

# 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 seek to provide an efficient java 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 graphs). The project is open source, and the code will be 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 transforming the more complicated problem into an instance of an easier problem, (and solving this simpler one 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 seek to create an open source code library 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 (MCP), the CPP on an undirected graph with directionally asymmetric costs (WPP for Windy Postman Problem), and the Rural Postman Problem on a directed 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 shall implement the exact algorithm precisely. For the details of each specific algorithm beyond what follows, consult references [1] for the Directed CPP, [2] for the Undirected CPP, [3] and [4] for the Mixed CPP, [5] for the WPP, and [6] and [7] for the Directed RPP.

### 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 set 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 shall 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  $i$ ). A link is called 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. the cost of going from vertex  $i$  to vertex  $j \neq i$  is not the cost of going from vertex  $j$  to vertex  $i$ ).

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  $(i_0, j_0), (j_0, j_1), (j_1, j_2) \dots (j_{k-1}, j_k)$  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  $V/S$  - number of arcs from  $V/S$  to  $S| \leq$  number of edges from  $S$  to  $V/S$ ). A sufficient condition to be Eulerian for a mixed graph is for every node to have even degree, and in-degree = out-degree.

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, or mixed) 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 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 (Fleury's algorithm).

- *The Rural Postman Problem:* Given a graph  $G = (V, A)$  (in our case, directed), a set of required arcs  $A_R \subseteq A$ , and a set of costs  $c_{ij}$  (associated with traversing the arc  $(i, j)$ ), 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 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 (Fleury's algorithm).

Other common notation includes:

- $\delta(v) = \text{out-degree} - \text{in-degree}$
- $D^+$  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 The Directed Chinese Postman Problem

Problem Statement:

In light of this 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.

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

subject to:

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

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

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

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.

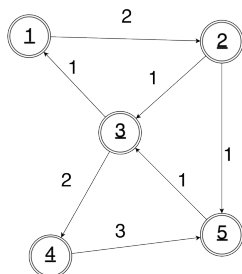


Figure 1: An example of a directed graph, upon which the the Directed Chinese Postman Problem may be solved.

### 2.2.1 An Exact Algorithm for the Directed Chinese Postman Problem

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  $\delta(v) = \text{out-degree} - \text{in-degree}$ ). 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 Fleury's algorithm, which greedily moves from vertex to vertex on the augmented graph, deleting edges once they have been traversed. Put precisely, suppose we are at node  $i$  in our tour. Then, if, in our augmentation, we added any paths from node  $i$  to some other node  $j$ , take this path to node  $j$ , deleting arcs as they are traversed. If we did not add any paths originating at node  $i$ , or if we have already traversed all of them, then we choose any remaining arc, so long as it does not belong to the minimum spanning tree we shall define shortly. Eventually, we must use this arc, but its traversal is saved for last.

All that remains is to compute a minimum spanning tree on the original graph. This is computationally cheap, but we can do even better than the usual greedy algorithms to do so by reusing results from our solution to the shortest paths problem, (which we solve when computing the min cost flow solution). In particular, for any vertex  $v$ , the entries  $\{(u, P_{uv}) : u \neq v\}$  of the paths matrix

define a spanning tree on the original graph; any choice of  $v$  will suffice.

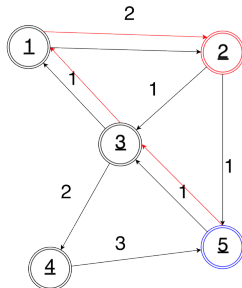


Figure 2: The solution to the DCP on the previous graph. Blue nodes are identified as belonging to  $D^-$ , and red nodes to  $D^+$  in the initial phase of the algorithm. Arcs added as part of the min-cost flow solution are shown in red.

### 2.2.2 The Min Cost Flow Algorithm

In order to obtain a minimum cost solution to the flow problem, we shall implement a cycle cancelling algorithm that operates over the residual graph and eliminates all negative cycles.

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). 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. 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.

### 2.2.3 Shortest Paths Algorithms

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 ideal.

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  $k$ .

### 2.2.4 Pseudocode

---

```
/*
 * DCPD Solver
 *
 * Input: Graph G
 * Output: Optimal Chinese Postman Tour
 *
 */
public Path solve(Graph G)
{
    //form D+, D-
    Set Dplus, Dminus, Dall;
    foreach(Vertex v: V)
    {
        if(v.getDelta() > 0)
        {
```

```

        Dplus.add(v);
    }
    else
    {
        Dminus.add(v);
    }
}

//store set of unbalanced nodes
Dall.addAll(Dplus);
Dall.addAll(Dminus);

//compute shortest paths
int [][] shortestPathsMatrix = shortestPaths(Dall, G);

//solve min cost flow
ArrayList<Path> flowSolution = minCostFlow(shortestPathsMatrix,
    Dplus, Dminus);

//add the flow solution
foreach (Path p: flowSolution)
{
    G.add(p);
}

return Fleurys(G);
}

```

---

### 2.3 The Undirected Chinese Postman Problem

Problem Statement:

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

subject to:

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

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

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).



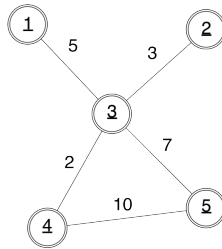


Figure 3: An example of a undirected graph, upon which the the Undirected Chinese Postman Problem may be solved.

### 2.3.1 An Exact Algorithm For The Undirected Chinese Postman Problem

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.

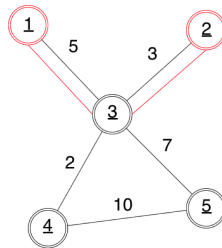


Figure 4: The solution to the UCPP on the previous graph. Red nodes are identified as odd in the initial phase of the algorithm. Edges added as part of the matching solution are shown in red.

### 2.3.2 Pseudocode

---

```

/*
 * UCPP Solver
 *
 * Input: Graph G

```

```

* Output: Optimal Chinese Postman Tour
*
*/
public Path solve(Graph G)
{
    //form odd
    Set odd;
    foreach(Vertex v: V)
    {
        if(v.getDelta() % 2 == 1)
        {
            odd.add(v);
        }
    }

    //compute shortest paths
    int[][] shortestPathsMatrix = shortestPaths(odd, G);

    //solve min cost matching
    ArrayList<Path> matchSolution =
        minCostMatching(shortestPathsMatrix, odd);

    //add the flow solution
    foreach (Path p: flowSolution)
    {
        G.add(p);
    }

    return Fleurys(G);
}

```

---

## 2.4 The Mixed Chinese Postman Problem

Problem Statement:

$$\begin{aligned}
& \text{minimize} && \sum_{s \in \{A \cup \hat{E} \cup \check{E}\}} c_s x_s \\
& \text{subject to:} && \\
& && y'_e + y'_{\check{e}} \geq 1, && \forall e \in E && (6) \\
& && x_s = y'_s + y_s, \forall s \in A \cup \hat{E} \cup \check{E} && (7) \\
& && \sum_{s \in S_v^+} x_s - \sum_{s \in S_v^-} x_s = 0, && \forall v \in V && (8) \\
& && y'_a = 1, && \forall a \in A && (9) \\
& && y'_e \in \{0, 1\}, && \forall e \in \hat{E} \cup \check{E} && (10) \\
& && y_s \in \mathbb{Z}_+^0 && (11)
\end{aligned}$$

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  $\check{E}$  contains edges  $\check{e}$  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  $y_s$ .

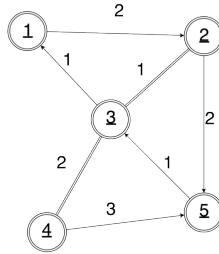


Figure 5: An example of a mixed graph, upon which the the Mixed Chinese Postman Problem may be solved.

#### 2.4.1 Even-Symmetric-Even

The first heuristic we plan to implement has the same intuitive motivation as the exact algorithms for the DCP 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  $V/S$  must be greater than or equal to the difference between the number of arcs from  $V$  to  $V/S$  and the number of arcs from  $V/S$  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 they consist 'mostly' of links that were added in the previous two phases).

1. *Phase I, Even*: Solve the UCPP on the original graph, treating all arcs as edges. This will produce an augmented graph  $G^E$ .
2. *Phase II, Symmetric*: Solve the DCP on  $G^E$ , treating each edge  $(u, v)$  as two arcs  $(u, v)$  and  $(v, u)$ . If the solution to the DCP 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.

## 2.4.2 Pseudocode

---

```

/*
 * MCPP Solver
 *
 * Input: Graph G

```

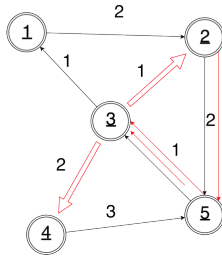


Figure 6: The solution to the MCP on the previous graph. The red arc (2,5) is added in the initial Even phase, while all other red links are added in the Symmetric phase. The thicker red arrows are to indicate that the edges in the original graph were oriented in the corresponding direction.

```

* Output: Optimal Chinese Postman Tour
*
*/
public Path solve(Graph G)
{
    //Phase I: Even

    //Solve the UCPP on G, ignoring link direction
    Graph G_alt = G;

    //make all edges undirected
    foreach(Edge e : E_alt)
    {
        e.setType(Undirected);
    }
    ArrayList<Path> ucppAugmentation = UCPPAugment(G_alt);

    //add the ucpp augmentation
    foreach (Path p : ucppAugmentation)
    {
        G.add(p);
    }

    //Phase II: Symmetric

    //Solve the DCP on a modified digraph
    Graph G_E = G;

    //modify the graph to replace edges with two arcs
    foreach(Edge e : E_E)
    {
        if(e.getType() == EDGE.UNDIRECTED)
        {
            G_E.remove(e);
        }
    }
}

```

```

        G_E.add(new Arc(e.getFirst(), e.getSecond(),
            e.getCost()));
        G_E.add(new Arc(e.getSecond(), e.getFirst(),
            e.getCost()));
    }
}

//Solve the DCP on the updated G, breaking edges up into two
//arcs
ArrayList<Path> dcppAugmentation = DCPAugment(G_E);

//add the dcpp augmentation, and keep track of whether an edge
//gets oriented
foreach (Path p: dcppAugmentation)
{
    foreach (Arc e: p)
    {
        if(e.isArcIn(G))
            G.add(e);
        else if(G.get(e).getType() == EDGE.ORIENTED)
        {
            G.get(e).setType(EDGE.UNORIENTED_PERM);
            G.add(e);
        }
        else if (G.get(e).getType() == EDGE.UNORIENTED)
        {
            G.get(e).setType(EDGE.ORIENTED);
        }
    }
}

//Phase III: Even
Set odd;

foreach (Vertex v : V)
{
    if(v.getDegree() % 2 == 1)
    {
        odd.add(v);
    }
}

while(!odd.isEmpty())
{
    Vertex vs = odd.getFirst();
    Vertex v = vs;
    while (odd.contains(v))
    {
        odd.remove(v);
    }
}

```

```

do
{
    //here, added edges is the set of arcs
    added in Phase II
    addedEdges.remove(<v,w>)
    if (<v,w>.getHead() == w )
    {
        G.add(<v,w>);
    }
    else
        G.remove(<w,v>);
    v = w;
} while (odd.contains(v))
odd.remove(v);
do
{
    //here, unoriented is the set of still
    unoriented edges
    unoriented.remove((v,w));
    G.add(v,w);
    v = w;
} while (odd.contains(v) || v == vs)
}

return Fleurys(G);
}

```

---

### 2.4.3 Shortest Additional Path

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.

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  $ShortestPath(A \rightarrow B)$ ,  $(B \rightarrow A)$ ,  $ShortestPath2(A \rightarrow B)$ . Although this second case may seem like a bizarre one to investigate (since the shortest path costs

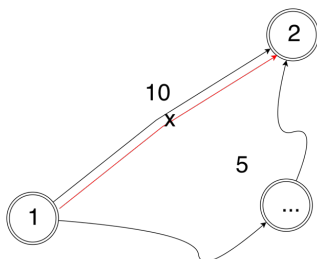


Figure 7: An example of the first SAPH idea, where we replace an added link (red) with a cheaper shortest path.

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.

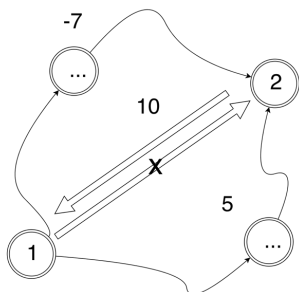


Figure 8: An example of the second SAPH idea, where we reverse the orientation of an edge and add two 'paths' from node  $i$  to  $j$  which sum to a negative cost.

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



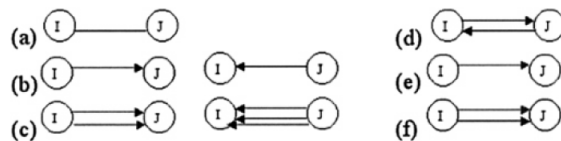


Figure 9: Taken from (4). Edges and arcs in  $G$  must end up in one of the following configurations in  $G^*$ :

- 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  $f$

6. Apply the first shortest paths idea to the chosen edge/arc.
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 are 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  $c_{fg}$  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  $c_{fg} = c_{gf} = \infty$ ).

#### 2.4.4 Pseudocode for SAPH Concepts

---

```

/*
 * SAPH Concept 1
 *
 * Input: Graph G1, G2, G_star, link L
 * Output: A modified G_star after applying SAPH Concept 1
 *
 */
public Graph SAPH1(Graph G1, Graph G2, Link L, Graph G_Star)
{
    Graph G_modified = G_Star;
    int tempCost = L.getCost();
    L.setCost(INFINITY);
    Vertex v1 = L.getTail();
    Vertex v2 = L.getHead();
    Path[] [] shortestPathsSolution = shortestPaths(G1);
    L.setCost(tempCost); //reset the cost

    int cost_ij = 0; //in G2
    int cost_ji = 0;
    foreach(Link L: shortestPathsSolution[i][j])
    {
        cost_ij += G2.getCost(L);
    }
    foreach(Link L: shortestPathsSolution[j][i])
    {
        cost_ji += G2.getCost(L);
    }

    if(cost_ij < tempCost && cost_ij < cost_ji)
    {
        G_modified.add(shortestPathsSolution[i][j]);
    }
    else if (cost_ji < tempCost && cost_ij > cost_ji)
    {
        G_modified.add(shortestPathsSolution[j][i]);
    }
}

```

```

        eliminateCycles(G_modified);

        return G_modified;
    }

    /*
    * SAPH Concept 2
    *
    * Input: Graph G, G_star, Edge e
    * Output: A modified G_star after applying SAPH Concept 2
    *
    */
    public Graph SAPH2(Graph G, Edge e, Graph G_Star)
    {
        Graph G_modified = G_Star;

        Graph G_temp = G_Star;
        Graph G3 = new Graph(G);
        Graph G4 = new Graph(G);

        //cheapest path
        costModify2(G3, G_Star); //the second cost modification
        described later
        Path[] [] shortestPathsSolution3 = shortestPaths(G3);
        G_temp.add(shortestPathsSolution3[i][j]);

        //second cheapest path
        costModify2(G4, G_modified);
        Path[] [] shortestPathsSolution4 = shortestPaths(G4);

        //if sum of costs is negative, add them to the graph, and remove
        the original added edge
        if (shortestPathsSolution3[i][j].getCost() +
            shortestPathsSolution4[i][j].getCost() < 0)
        {
            G_modified.add(shortestPathsSolution3[i][j]);
            G_modified.add(shortestPathsSolution4[i][j]);
            e.reverseOrientation();
        }

        eliminateCycles(G_modified);

        return G_modified;
    }
}

```

---

## 2.5 The Windy Postman Problem

Problem Statement:

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

subject to:

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

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

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

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.

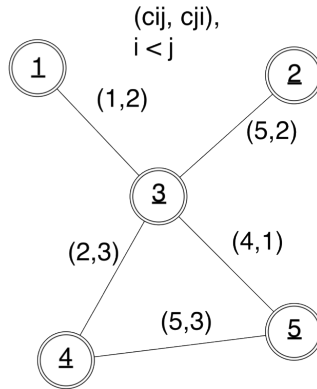


Figure 10: An example of a windy graph, upon which the the Windy Postman Problem may be solved. Notice that each of the edges now has two costs. As a matter of convention here, the first one will be the cost of traversing the edge from  $i$  to  $j$  where  $i < j$ , and the second is obviously the cost of traversing it in the opposite direction.

### 2.5.1 Win's Algorithm

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, 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, it solves the UCPP on the graph  $G_{\bar{E}}$  which is identical to the input graph  $G$  except that it is undirected, with costs  $\bar{c}_{ij} = \frac{c_{ij} + c_{ji}}{2}$ . 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 DCP 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  $G$ .

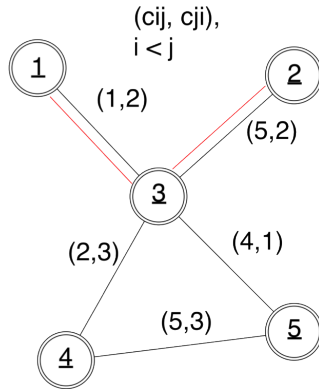


Figure 11: The solution to the WPP on the previous graph. The red arcs are added as part of the flow solution.

## 2.5.2 Pseudocode

---

```
/*
 * WPP Solver
 *
 * Input: Graph G
 * Output: Approximate solution (path over the G) to the WPP.
 *
 */
public Path solve(Graph G)
{
    //solve the UCPP on a modified graph that has average costs.
    Set E_Undir = new Set();
    Graph G_Undir = (V, E_Undir);
    //calculate average costs
    foreach (Edge e : E)
    {
        E_Undir.add(new Edge(e.getFirst(), e.getSecond(), (c_ij +
            c_ji / 2)));
    }

    //form odd
    Set odd;
    foreach(Vertex v: V)
    {
        if(v.getDelta() % 2 == 1)
        {
            odd.add(v);
        }
    }

    //compute shortest paths
    int[][] shortestPathsMatrix = shortestPaths(odd, G);

    //solve min cost matching
    ArrayList<Path> matchSolution =
        minCostMatching(shortestPathsMatrix, odd);

    //add the flow solution
    foreach (Path p: flowSolution)
    {
        G.add(p);
    }

    //Now run Win's Algorithm to find the optimal WP Cycle on the
        Eulerian Graph produced
    //by our solution to the UCPP
    Set A = new Set();
    Set A_prime = new Set();
}
```

```

Set A_dprime = new Set();
Graph D_G = (V, A);
Graph D_prime = (V, A_prime);

//Add an arc in the direction of lesser cost for each edge in E
foreach (Vertex e : E)
{
    if (c_ij < c_ji)
    {
        A.add(new Arc(i,j,e.getCost()));
        //artificial arc in the high cost direction
        A_prime.add(new Arc(j,i,(c_ji - c_ij) /
            2).setFlow(2))
    }
    else
    {
        A.add(new Arc(j,i,e.getCost()));
        //artificla arc in the high cost direction
        A_prime.add(new Arc(i,j,(c_ij - c_ji) /
            2).setFlow(2))
    }
    A_prime.add(new Arc(i,j,e.getCost()));
    A_prime.add(new Arc(j,i,e.getCost()));
}

ArrayList<Path> minCostSolution = minCostFlow(D_prime);

int[] [] flowSolution = new int[] []; //entry is amount of flow
    pushed through the arc i,j in the flow solution
int[] [] usedArtificial = new int[] []; //entry is 1 if we used
    the artificial arc in the flow solution, 0 oth.
foreach(Path p: minCostSolution)
{
    foreach(Arc a: p)
    {
        flowSolution[a.getTail()][a.getHead()] += 1;
        //increment the guy in the flow solution matrix
        if (a.isArtificial())
            usedArtificial[a.getTail()][a.getHead()] =
                1;
    }
}

for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        //If the artificial guy appears in the flow
        solution, then add 1 + x copies of the
        opposite direction arc
    }
}

```

```

        if(usedArtificial[i][j] == 1)
        {
            for(k=0;k<=flowSolution[j][i];k++)
            {
                A_dprime.add(new Arc(j,i,c_ji));
            }
        }
        else
        {
            for(k=0;k<=flowSolution[i][j];k++)
            {
                A_dprime.add(new Arc(i,j,c_ij));
            }
        }
    }
}

Graph G_final = new Graph(V,A_dprime);
return Fleurys(G_final);
}

```

---

### 2.5.3 Benavent's H1

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 earlier in the process. 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  $E_2 = E \setminus E_1$ .
2. Set up a digraph  $G_R^d = (V, A')$ , 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  $c_{ij} < c_{ji}$  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.

#### 2.5.4 Pseudocode

---

```

/*
 * WPP Solver2
 * H1
 *
 * Input: Graph G
 * Output: Approximate solution (path over the G) to the WPP.
 *
 */
public Path solve(Graph G)
{
    //compute average traversal cost
    int sumCost;
    foreach(Edge e: E)
    {
        sumCost += c_ij + c_ji;
    }
    double averageCost = sumCost / (2*E.getSize());
    double cutOff = .2*averageCost;

    //figure out the high disparity edges, and care for them first.
    Set E_1, E_2;
    foreach(Edge e: E)
    {
        if (abs(c_ji - c_ij) > cutOff)
        {
            E_1.add(e);
        }
        else
            E_2.add(e);
    }

    Set A_prime;
    Set A; //holds only arcs in E_1 (cheaper direction of the edge)
    Graph G_Rd = (V, A_prime);

    foreach(Edge e: E)
    {
        A_prime.add(new Arc(i,j,c_ij));
        A_prime.add(new Arc(j,i,c_ji));
    }
}

```

```

        if(E_1.contains(e))
        {
            if(c_ij < c_ji)
            {
                A_prime.add(new Arc(i,j,c_ij).setFlow(2));
                A.add(new Arc(i,j,c_ij));
            }
            else
            {
                A_prime.add(new Arc(j,i,c_ji).setFlow(2));
                A.add(new Arc(j,i,c_ji));
            }
        }
    }

    //use demands from reduced graph
    Graph G_E_1 = (V,A);
    G_Rd.setDemands(G_E_1);
    Set L;

    ArrayList<Path> minCostFlowSolution = minCostFlow(G_Rd);
    foreach(Path p: minCostFlowSolution)
    {
        foreach(Edge e: p)
        {
            if (E_1.contains(e) &&
                minCostFlowSolution.contains(a_ij))
                L.add(e);
            else if(E_2.contains(e) &&
                minCostFlowSolution.contains(a_ji))
                L.add(e);
        }
    }

    //solve a min cost flow on the reduced cost graph (produced by
    //solution to flow problem)
    Set E_ReducedCost = E;
    Graph G_ReducedCost = (V,E_ReducedCost);
    foreach(Edge e: E_ReducedCost)
    {
        if(L.contains(e))
            e.setCost(0);
    }

    ArrayList<Path> minCostMatchingSolution =
        minCostMatching(G_ReducedCost);
    foreach(Path p: minCostMatchingSolution)
    {
        G.add(p);
    }

```

```

    return WPPSolver(G);
}

```

---

## 2.6 The Directed Rural Postman Problem

Problem Statement:

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

subject to:

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

$$\sum_{\{a \in A: h e_a = i\}} x_a - \sum_{\{a \in A: t a_a = i\}} x_a = 0, \forall i \in V \quad (16)$$

$$\sum_{\{a \in A: t a_a \in S \nexists h e_a\}} x_a \geq 1, \forall \emptyset \neq S \subset V, |S| \leq \lfloor \frac{|V|}{2} \rfloor \quad (17)$$

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

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.

### 2.6.1 Christofides' Algorithm

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

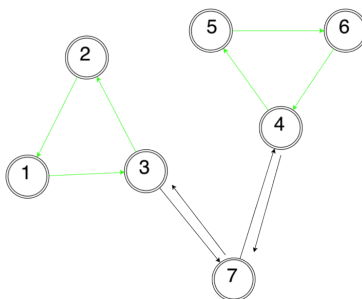


Figure 12: An example of a directed graph, upon which the the Directed Rural Postman Problem may be solved. Green arcs here are required, (i.e. our solution must traverse them at least once) while black arcs are not. Also, link costs are omitted for aesthetic reasons.

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 two improvement procedures which we shall attempt to implement: first, 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; second, once we have constructed our final augmented graph  $G_S$ , it's possible that replacing two arcs  $(i, j)$  and  $(j, l)$  with the single arc  $(i, l)$  (if this is a valid arc) would leave  $G_S$  still strongly connected, and so replacing the two arcs with this one would leave us feasible and at least as good from a cost perspective.

## 2.6.2 Pseudocode

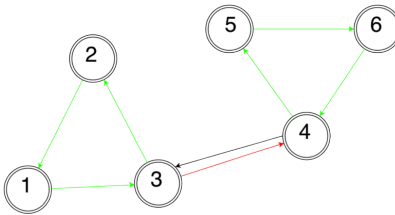


Figure 13: The solution to the DRPP on the previous graph. Notice that node 7 has disappeared because of our simplification of the graph, however it is important to note that  $c_{34}$  will be increased to  $c_{37} + c_{74}$ , and an analogous alteration will be made for  $c_{43}$ .

---

```

/*
 * DRPP
 * Christofides
 *
 * Input: Graph G
 * Output: Approximate solution (path over the G) to the WPP.
 *
 */
public Path solve(Graph G)
{
    //Set up the required graph

    //the required items
    Set A_R, V_R;
    Graph G_R = (V_R,A_R);
    foreach(Arc a: A)
    {
        if(a.isRequired())
        {
            A_R.add(a);
            V_R.add(a.getTail());
            V_R.add(a.getHead());
        }
    }

    //the required strongly connected components
    Set<Set<Vertex>> stronglyConnectedComponents =
        G_R.getStronglyConnectedComponents();

    //make it complete
    Path[] [] shortestPaths = computeShortestPaths(G,V_R);
    for(int i=0;i<V_R.size();i++)
    {
        for(int j=0;j<=i;j++)

```

```

    {
        boolean skip = false;
        //don't add a path if it represents a redundancy
        cost = shortestPaths[i][j].getCost();
        foreach(Vertex v_k: V_R)
        {
            if(cost == shortestPaths[i][k].getCost() +
                shortestPaths[k][j].getCost())
            {
                skip = true;
                break;
            }
        }
        if(skip)
            continue;
        //don't add a copy of a required arc
        if(shortestPaths[i][j] == a_ij &&
            a_ij.isRequired())
            continue;
        G_R.add(shortestPaths[i][j]);
    }

    //form a graph on which we solve the minimum spanning
    //arborescence on
    Set V_Collapsed, A_Collapsed;
    Graph G_R_Collapsed = G_R.collapse(stronglyConnectedComponents);
    Set<Arc> minimumSpanningArborescence = MSA(G_R_Collapsed);

    //arc set A_R U minSpanningArborescence
    Set A_demands = A_R;
    A_demands.addAll(minimumSpanningArborescence);
    G_R.setDemands(A_demands);

    ArrayList<Path> minCostFlowSolution = minCostFlow(G_R);
    foreach(Path p: minCostFlowSolution)
    {
        G_R.add(p);
    }

    return Fleurys(G_final);
}

```

---

### 2.6.3 Benavent's WRPP1

Although originally created to deal with the more general Windy Rural Postman Problem, we may model the DRPP as an instance of the WRPP by assigning infinite cost to traversing edges in the opposite direction of their corresponding

arc. Once we do this, the procedure is identical to Win's algorithm for the Windy Postman Problem, except that here, before we begin, we compute the connected components of the graph induced by the required arcs  $G_R$ , and connect them with a shortest spanning arborescence. The arcs that are associated with this SSA are added to the set of required arcs, forming a new required graph  $G'_R$  on which we run Win's WPP algorithm.

#### 2.6.4 Pseudocode

---

```

/*
 * DRPP
 * Benavent
 *
 * Input: Graph G
 * Output: Approximate solution (path over the G) to the WPP.
 *
 */
public Path solve(Graph G)
{
    private static final INFINITY = INTEGER.MAX;
        //Set up the required graph

        //the required items
        Set A_R, V_R;
        Graph G_R = (V_R,A_R);
        foreach(Arc a: A)
        {
            if(a.isRequired())
            {
                A_R.add(a);
                V_R.add(a.getTail());
                V_R.add(a.getHead());
            }
        }

        //the required strongly connected components
        Set<Set<Vertex>> stronglyConnectedComponents =
            G_R.getStronglyConnectedComponents();

        //form a graph on which we solve the minimum spanning
            arborescence on
        Set V_Collapsed, A_Collapsed;
        Graph G_R_Collapsed = G_R.collapse(stronglyConnectedComponents);
        Set<Arc> minimumSpanningArborescence = MSA(G_R_Collapsed);

        //arc set A_R U minSpanningArborescence
        Set A_demands = A_R;
        A_demands.addAll(minimumSpanningArborescence);

```

```

G_R.setDemands(A_demands);

ArrayList<Path> minCostFlowSolution = minCostFlow(G_R);
foreach(Path p: minCostFlowSolution)
{
    A_R.add(p);
}

Set E_Windy;
Graph G_W = (V_R, E_Windy);
//Now set up the Windy Graph
foreach(Arc a: A_R)
{
    E_Windy.add(new
        Edge(a.getTail(),a.getHead(),a.getCost(),INFINITY));
}

return WPP2Solve(G_W);
}

```

---

## 2.7 Remarks

Furthermore, since we fully intend for this library to be used in circumstances beyond the context of our own endeavors, if time permits, we shall attempt to include several useful utilities. These include, but are not necessarily limited to the following: the capability to use existing software to visualize graphs created using our library (currently Gephi seems to be the 3rd party software of choice); the ability to easily exchange problem instances by allowing export to the prevailing industry standard graphml format; and the option to make calls to CPLEX and Gurobi in solvers.

Ultimately, upon completion of this code base, we seek to extend the work done by Benjamin Dussault et al. on the variant of the Windy Postman Problem with precedence discussed in [8]. In this work, an additional complication to the standard WPP is thrown in: an edge has reduced cost upon repeat traversal (of course, the costs are still asymmetric). This is meant to reflect a scenario where some impediment to progress is removed during the first traversal, and so any further traversal takes less resources. The paper presents a method of producing a good initial feasible solution, and then details an improvement procedure that involves permuting cycles found in the original solution. However, these permutations are essentially arrived at randomly. We hope to devise a more systematic way of choosing which permutations tend to yield better solutions, and use this knowledge to refine the local search procedure.



### 3 Implementation

The library will originally be written in Java, which exposes an interface that allows C++ code to be run in the event that performance concerns necessitate certain portions to be sped up. The code will be 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. 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 will help 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.

### 4 Databases

In order to test the performance and accuracy of our library, we shall run 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 shall simply generate our own test instances, (since these algorithms are old, exact, and polynomial, test instances aren't prevalent in the literature). Furthermore, we shall do the same for the Directed Rural Postman, since there do not seem to be public instances available for this particular variant.

The procedure for generating test instances for the DCP and UCP will 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 DCP, and then just pick a subset of required arcs with probability  $p_2$ . If time permits, we shall investigate the effect that these have on performance.

For the Mixed and Windy Postman, we shall use 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.

### 5 Validation

For the DCP and UCP, 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). For each of the NP-Hard problems we shall attempt to solve, we attempt to validate our efforts by performing Wilcoxon's Signed Rank Test. Wilcoxon's

Number Of Vertices	% from Opt. (Paper)	% From Opt. (Us)
100	2.4	2.7
200	6.3	5.7
300	7.9	8.6
400	9.7	8.9
500	12.5	15

Table 1: An Example of Using the Wilcoxon Signed-Rank Test: The largest difference between observations would be considered first, (here, the largest difference is between 12.5 and 15). It is the first in the list, and so it has weight 1. The next greatest difference is between 6.3 and 5.7, and so this receives a weight of -2, etc. The test statistic is then the weighted sum of these differences (i.e.  $W = 1 - 2 + 3 - 4 + 5 = 3$ ). For  $N > 10$ , there is a way of computing a z-score directly from this  $W$ , but here, since we only have 3 pairs of data, we may simply use a reference table with a standard  $\alpha = .05$ , and conclude that since  $W > W_{crit} = 1$ , then these are unacceptable results.

Signed Rank Test proceeds by considering both the distance between paired observations, and the sign of the distance. The null hypothesis will be that our results are identical to the results provided in the papers in which the heuristics are proposed. Although our test instances will be different from the instances used in the paper, a common metric in the literature is to consider *percent from optimality* which is exactly the metric we shall use in order to determine the test statistic.

## 6 Testing

In addition to running our heuristics on several more complex benchmark instances taken from our data sources, we shall run our solvers on several simple, small-sized instances that are constructed to be solvable to optimality by inspection, as well as several non-trivial instances that we solve in CPLEX and Gurobi to optimality. In this way, we can ensure that the algorithm is executing in the way that we expect it to, while also testing scalability on more complex problems.

## 7 Project Schedule

- **September:** Complete proposal, begin exact solvers for DCP, and UCP, and finalize graph architecture.
- **October:** Complete and validate exact solvers for DCP, and UCP.

- **November:** Complete and validate heuristic solvers for MCPP, begin CPLEX & Gurobi support.
- **December:** Complete and validate heuristic solvers for WPP, complete CPLEX & Gurobi support.
- **January:** Complete and validate heuristic solvers for DRPP.
- **February:** Investigate, and implement cycle search operators for Plowing with Precedence.
- **March:** Performance optimization
- **April – May:** Visualization, I/O utilities to allow for information exchange (Gephi, Graphviz potential candidates)

## 8 Milestones

At the conclusion of each month, solvers that ought to be completed should have test cases ready to benchmark performance, and verify the validity of the solutions produced by the library. For instance, by the end of October, (assuming all goes according to plan), we hope to be able to set up a test instance, run our DCP or UCPP solver on it, and collect metrics like run time, solution feasibility, etc.

## 9 Deliverables

By the conclusion of the year, we hope to have compiled an easily accessible, easily usable, easily extensible library of code designed to solve the aforementioned problems. In addition, if time allows, we also hope to be able to integrate benchmarking and visualization utilities into the library (as opposed to simply cobbled together for our own use). 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) will be available on the central github page.

## References

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

- [2] [http://www.ise.ncsu.edu/fangroup/or766.dir/or766\\_ch9.pdf](http://www.ise.ncsu.edu/fangroup/or766.dir/or766_ch9.pdf)
- [3] Eiselt, Horst A., Michel Gendreau, and Gilbert Laporte. "Arc routing problems, part I: The Chinese postman problem." *Operations Research* 43.2 (1995): 231-242.
- [4] Yaoyuenyong, Kriangchai, Peerayuth Charnsethikul, and Vira Chankong. "A heuristic algorithm for the mixed Chinese postman problem." *Optimization and Engineering* 3.2 (2002): 157-187.
- [5] Benavent, Enrique, et al. "New heuristic algorithms for the windy rural postman problem." *Computers & operations research* 32.12 (2005): 3111-3128.
- [6] Eiselt, Horst A., Michel Gendreau, and Gilbert Laporte. "Arc routing problems, part II: The rural postman problem." *Operations Research* 43.3 (1995): 399-414.
- [7] 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).
- [8] Dussault, Benjamin, et al. "Plowing with precedence: A variant of the windy postman problem." *Computers & Operations Research* (2012).