkstra's Algorithm, where vertex 1 is the source vertex, and we have already assigned distances to nodes 2, 3, and 6. Once we have examined all of 1's neighbors, we then choose the vertex with the least distance from the source (that hasn't yet been interrogated), and repeat the process. In this case, 2 was selected as the next node for interrogation, and the algorithm is currently considering whether or not going from source $\rightarrow$ vertex 2 $\rightarrow$ vertex 3 beats the previously recorded best distance of 9. It does not, so vertex 3 will retain its distance of 9.

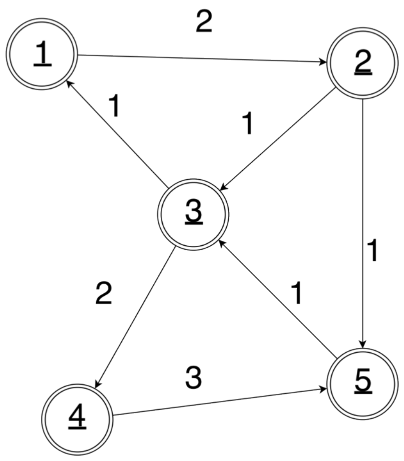


Figure 7: An example of a directed graph, upon which the the Directed Chinese Postman Problem may be solved.

### 3.1.1 An Exact Algorithm for the Directed Chinese Postman Problem (15)

Recall that for a directed graph, we require that, in order to be Eulerian, each node must exhibit symmetry. To that end, we first identify the net degree of each node in our graph (the sign convention is not so important, but we'll define it as $\delta(v)$ = out-degree − in-degree). This leaves us with three classes of nodes. Those that originally have an excess of outgoing arcs, those that have an excess of incoming arcs, and nodes that are already balanced. This last group of nodes requires no additional consideration, since they are currently balanced, and any paths we add to the graph from one unbalanced node to another will keep them balanced. Thus, our goal is simply to find a least cost way to add a series of paths to the graph from nodes with too many incoming arcs to one with too many outgoing arcs at minimal cost. To do this, we use a min cost flow algorithm to solve the emergent flow problem. In a flow problem on a graph, each node is assigned a demand, (negative demand corresponds to supply), and a least cost way is sought of satisfying these demands, (where edge costs reflect per unit transportation costs). In our case, the demand of node v corresponds exactly to $\delta(v)$.

Finally, in order to actually obtain the tour, (and not simply its cost), we use Hierholzer's algorithm, which greedily moves from vertex to vertex on the augmented graph, deleting edges once they have been traversed. We continue until we return to the starting vertex, at which point our current solution contains a cycle. Then, check to see if there are any remaining edges incident to a previously visited vertex $v$. If not, then we are done; if so, then repeat the process, with $v$ as the new starting vertex. Once this process terminates, we simply merge all the subcycles to get the full tour. Figures 7 and 8 depict the process.

### 3.1.2 Pseudocode

```
Result: An optimal directed postman tour
        over the input graph $G$.
1  foreach vertex $v \in V$ do
2      $\delta(v) \leftarrow$ in-degree − out-degree;
3      $supply(v) \leftarrow \delta$
4  end
5  Solve a min-cost flow over $G$;
6  for $i \leftarrow 1$to$|E|$ do
7      for $j \leftarrow 1$to$flow(e_i)$ do
8          Add copy of $e_i$ to $G$;
9      end
10 end
11 Return Hierholzers($G$);
```

## 3.2 The Undirected Chinese Postman Problem

Problem Statement:

$$\text{minimize} \quad \sum_{(i,j)\in E} c_{ij}x_{ij}$$

subject to:

$$\sum_{(i,j)\in E_v} (x_{ij}+1) \equiv 0 \; mod \; 2, \forall v \in V \quad (4)$$

$$x_{ij} \in \mathbb{Z}^0_+ \quad (5)$$

Here, $x_{ij}$ represents the number of *additional* copies of edge $(i,j)$ in our augmented graph. As before, we wish to minimize the added cost, while ensuring evenness of the augmented graph, (constraints (1) and (2) achieve this).

### 3.2.1 An Exact Algorithm For The Undirected Chinese Postman Problem (4)

The algorithm for the Undirected Chinese Postman Problem is extremely similar to that for the directed variant. We know that an Euler Tour must exist on an undirected graph if every node has even degree, (intuitively, every time we enter a node, we may exit it using a new edge). Thus, the only thing that changes here is that, rather than worrying about in-degree and out-degree, we simply seek to pair nodes of odd degree together in a least cost way (so rather than solving a more complex flow problem, we may solve a min cost perfect matching problem). It suffices to identify all of the odd-degree nodes, and carry out a matching algorithm on those (trivially, there will be an even number of them, so parity is not a concern) to solve the undirected Chinese Postman Problem.

### 3.2.2 Pseudocode

```
Result: An optimal undirected postman tour
        over the input graph $G$.
1  $V_{matching} \leftarrow V_{odd}$;
2  $E_{matching} \leftarrow \emptyset$;
3  for $i \in V_{odd}$ do
4      for $j \in V_{odd}$ do
5          Add $e_{ij}$ to $E_{matching}$ with $c_{ij} = sp_{ij}$;
6      end
7  end
8  Let $G_{matching} = (V_{matching}, E_{matching})$;
9  Solve a min-cost matching over $G_{matching}$;
10 for $e_{ij} \in$ matching do
11     Add copy of $e_{ij}$ to $G$;
12 end
13 Return Hierholzers($G$);
```

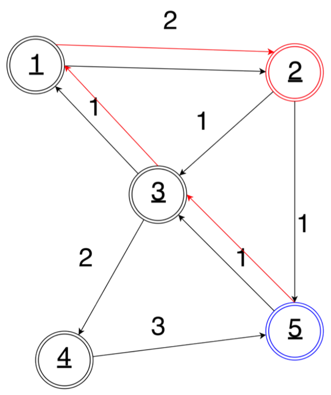


Figure 8: The solution to the DCPP on the previous graph. Blue nodes are identified as belonging to $D^-$, and red nodes to $D^+$ in the initial phase of the algorithm. Arcs added as part of the min-cost flow solution are shown in red.

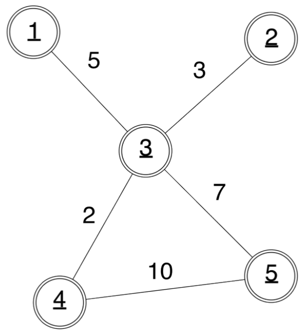


Figure 9: An example of a undirected graph, upon which the the Undirected Chinese Postman Problem may be solved.

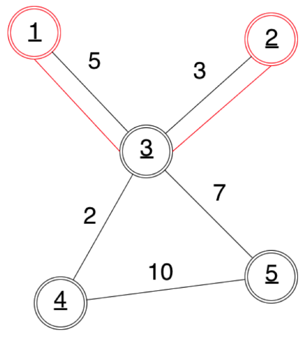


Figure 10: The solution to the UCPP on the previous graph. Red nodes are identified as odd in the initial phase of the algorithm. Edges added as part of the matching solution are shown in red.

Figures 7 and 8 depict this process.

## 3.3 The Mixed Chinese Postman Problem

Problem Statement:

$$\text{minimize} \qquad \sum_{s \in \{A \cup \hat{E} \cup \breve{E}\}} c_s x_s$$

subject to:

$$y'_e + y'_{\tilde{e}} \geq 1, \forall e \in E \qquad (6)$$

$$x_s = y'_s + y_s, \forall s \in A \cup \hat{E} \cup \breve{E} \qquad (7)$$

$$\sum_{s \in S_v^+} x_s - \sum_{s \in S_v^-} x_s = 0, \ \forall v \in V \qquad (8)$$

$$y'_a = 1, \ \forall a \in A \qquad (9)$$

$$y'_e \in \{0,1\}, \ \forall e \in \hat{E} \cup \breve{E} \qquad (10)$$

$$y_s \in \mathbb{Z}_+^0 \qquad (11)$$

First, some notation: $y'_s$ is 0 if link $s$ is never traversed, and 1 if it is; $y_s$ is the number of *additional* times link $s$ is traversed. The set $\hat{E}$ contains edges $e$ that are traversed from $i$ to $j$ in the solution, while the set $\breve{E}$ contains edges $\tilde{e}$ that are traversed from $j$ to $i$. Thus, $x_s$ is the total number of times link s is traversed, and so constraint (1) ensures that each edge is traversed at least once, constraint (2) defines $x_s$, constraint (3) ensures symmetry, constraint (4) ensures that arcs are traversed at least once, and constraints (5) and (6) are the binary constraint for $y'_s$ and the integrality constraint for the $y_s$

### 3.3.1 Even-Symmetric-Even (9)

The first heuristic we plan to implement has the same intuitive motivation as the exact algorithms for the DCPP and UCPP: namely, we try to augment the graph to reach an Eulerian supergraph in which we know we may locate an Euler tour. In order for a mixed graph to be Eulerian, it must fulfill both of the following properties:

- *Evenness*: Each node as an even number of incident links.
- *Balanced*: For each subset of nodes $V$, the number of undirected arcs between $V$ and $V \setminus S$ must be greater than or equal to the difference between the number of arcs from $V$ to $V \setminus S$ and the number of arcs from $V \setminus S$ to $V$. (Intuitively, this second condition ensures that we cannot get 'stuck' in a portion of the graph.)

Prima facie, it is difficult to see how one would easily verify the second property, and so this particular heuristic instead aims to create an even, symmetric graph, (which, in general, is guaranteed to be balanced).

The Even-Symmetric-Even heuristic has three eponymous phrases; in the first, it achieves evenness by carrying out a min-cost matching among the odd-vertices, in the second, it achieves symmetry by using a min-cost flow algorithm on the asymmetric nodes, and in the third, it restores evenness by looking for cycles that may be eliminated safely (because the consist 'mostly' of links that were added in the previous two phases). This process is depicted in Figures 11 and 12.

1. *Phase I, Even*: Solve the UCPP on the original graph, treating all arcs as edges. This produces an augmented graph $G^E$.

2. *Phase II, Symmetric*: Solve a min cost flow problem on $G^E$, treating each edge $(u, v)$ as four arcs: the first two $(u, v)$ and $(v, u)$ with cost equal to the original edge cost and infinite flow capacity; and two $(u, v)$ and $(v, u)$ with zero cost, and flow capacity of 1. If the solution to the flow problem singularly walks edge $(u, v)$, (that is, in the flow solution, arc $(u, v)$ is only traversed once, or arc $(v, u)$ is traversed only once), then we 'orient' the edge in that direction, otherwise it remains as an edge in our output graph $G^S$.

3. *Phase III, Even*: Greedily search for cycles that consist of paths between any odd-degree nodes left in $G^S$ (if there are none, Phase III is unnecessary). Importantly these paths must alternate between only containing arcs / oriented edges added in Phase II, and only containing edges left undirected by Phase II. In this way, we ensure that only the parity of the odd-degree nodes is changed, while also assigning a direction to all remaining undirected edges. There is a chance that no such cycle exists, and that there are still undirected edges, but the graph will be Eulerian at this point, and so we are done. Once we find one of these *alternating paths*, we orient it (either direction will be equivalent) and duplicate arcs/oriented edges along the path that follow the orientation, while deleting arcs that are in the opposite direction. Meanwhile, for the sections of the cycle that consist entirely of undirected edges, we simply orient them in the direction we have chosen to orient the cycle.

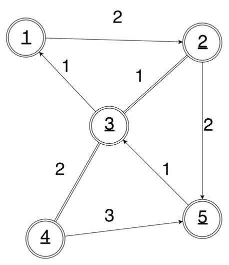


Figure 11: An example of a mixed graph, upon which the the Mixed Chinese Postman Problem may be solved.

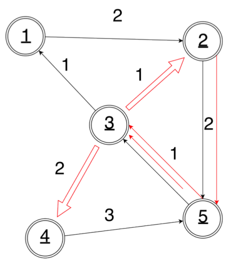


Figure 12: The solution to the MCPP on the previous graph. The red arc $(2, 5)$ is added in the initial Even phase, while all other red links are added in the Symmetric phase. The thicker red arrows are to indicate that the edges in the original graph were oriented in the corresponding direction.

### 3.3.2 Pseudocode

```
Result: An approximately optimal mixed postman tour over the input graph $G$.
1 $G_{even} \leftarrow EvenDegree(G)$;
2 $G_{symm} \leftarrow Symmetric(G_{even})$;
3 $G_{final} \leftarrow EvenParity(G_{symm})$;
4 Return Hierholzers($G_{final}$);
```

### 3.3.3 Shortest Additional Path (17)

While most other heuristics for the MCPP do roughly the same thing as Even-Symmetric-Even, (and then sometimes implement an improvement procedure on the generated solution), the Shortest Additional Path Heuristic (SAPH) performs the bulk of its work on a graph that may not even contain a feasible Euler tour, but manages to ensure that the final output does.

The initial step of SAPH is in fact identical to the *second* phase of the Even-Symmetric-Even heuristic (where the graph is transformed into a symmetric one). The heuristic then proceeds by exploiting two ideas: first, suppose that an edge or arc was added to the original graph, and oriented from node A to node B. Then, if the shortest path cost of going from node A to B is less than the cost of traversing this added link, then we ought to replace said link with the shortest path from A to B (Figure 13).

Second, if an edge was oriented from node A to B, and the two shortest paths have costs that sum to less than zero, then it's advantageous to use $ShortestPath(A \to B), (B \to A), ShortestPath2(A \to B)$. Although this second case may seem like a bizarre one to investigate (since the shortest path costs will generally be positive), it is an important one to consider for the SAPH because we may consider a path from A to B as traversing added arcs in the *opposite* direction (which would correspond to deleting them) and incurring the negative of its cost (Figure 14).

1. Given a mixed graph $G$, generate a graph $G* = (N, M, U)$ and set of added arcs $M*$ by solving Phase II of Even-Symmetric-Even on $G$. Also, generate a graph $G_M = (N, E + E_M, A + A_M)$ by solving Phase I of Even-Symmetric-Even on $G$, where $E_M$ and $A_M$ are the sets of edges and arcs added from the matching.

2. Choose a random edge/arc in $G*$ of type $a, c, d$ or $f$.

3. Initialize two graphs $G^1_{ij} = G$ and $G^2_{ij} = G$

4. Perform *Cost modification 1* on $G^1_{ij}$.

5. Perform *Cost modification 2* on $G^1_{ij}$ and $G^2_{ij}$.

6. Apply the first shortest paths idea to the chosen edge/arc.

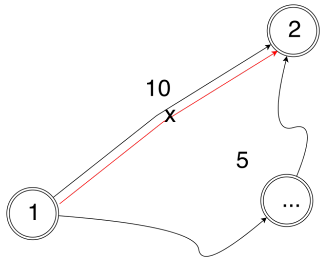


Figure 13: An example of the first SAPH idea, where we replace an added link (red) with a cheaper shortest path.

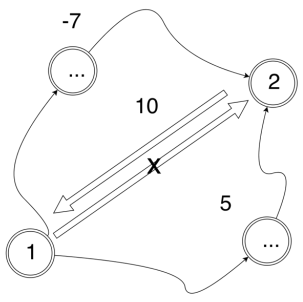


Figure 14: An example of the second SAPH idea, where we reverse the orientation of an edge and add two 'paths' from node $i$ to $j$ which sum to a negative cost.

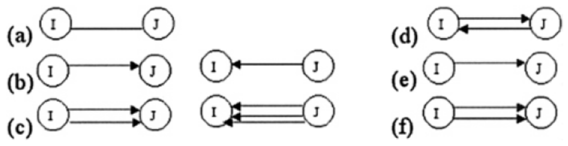


Figure 15: Taken from (17). Edges and arcs in $G$ must end up in one of the following configurations in $G*$:

- If an edge remains undirected, it is of type $a$.
- If an edge gets directed, but not copied, it is of type $b$.
- If an edge gets directed and copied, but all copies are in the same direction, then it is of type $c$.
- If an edge gets copied once, and oriented in the opposite direction as the original, it is of type $d$.
- If an arc is not copied, it is of type $e$.
- If an arc is copied, it is of type $f$

7. Repeat all steps until there are no more edges of type $a, c, d$ or $f$.

8. Choose a random edge of type $b$.

9. Apply the second shortest paths idea to the chosen edge/arc.

10. Go back to Step 8 until there are no more edges of type $b$.

11. If we were at all able to apply the second shortest paths idea to make *any* improvements, go back to step 1.

12. If there are any more edges $(i, j)$ of type $a$ left in $G*$, orient it from $i$ to $j$, and add a copy $(j, i)'$ oriented in the opposite direction.

All that remains is to elaborate on exactly what these cost modification procedures are, and what their objective is.

*Cost modification 1*: This procedure tries to force our shortest paths algorithm to traverse links from the matching solution.

1. Given a graph $G_{ij}$, and $G_M$, and a nonpositive number $K$, find all edges $(f, g)$ in $G_{ij}$ that are also in $E_M$, and, (in $G_{ij}$), set the costs $c_{fg} = c_{gf} = K$.

2. Locate in $G_{ij}$, all arcs from $A_M$. If they area of type $f$ (in $G*$), then set the costs $c_{fg} = c_{gf} = K$. If the arc is of type $e$, then set $c_{fg} = 0, c_{gf} = \infty$

*Cost modification 2*: This procedure tries to force our shortest paths algorithm to traverse links that will benefit from our two improvement procedures at the same time as we examine our chosen link, (which may, for instance, get eliminated as part of a shortest 'path' from $i$ to $j$).

1. Given graphs $G_{ij}$, and $G*$, find all edges $(f, g)$ in $G*$ that are of type $a$ or $d$. Also, let $c^*_{fg}$ denote the cost of link $(f, g)$ in the original graph $G$. Then, set the costs $c_{fg}$ and $c_{gf}$ in $G_{ij}$ to be $-c^*_{fg}$ and $-c^*_{gf}$.

2. Locate in $G*$ all links $(f, g)$ of type $c$ or $f$, and set the cost $c_{gf}$ in $G_{ij}$ to $-c^*_{fg}$

3. At whatever point in the process this procedure is being called, set the cost of the selected link in $G_{ij}$ to $\infty$ in both directions, (that is $c_{fg} = c_{gf} = \infty$).

### 3.3.4 Pseudocode for SAPH Concepts

```
   Result: The input graph with modified costs.
   input : An added arc $a_{ij} \in G$
1  $c_{ij} \leftarrow \infty$;
2  Cost modify $G$;
3  Calculate shortest path from $i$ to $j$;
4  if $sp_{ij} < c^{orig}_{ij}$ then
5      Delete a copy of $a_{ij}$ in $G$;
6      Add a copy of $sp_{ij}$ to $G$;
7  end
```

```
   Result: The input graph with modified costs.
   input : A oriented edge $e_{ij} \in G$
1  Cost modify $G$;
2  Calculate two shortest paths from $i$ to $j$, $sp^1_{ij}$
   and $sp^2_{ij}$;
3  if $sp^1_{ij} + sp^2_{ij} < c_{ij}$ then
4      Change the orientation of $e_{ij}$ in $G$;
5      Add a copy of $sp^1_{ij}$ and $sp^2_{ij}$ to $G$;
6  end
```

## 3.4 The Windy Postman Problem

Problem Statement:

$$\text{minimize} \quad \sum_{e^+ \in E^+} c_{e^+} x_{e^+} + \sum_{e^- \in E^-} c_{e^-} x_{e^-}$$

subject to:

$$\sum_{e^+ \in E^+} x_{e^+} - \sum_{e^- \in E^-} x_{e^-} = 0, \forall v \in V \quad (12)$$

$$x_{e^+} + x_{e^-} \geq 1, \forall e \in E \quad (13)$$

$$x_{e^+}, x_{e^-} \in \mathbb{Z}^0_+, \forall e \in E \quad (14)$$

As is to be expected, the formulation of the CPP on a windy graph bears a close resemblance to the formulation of the CPP on an undirected graph. This time, $x_{e^+}$ and $x_{e^-}$ represent the number of times an edge $e$ is traversed in the forward direction, and in the reverse direction respectively. With this in mind, constraint (1) enforces symmetry for each vertex (whenever we enter, we must leave), while constraints (2) and (3) are the usual traversal and integrality requirements.

### 3.4.1 Win's Algorithm (16)

With the Windy Postman Problem, the strategy is a bit different than it has been for the previous three cases. The reason is that, before, we could precisely quantify *a priori* the cost of an augmentation. For instance, if we added edges whose costs summed to 15,

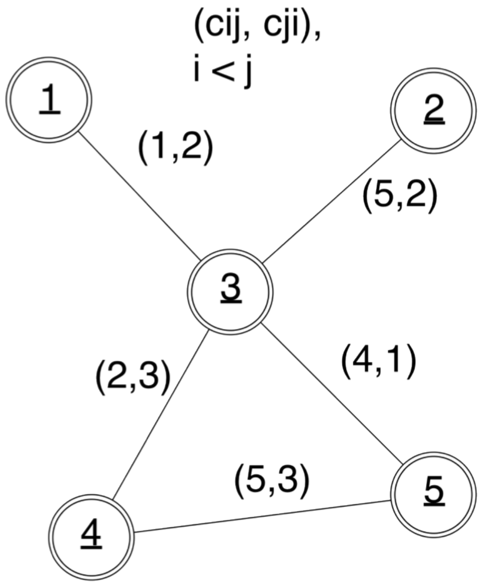


Figure 16: An example of a windy graph, upon which the the Windy Postman Problem may be solved. Notice that each of the edges now has two costs. As a matter of convention here, the first one will be the cost of traversing the edge from $i$ to $j$ where $i < j$, and the second is obviously the cost of traversing it in the opposite direction.

then if we could find an Eulerian augmentation which added edges whose costs summed to 12, it would obviously be preferable. Unfortunately, we don't have that luxury here, since we aren't sure which direction the postman will traverse the edge in his tour, and so the cost of adding an edge is more difficult to assess.

Win's algorithm attempts to address this difficulty in the simplest way possible: it considers average costs. Thus, is solves the UCPP on the graph $G_{\bar{E}}$ with costs $\bar{c}_{ij} = \min\left(sp_{ij} + sp_{ji}\right)$ (Figures 16 and 17). Thus produces an Eulerian augmentation to the original graph. Now, we run a polynomial time algorithm that determines the *optimal* tour on this augmented graph:

1. Given the Eulerian graph $G$, form the digraph $D_G = (V, A)$ where the vertex set is identical to that of $G$, and for each edge in $G$, if $c_{ij} < c_{ji}$, then arc $(i, j)$ is added to $A$. Otherwise, arc $(j, i)$ is added to $A$.
2. Create a second digraph $D' = (V, A')$ by, for each arc $(i, j) \in A$, adding 3 arcs to $A'$: one arc $(i, j)$ with cost $c_{ij}$ and infinite capacity, one arc $(j, i)$ with cost $c_{ji}$ and infinite capacity, and one arc $(j, i)'$ with cost $\frac{c_{ji} - c_{ij}}{2}$ and capacity 2. This last arc is referred to as being *artificial*.
3. Solve a min cost flow problem on $D'$, with demands calculated as they are for the DCPP on $D_G$.
4. Construct an Eulerian digraph $D'' = (V, A'')$ in the following manner. If, in the flow solution, there is 0 flow along the arc $(j, i)'$, then add $1 + x_{ij}$ copies of arc $(i, j)$ to $A''$. Otherwise, add $1 + x_{ji}$ copies of arc $(j, i)$ to $A''$. The Euler cycle on this digraph is an optimal solution to the WPP on $G$.

### 3.4.2 Pseudocode

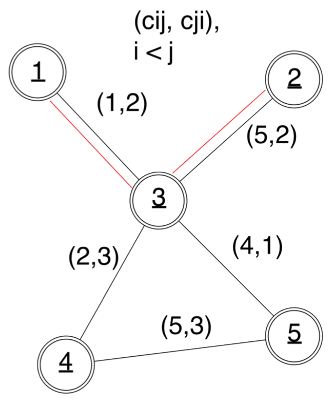


Figure 17: The solution to the WPP on the previous graph. The red arcs are added as part of the flow solution.

```
   **Result**: An approximately optimal windy postman tour over the input graph $G$.
1  $V_{matching} \leftarrow V$;
2  $E_{matching} \leftarrow \emptyset$;
3  **foreach** $v_i \in V_{odd}$ **do**
4      **foreach** $v_j \in V_{odd}$ **do**
5          Add $e_{ij}$ to $E_{matching}$ with $c_{ij}^{matching} = \min(sp_{ij}^{avg}, sp_{ji}^{avg})$;
6      **end**
7  **end**
8  $G_{matching} = (V_{matching}, E_{matching})$;
9  Solve a min-cost matching on $G_{matching}$;
10 Add a copy of each edge in the matching to $G$;
11 $V_{flow} \leftarrow V$;
12 $A_{flow} \leftarrow \emptyset$;
13 **foreach** *Windy edge* $w_{ij} \in E$ **do**
14     Let $c_{ij}^{orig} < c_{ji}^{orig}$;
15     Add $a_{ij}$ to $A_{flow}$ with cost $c_{ij}$;
16     Add $a_{ji}$ to $A_{flow}$ with cost $c_{ji}$;
17     Add an artificial $a_{ji}$ to $A_{flow}$ with cost $\frac{c_{ji} - c_{ij}}{2}$ and flow capacity 2;
18 **end**
19 $G_{flow} = (V_{flow}, A_{flow})$;
20 Solve a min-cost flow on $G_{flow}$;
21 $V_{ans} \leftarrow V$;
22 $A_{ans} \leftarrow \emptyset$;
23 **foreach** *Windy edge* $w_{ij} \in E$ **do**
24     **if** *flow along* $a_{ij} = 0$ **then**
25         Add $x + 1$ copies of $a_{ij}$ to $A_{ans}$;
26     **else**
27         Add $x + 1$ copies of $a_{ji}$ to $A_{ans}$;
28     **end**
29 **end**
30 $G_{ans} = (V_{ans}, A_{ans})$;
31 Return Hierholzers($G_{ans}$);
```

### 3.4.3 Benavent's H1 (1)

This algorithm is essentially an improvement over Win's original algorithm in that it attempts to anticipate the results of the min-cost flow problem solved to obtain the optimal windy tour. In order to accomplish this, edge costs are modified before the matching is solved (to produce an Eulerian undirected graph):

1. Given the original windy graph $G = (V, E)$, calculate the average edge cost for the whole graph ($C_a = \frac{1}{2|E|}\sum_{(i,j)\in E} c_{ij} + c_{ji}$). Now, consider edge set $E_1 = (i, j) \in E : \{|c_{ij} - c_{ji}|\} > K * C_a$. Also, define $E_2 = E \backslash E_1$.

2. Set up a digraph $G^d_R = (V, A')$, where, for each $e \in E$, add 2 arcs in $A'$, $(i, j)$ with cost $c_{ij}$ and infinite capacity, and $(j, i)$ with cost $c_{ji}$ and infinite capacity. Then, for each $e \in E_1$, add an additional artificial arc $(j, i)$ with cost $\frac{c_{ji} - c_{ij}}{2}$ and capacity 2.

3. Solve a min cost flow problem, with demands given by a reduced graph $G' = (V, A)$ which contains an arc $(i, j)$ for each edge $(i, j) \in E_1$, (here we assume $c_{ij} < cji$ so that the arcs in $A$ are in the direction of cheaper traversal).

4. Compile a list L of edges such that:

   - $e \in E_1$ and, in the flow solution, there is positive flow across its corresponding (non-artificial) arcs.

   - $e \in E_2$ and, in the flow solution, there is at least a flow of 2 across its corresponding (non-artificial) arcs.

5. For each edge $e \in L$, set its cost to 0 in the original graph, and then compute the min-cost matching, just as in Win's algorithm. Then, set all costs back to what they were in the original graph, and proceed normally as in Win's algorithm.

### 3.4.4 Pseudocode

```
Result: An approximately optimal windy
        postman tour over the input graph $G$.
1  $C_a \leftarrow$ avg. cost of traversal in $G$;
2  $E_1 \leftarrow \emptyset$;
3  $E_2 \leftarrow \emptyset$;
4  foreach Windy edge $w_{ij} \in E$ do
5      if $|c_{ij} - c_{ji}| > K * C_a$ then
6          Add $w$ to $E_1$;
7      else
8          Add $w$ to $E_2$;
9      end
10 end
11 $A' = \emptyset$;
12 foreach Windy edge $w_{ij} \in E$ do
13     Let $c^{orig}_{ij} < c^{orig}_{ji}$;
14     Add $a_{ij}$ to $A_{flow}$ with cost $c_{ij}$;
15     Add $a_{ji}$ to $A_{flow}$ with cost $c_{ji}$;
16     if $w_{ij} \in E_1$ then
17         Add an artificial $a_{ji}$ to $A_{flow}$ with cost
           $\frac{c_{ji} - c_{ij}}{2}$ and flow capacity 2;
18     end
19 end
20 $G^d_R = (V, A')$;
21 Solve a min-cost flow problem on $G^d_R$;
22 $L \leftarrow \emptyset$;
23 foreach Windy edge $w_{ij} \in E$ do
24     if $w_{ij} \in E_1$ and flow($a_{ij}$) + flow($a_{ji}$) ¿ 0
       then
25         Add $w_{ij}$ to $L$;
26     end
27     if $w_{ij} \in E_2$ and flow($a_{ij}$) + flow($a_{ji}$) ¿ 1
       then
28         Add $w_{ij}$ to $L$;
29     end
30 end
31 foreach Windy edge $w_{ij} \in L$ do
32     Set $c_{ij} = c_{ji} = 0$;
33 end
34 Perform avg. min-cost matching over $G$;
35 Add a copy of each edge included in the
   matching;
36 Reset all costs back to original;
37 Construct the optimal windy tour on $G$;
```

## 3.5 The Directed Rural Postman Problem

Problem Statement:

minimize $\sum_{a \in A} c_a x_a$

subject to:

$$x_a \geq 1, \ \forall a \in A_R \quad (15)$$

$$\sum_{\{a \in A : he_a = i\}} x_a - \sum_{\{a \in A : ta_a = i\}} x_a = 0 \ , \ \forall i \in V \quad (16)$$

$$\sum_{\{a \in A : ta_a \in S \not\ni he_a\}} x_a \geq 1, \forall \emptyset \neq S \subset V, |S| \leq \lfloor \frac{|V|}{2} \rfloor \quad (17)$$

$$x_a \in \mathbb{Z}_+^0 \quad (18)$$

This IP formulation is a bit more tricky: constraints (1), (2), and (4) should look familiar by now; the first enforces traversal of required arcs, the second enforces that our path is indeed a cycle, and the fourth demands integrality. However, constraint (3) requires a bit more elaboration. This is a subtour elimination constraint, that prevents a spurious solutions from consideration. For example, suppose a vehicle must service two streets, one in the west end of town, and one in the east. Then, it is unavoidable that this vehicle must travel the east-west length of town. However, if we did not have constraint (3), it would be considered feasible to have one small cycle in the west part of town, and one in the east, but nothing connecting them. Obviously, this will likely be cheaper than any valid route, but this is clearly not admissible as a candidate circuit.

### 3.5.1 Christofides' Algorithm (5)

Broadly speaking, Christofides' algorithm begins by simplifying the original graph (to discard a lot of the unrequired nodes and arcs), and connecting the required connected components of the graph. It finally solves a min cost flow problem over the remaining graph to obtain a feasible solution to the DRPP.

1. Given the input graph $G = (V, A_R \cup A_{NR})$, define the vertex set $V_R$ to be the set of nodes which have at least one required arc incident on them. Then, consider the graph $G_R = (V_R, A_R)$. We form a modified graph $G' = (V_R, A_R \cup A_S)$ by making it complete, connecting all vertices in $V_R$ with arcs $(i,j)$ that have cost equal to the shortest path in $G$ between node $i$ and node $j$, (these costs are finite because the graph is strongly connected). These added arcs comprise the set $A_S$. Now, remove from $G'$ any arc $(i,j) \in A_S$ that:
   - Has cost $c_{ij} = c_{ik} + c_{kj}$ for some $k \in V_R$.
   - Is a duplicate of an arc in $A_R$.
2. Now, starting with the digraph $G'$, collapse connected required components into nodes, and solve the minimum spanning arborescence problem on this collapsed graph. Add arcs found in this shortest spanning arborescence to a set $T_{t_a}$ to indicate that the SSA was rooted in the connected component $t_a$. Our choice of $t_a$ here is arbitrary.
3. Solve a min cost flow on the graph $G'$ with demands calculated as $out - degree - in - degree$ relative to the arc set $A_R \cup T_{t_a}$, where every arc has infinite capacity. Let $f_{ij}$ be the amount of flow through arc $(i,j)$ in the flow solution. Then, add $f_{ij}$ copies of arc $(i,j)$ to an arc set $F$. The final feasible solution graph is given by $G_S = (V_R, A_R \cup T_{t_a} \cup F)$.

It is worth mentioning an improvement procedures which we implement: when we constructed the shortest spanning arborescence, we fixed a root node, and so we may repeat the algorithm with $k$ different SSA's, where $k$ is the number of required components of the simplified graph $G'$, and choose the best solution. This process is depicted in Figures 18 and 19.

### 3.5.2 Pseudocode

```
Result: An approximately optimal directed
        postman tour over the input graph $G$.
1  $G_R \leftarrow (V, A_R)$ $C = C_1, C_2, ...$ =connected
   components of $G_R$;
2  $V_{Arb} \leftarrow \emptyset$;
3  $A_{Arb} \leftarrow \emptyset$;
4  foreach Component $c_i \in C$ do
5      Add $v_i$ to $V_{Arb}$;
6  end
7  foreach Arc $a_{ij} \in A$ do
8      if $v_i \in C_i$ and $v_j \in C_j$ and $i \neq j$ then
9          Add $a_{ij}$ to $A_{Arb}$;
10     end
11 end
12 $G_{Arb} = (V_{Arb}, A_{Arb})$;
13 Solve a Minimum Spanning Arborescence over
   $G_{Arb}$;
14 foreach Arc $a \in MSA$ do
15     Set $a$ to required in $G$;
16 end
17 Solve a DCPP over $G$ where supplies and
   demands given by $G_R$;
18 Return Hierholzers($G$);
```

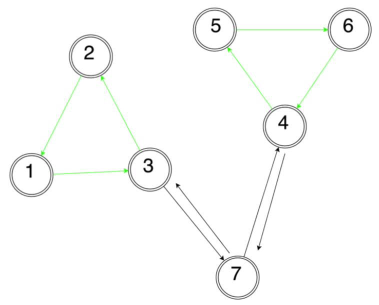


Figure 18: An example of a directed graph, upon which the the Directed Rural Postman Problem may be solved. Green arcs here are required, (i.e. our solution must traverse them at least once) while black arcs are not. Also, link costs are omitted for aesthetic reasons.

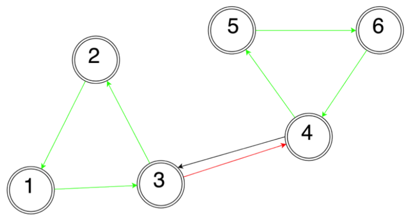


Figure 19: The solution to the DRPP on the previous graph. Notice that node 7 has disappeared because of our simplification of the graph, however it is important to note that $c_{34}$ will be increased to $c_{37} + c_{74}$, and an analogous alteration will be made for $c_{43}$.

### 3.6 The Windy Rural Postman Problem

Problem Statement:

$$\begin{aligned}
\text{minimize} \quad & \sum_{e^+ \in E^+} c_{e^+} x_{e^+} + \sum_{e^- \in E^-} c_{e^-} x_{e^-} \\
\text{subject to:} \quad & \\
& \sum_{e^+ \in E^+} x_{e^+} - \sum_{e^- \in E^-} x_{e^-} = 0, \forall v \in V \quad (19) \\
& x_{e^+} + x_{e^-} \geq 1, \forall e \in E_R \quad (20) \\
& x_{e^+}, x_{e^-} \in \mathbb{Z}^0_+, \forall e \in E \quad (21)
\end{aligned}$$

#### 3.6.1 Benavent's WRPP1 and Benavent's H1(2)

The procedures are identical to their counterparts for solving the WPP, (WRPP1 corresponds to Win's algorithm), except that an initially minimum spanning tree problem must be solved to connect the required components of the graph. The procedure for this is as follows:

1. Compute the connected components $C_1, C_2, C_3, ...$ of the graph $G_R$, which is the graph induced by the required edges in the original problem graph $G$.

2. Construct the graph $G_C$, where the vertex set $V_C$ contains one vertex for each connected component in $G_R$.

3. Complete $G_C$ by adding edges $e_{ij}$ with costs $c_{ij} = \min(c_{avg}(sp_{ij}), c_{avg}(sp_{ji}))$.

4. Solve the minimum spanning tree (MST) problem on $G_C$.

5. If $e_{ij}$ was included in the MST, then set each edge in the shortest path represented by $e_{ij}$ to be 'required.'

#### 3.6.2 Pseudocode

```
   Result: An approximately optimal windy
           postman tour over the input graph $G$.
1  $G_R \leftarrow (V, E_R)$ $C = C_1, C_2, ...$ =connected
   components of $G_R$;
2  $V_{MST} \leftarrow \emptyset$;
3  $E_{MST} \leftarrow \emptyset$;
4  foreach Component $c_i \in C$ do
5      Add $v_i$ to $V_{MST}$;
6  end
7  foreach Windy edge $e_{ij} \in E$ do
8      if $v_i \in C_i$ and $v_j \in C_j$ and $i \neq j$ then
9          Add $e_{ij}$ to $E_{MST}$ with
           $c_{ij} = \min(c_{avg}(sp_{ij}), c_{avg}(sp_{ji})$;
10     end
11 end
12 $G_{MST} = (V_{MST}, A_{MST})$;
13 Solve a Minimum Spanning Tree over $G_{MST}$;
14 foreach Windy edge $e \in MST$ do
15     Set $e$ to required in $G$;
16 end
17 Solve a WPP over $G$ where degree is
   determined in $G_R$;
18 Return Hierholzers($G$);
```

## 4 Implementation

The library is written in Java, which exposes an interface that allows C++ code to be run in the event that performance concerns necessitate that certain portions to be sped up. The code is hosted, (both during development, and upon completion) as a repository on my personal github at `https://github.com/Olibear/ArcRoutingLibrary`.

We choose Java for several reasons. First, it appears as though the general developer industry is moving towards adopting Java as the de facto standard for most modern projects, (e.g. Google has backed Java as the backend for Android, and megascale parallelization technologies like Hadoop are natively compliant). Second, as has already been mentioned, Java provides ways to interface with its main competitor C++, and so concerns over loss of flexibility may be eschewed. Third, libraries in the field of combinatorial optimization (e.g. LEMON, Boost, etc.) have traditionally been written in C++, and so we hope that ours manages to fill the void of Java-based graph libraries.

Git provides convenient means of synchronizing, sharing, and storing code. Github also tracks accesses to the repository so that we may collect meaningful metrics on dissemination of the project.

## 5 Databases

In order to test the performance and accuracy of our library, we ran our solvers on a collection of benchmark instances that have known solutions and are publicly available. For the the directed and undirected Chinese postman problem, we simply generated our own test instances, (since these algorithms are old, exact, and polynomial, test instances aren't prevalent in the literature). Furthermore, for the Directed Rural Postman, we used the test instances used to benchmark DRPP solvers in Campos' computational study(2).

The procedure for generating test instances for the DCPP and UCPP is simply be to randomly generate a graph, (consider all pairs $(i, j)$ and add them to the graph with probability $p_1$, and then connect components of the graph arbitrarily). We generate instances to the DRPP the same way we get instances to the DCPP, and then just pick a subset of required arcs with probability $p_2$.

For the Mixed and Windy Postman problems, we used the test instances made public by Dr. Angel Corberan at his website (`http://www.uv.es/corberan/instancias.htm`) which have documented solutions available at the same place. Additionally, we validated our mixed solvers on the test instances used in Yaoyuenyong's paper where he presents a computational comparison of his own method against Frederickson's (17).

## 6 Validation and Testing

Broadly speaking, there are two types of components to the project that need validation: the subroutines, and the solvers themselves (therefore, the high-level algorithm that makes use of the subroutines). For the former, we compare the output of our implementations to those given in (12). For both the flow algorithms and the shortest path calculations, we compare only the costs, as the specific flow / shortest path solutions are liable to be different if multiple exist with the same objective value, but none of the solvers in which these subroutines are invoked specify any particular type of tie-breaking mechanism as preferable, so we do not worry about any such discrepancies. All we note here, is that, for our own implementations, so long as the ordering of the vertices and arcs is the same, we get identical results on consecutive runs.

For the DCPP and UCPP, we simply ensure that we are reaching the optimal solution, as well as that the run time of the algorithm scales polynomially (just by recording runtime as a function of the problem size of the test instances). We do so by making calls to the Gurobi Java API and setting up Integer Program formulations for each of the problems, and then comparing the optimal objective value to our own.

For the MCPP, we validated our solvers by reproducing the results contained in the table given in Yaoyuenyong's paper (17) which were obtained by running our suite of MCPP solvers on test instances generated by the author. We contacted the author and were able to procure these instances for our own purposes.

For the WPP, we validated our solvers indirectly by validating the more general WRPP solvers (since they are the same, except with an connection step that does not get carried out for a WPP instance). In order to validate the WRPP Solvers, we reproduced the results from Benavent's paper (1) (which presents both the WRPP1 and H1 heuristics) and got an average deviation of 4.24% for WRPP1 and 4.2% for H1. However, given that these results are very close together, we implement three improvement procedures mentioned in the paper, after which the average % deviations drop to 3.45 and 3.04 respectively, similar to the results presented in the paper (which gives a percent deviation before and after applying these procedures).

Similarly for the DRPP, we validated our solver by reproducing results presented in the paper by Campos (2), the test instances for which were graciously provided by the author. Note that here we do not present any solution quality results because no optimal solutions are presented in the paper, so it is not clear what the optimal objective value is for these test instances. We merely use proximity to the tour costs reported in the paper in order to validate.

For testing, in the instances where our validation was performed on large test instances ( $> 1000$ links), we consider the same instances to be our test instances, as they are able to test the computational limits of the algorithms. However, for the MCPP solvers, and WPP solvers, we test on the corresponding Albaida Madrigueras instances (provided on Corberan's website).

## 7 Results

In the following section, we present computational results for the current contents of the library. All tests were performed on a MacBook Air(August 2012), running an i5-3427u processor. Whenever possible, we test on publicly available test instances modeled on real street networks, posted at `http://www.uv.es/corberan/instancias.htm`. Our library contains a parser for the format provided therein which outputs a graph object that is used as input to our solvers. For the UCPP and DCPP, and the subroutines shown here, we have written a graph generator that randomly generates a graph given density, number of vertices, and connectedness (boolean) as inputs.

As mentioned when the details of the algorithm were presented, the Floyd-Warshall All-Pairs Shortest Paths algorithm ought to have an expected asymptotic complexity of $O(n^3)$, and indeed, we can see this borne out by our particular implementation.

Run times for our first attempt at implementing the Successive Shortest Paths Min-Cost Flow algorithm. This algorithm was deemed necessary after a cycle-cancelling algorithm produced run times that were prohibitively high (Figure 22). Performance increased nearly an order of magnitude, even for graphs of relatively small size. Obviously, since the algorithm has super-linear complexity, this improvement is amplified for more complex instances. Still, an analysis of the amount of time spent in each subroutine revealed an algorithmic inefficiency. Namely, an All-Pairs Shortest Paths when calculating the shortest path along which to push flow, when a Single-Source Shortest Paths algorithm would suffice (we are always pushing flow from the source to the sink). Thus, once this correction was made, and Dijkstra's algorithm was substituted, we achieved much better run times, as illustrated in Figure 17.

For the Directed and Undirected Chinese Postman Problems, the majority of the work involved is simply obtaining the solution to the flow / matching problem induced by the original graph, and so for similar problem complexity, the two have comparable performance. It is worth noting that in order to solve the min cost perfect matching problem, we use the publicly available, and extremely efficient C++ implementation of the Blossom algorithm presented in a paper by Kolmogorov in 2009. To call this code from Java, we write a simple function wrapper, and use the Java Native Interface to communicate cross-platform. This explains the seemingly sporadic nature of the UCPP Solver's performance on smaller problem instances, since in these instances it's the overhead of calling the function rather than the function itself that dominates the compute time.

The computational results for Frederickson's MCPP Heuristic are actually quite surprising. Seeing as the heuristic is, at the worst, a 5/3-approximation (meaning we could at worst be 66% away from optimality), one might reasonably expect to see solution

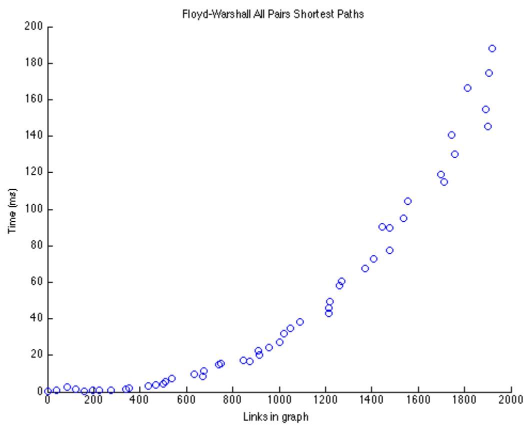


Figure 20: Run times for our implementation of the Floyd-Warshall All-Pairs Shortest Paths subroutine.

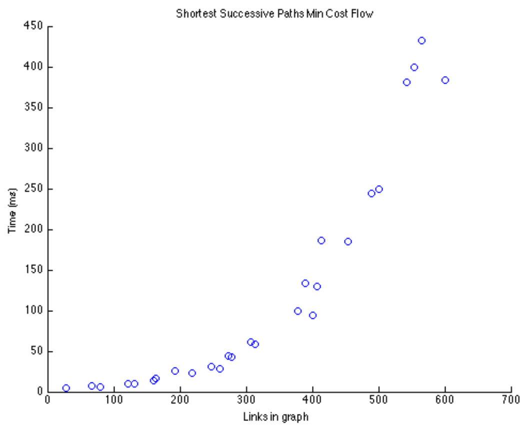


Figure 21: Run times for our implementation of the Successive Shortest Paths Min-Cost Flow subroutine.

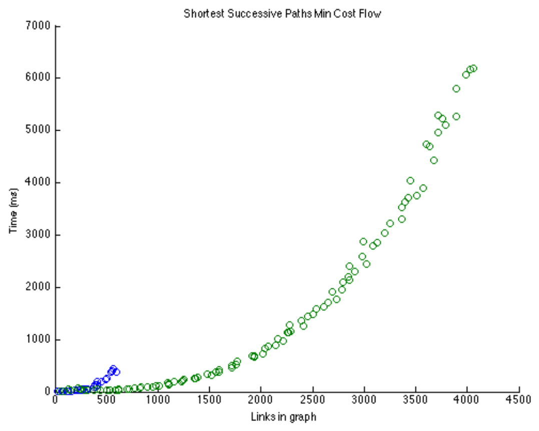


Figure 22: Run times for our implementation of the Successive Shortest Paths Min-Cost FLow subroutine (blue), and again after correcting for the algorithmic inefficiency of using an All-Pairs Shortest Paths algorithm where a single-source one would suffice.

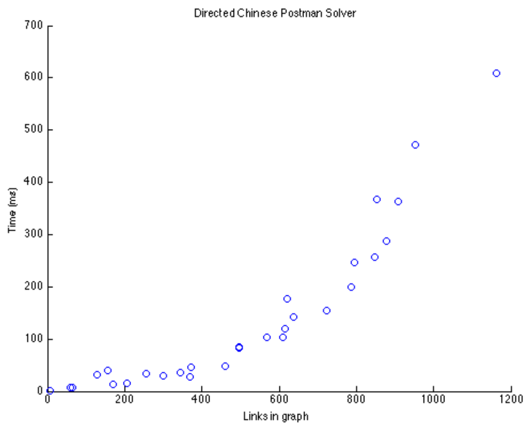


Figure 23: Run times for our implementation of the Directed Chinese Postman Problem Exact Solver.

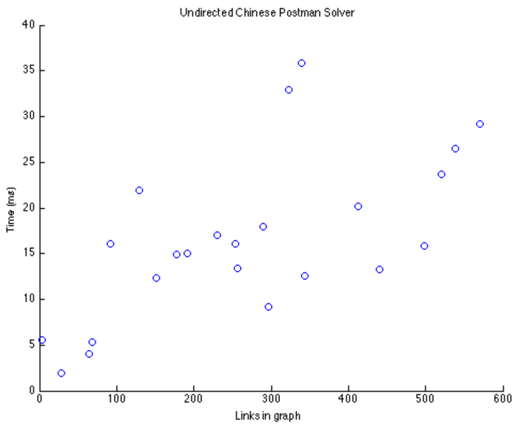


Figure 24: Run times for our implementation of the Undirected Chinese Postman Problem Exact Solver.

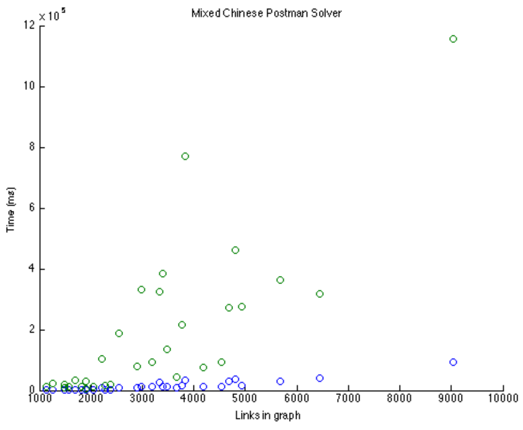


Figure 25: Run times for our implementation of Frederickson's (blue) and Yaoyuenyong's (green) Mixed Chinese Postman Problem heuristics.

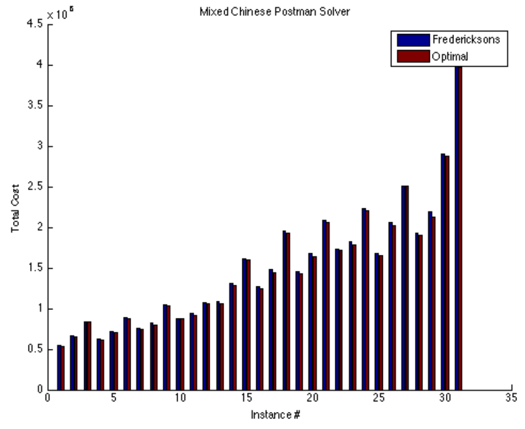


Figure 26: An illustration of the solution quality of Frederickson's (red) MCPP Heuristic compared with the optimal objective value (blue).

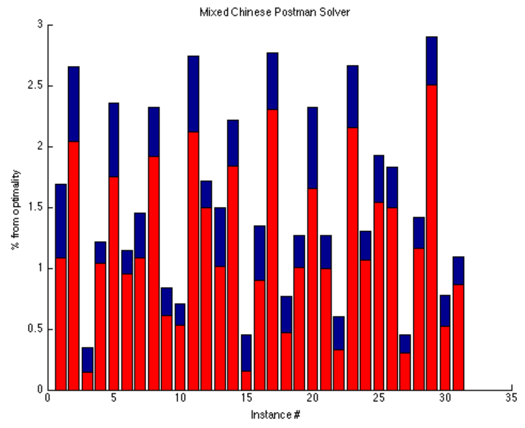


Figure 27: The percent away from optimality that Frederickson's (blue) and Yaoyuenyong's (red) MCPP Heuristics achieves. As a rule of thumb, the instances grow in complexity as a function of their instance number.

quality vary over that range, (and indeed, it is theoretically possible for this to be the case). However, on these instances, modeled after real-world street networks, we see that the solution quality is actually quite good; we never are worse than 3% away from optimality, and are frequently less than 1.5% away (Figure 27). Furthermore, the distance from optimality does not appear to grow in any predictable way with problem size or complexity, which is encouraging in terms of generalizing these results. As expected, we see that Yaoyuenyong's heuristic always does at least as well as Frederickson's heuristic, and almost always better. Unfortunately, the run times are drastically higher, so for instances much larger than the ones we validated on, it may be impractical to use (Figure 25).

The computational results for the two W(R)PP heuristics are more in line with expectation. As we can see, the variation in the solution quality varies a good deal. However, it is worth mentioning that the performance is, on average, still quite close to optimality. Over the 144 test instances for the WRPP, the average % deviation was 3.45 for Win's algorithm, and 3.04 for Benavent's (Figure 29). Again we see a time penalty paid for the improved solution quality (Figure 28), but the growth in time is more commensurate than in the mixed case. Additionally, test instances for rural variants tend to be much smaller in size (i.e. fewer vertices and edges; for example, the 144 WRPP instances had a max of 320 links in the graph, as opposed to the 9000 in the largest wpp instances), but in order to get a more complete idea of the scalability of the code, we ran the rural solvers on the WPP instances for timing purposes. This means that both heuristics are capable of solving instances far larger than they were originally tested on.

## 8 Project Schedule

Although the initial plan was to designate an isolated, consecutive block of time for integration of Gurobi and its associated solvers, this had to be completed earlier for validation of the exact solvers for the DCPP and UCPP. To this end, we have removed this piece from the end of the schedule. In addition, the last month was originally supposed to be used for some work on new research that extends the functionality contained within the library. However, this was an optional component to the project, and was simply delayed until after the completion of the class. With each of these pieces either integrated into the existing steps, or eliminated from the schedule, we were able to implement our heuristic solvers.

- **October:** Complete proposal, begin exact solvers for DCPP, and UCPP, and finalize graph architecture.
- **November:** Complete and validate exact solvers for DCPP, and UCPP.
- **December:** Complete and validate heuristic solvers for MCPP.
- **January:** Complete and validate heuristic solvers for WPP.
- **February - March:** Complete and validate heuristic solvers for DRPP.
- **April:** Modify WPP solvers to deal with rural instances, and validate the new solvers. Add Gurobi solvers for the MCPP and WPP.
- **May:** Final Report

## 9 Deliverables

We have compiled an easily accessible, easily usable, easily extensible library of code designed to solve the aforementioned problems. The mid-year and final reports, as well as documentation (including a readme, and tutorials/examples) and all of the test instances (both generated, and taken from Corberan, Yaoyuenyong, and Campos) are available on the central github page.

## 10 Conclusions and Future Work

Ultimately, we hope that this work will form the basis for many future operations researchers in the field of routing and scheduling. While we fully concede that the particular solver implmentations contained in the library currently aren't necessarily optimal (from a coding standpoint), we have sought to leave the possibility open for contributors who find performance unsatisfactory to extend our abstractions and write their own specific implmentations. As we have proved, the architecture is flexible enough to solve a host of different problems, and will continue to evolve as we encounter complexities that cannot be addressed given the current state of the library.

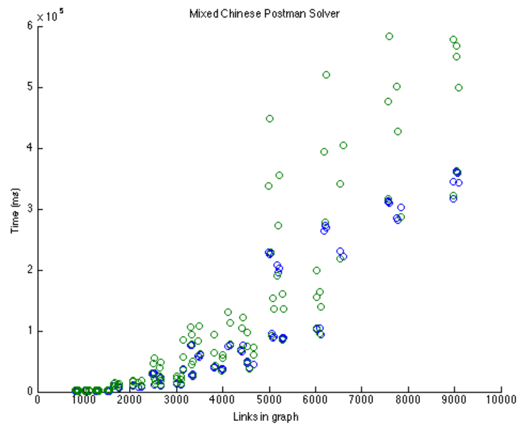


Figure 28: Run times for our implementation of Win's (blue) and Benavent's (green) Windy Postman heuristics.

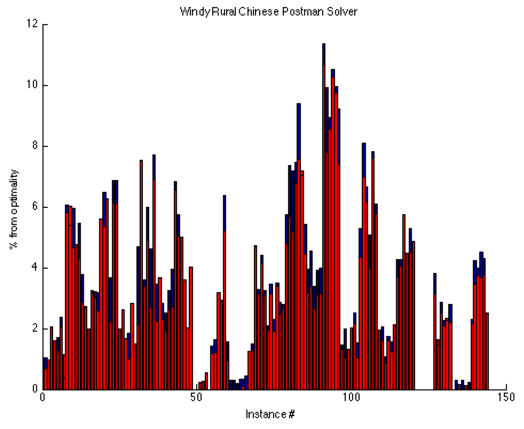


Figure 29: The percent away from optimality that Win's (blue) and Benavent's (red) WRPP Heuristic achieves. As a rule of thumb, the instances grow in size and complexity as a function of their instance number.

Although not necessarily realistic, it would be ideal if this work could form the beginnings of support for the open source movement from the Vehicle Routing community, as the irony of the inherent redundancy required by the current process is decidedly significant.

In terms of future work, we hope to incorporate visualization utilities in order to automate the illustration process, and hopefully provide an elegant way of presenting even large graphs that are traditionally not presented in the literature due their considerable complexity. In order to do so, we plan to leverage other existing technologies, and integrating them into our library (e.g. Gephi).

Furthermore, over the summer, we shall incorporate partitionining code to be able to transition from solving uncapacitated, single-vehicle arc routing variants to capacitated, multi-vehicle problems. Although it is unlikely that these solvers will outperform the state-of-the-art, it will be interesting to see how close we may get with such a rudimentary structural approach.

Finally, we are always looking to expand the utility of the library by incorporating more formats and representations into our IO architecture. One of these, (OSM), is especially promising because it offers an opportunity to generate real-world test instances with relative ease.

## References

[1] Benavent, Enrique, et al. ”New heuristic algorithms for the windy rural postman problem.” Computers & operations research 32.12 (2005): 3111-3128.

[2] Campos, V., and J. V. Savall. ”A computational study of several heuristics for the DRPP.” Computational Optimization and Applications 4.1 (1995): 67-77. (Replace this with Carmine’s paper when I get it).

[3] `http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=minimumCostFlow2`

[4] Edmonds, Jack, and Ellis L. Johnson. ”Matching, Euler tours and the Chinese postman.” Mathematical programming 5.1 (1973): 88-124.

[5] Eiselt, Horst A., Michel Gendreau, and Gilbert Laporte. ”Arc routing problems, part II: The rural postman problem.” Operations Research 43.3 (1995): 399-414.

[6] Derigs, Ulrich. Optimization and operations research. Eolss Publishers Company Limited, 2009.

[7] `http://en.wikipedia.org/wiki/Dijkstra’s_algorithm`

[8] Dussault, Benjamin, et al. ”Plowing with precedence: A variant of the windy postman problem.” Computers & Operations Research (2012).

[9] Frederickson, Greg N. ”Approximation algorithms for some postman problems.” Journal of the ACM (JACM) 26.3 (1979): 538-554.

[10] Grötschel, Martin, and Zaw Win. ”A cutting plane algorithm for the windy postman problem.” Mathematical Programming 55.1-3 (1992): 339-358.

[11] Hierholzer, Carl, and Chr Wiener. ”Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren.” Mathematische Annalen 6.1 (1873): 30-32.

[12] Lau, Hang T. A Java library of graph algorithms and optimization. CRC Press, 2010.

[13] Letchford, Adam N., Gerhard Reinelt, and Dirk Oliver Theis. ”A faster exact separation algorithm for blossom inequalities.” Integer programming and combinatorial optimization. Springer Berlin Heidelberg, 2004. 196-205.

[14] Padberg, Manfred W., and M. Ram Rao. ”Odd minimum cut-sets and b-matchings.” Mathematics of Operations Research 7.1 (1982): 67-80.

[15] Thimbleby, Harold. ”The directed chinese postman problem.” Software: Practice and Experience 33.11 (2003): 1081-1096.

[16] Win, Zaw. "On the windy postman problem on Eulerian graphs." Mathematical Programming 44.1-3 (1989): 97-112.

[17] Yaoyuenyong, Kriangchai, Peerayuth Charnsethikul, and Vira Chankong. "A heuristic algorithm for the mixed Chinese postman problem." Optimization and Engineering 3.2 (2002): 157-187.