Comput. & Ops. Res. Vol. 10, No. 1, pp. 47–59, 1983
Printed in Great Britain.

0305-0548/83/010061-02$03.00/0
Pergamon Press Ltd.

COMPUTATIONAL EXPERIMENTS WITH ALGORITHMS FOR A CLASS OF ROUTING PROBLEMS

B. L. GOLDEN†
College of Business and Management, University of Maryland, College Park, MD 20742, U.S.A.

J. S. DEARMON‡
Mitre Corporation, McLean, Virginia, U.S.A.

and

E. K. BAKER§
Department of Management Science, School of Business Administration, University of Miami, FL 33146, U.S.A.

Scope and Purpose—The problem of traversing each arc in a network at least once and returning to the starting point while minimizing total distance traveled is the well-known Chinese postman problem. In this paper, we address a capacitated version of this problem which has many applications. Several heuristic solution procedures are proposed, tested, and analyzed, and the results are discussed at length.

Abstract—The vehicle routing problem, a generalization of the infamous traveling salesman problem, is a well-known distribution management problem that has been the focus of much research attention. On the other hand, generalizations of arc routing problems, such as the Chinese postman problem, have been comparatively neglected. In a recent paper, we studied a class of capacitated arc routing problems from primarily a theoretical point of view. In this paper, we focus on the development and testing of algorithms for solving the capacitated Chinese postman problem. Extensive computational results are presented and analyzed.

INTRODUCTION

Two well-known uncapacitated routing problems are the traveling salesman problem (TSP) and the Chinese postman problem (CPP). The problem of visiting each node in a network and returning to the starting point while incurring minimal distance is the TSP. Since demands occur only at nodes, this is called a *node routing* problem. The problem of covering each arc in a network and returning to the starting point while minimizing the total distance traveled is the CPP. Since demands occur only on arcs in this setting, this is labeled an *arc routing* problem.

Capacitated variations of the TSP and the CPP reflect real-world situations more accurately. The vehicle routing problem (VRP), a capacitated node routing problem, has been studied by Beltrami and Bodin[1], Bodin and Golden[2], Christofides *et al.*[4], Golden *et al.*[8], Magnanti[10], and many others. Capacitated arc routing problems have received relatively little attention; exceptions include papers by Christofides[3], and Golden and Wong[9]. In a certain sense, the present paper may be viewed as a continuation of the paper by Golden and Wong and an outgrowth of the earlier paper by Christofides.

Our intent in this paper will be to focus on the development and testing of algorithms for solving the capacitated Chinese postman problem (CCPP). Applications of the CCPP include routing of street sweepers, snow plows, refuse collection vehicles, inspection of power lines or pipelines, etc.

†Bruce L. Golden is Chairman of the Department of Management Science and Statistics at the University of Maryland. His research interests include network optimization, mathematical programming, and applied statistics, and he has published numerous articles in these fields. He is currently an Associate Editor of *Networks*, a member of ORSA's Long Range Planning Committee, and Chairman of ORSA's Transportation Science Dissertation Prize Committee. Dr. Golden received his B.A. from the University of Pennsylvania in Mathematics. Later he earned his M.S. and Ph.D. in Operations Research from M.I.T.

‡James S. DeArmon is a systems engineer at the Mitre Corporation in McLean, Virginia, working on the development of aircraft collision avoidance systems. He is also a Ph.D. candidate at the University of Maryland, where he received his M.S. in Operations Research in 1981.

§Edward K. Baker is an Assistant Professor of Management Science in the School of Business Administration at the University of Miami. He received B.E.S. and M.S. degrees from Johns Hopkins University and a DBA from the University of Maryland. His research interests include integer programming, networks, and applications of operations research to problems in transportation and finance.

BACKGROUND

The CCPP is formally defined over an undirected network $G(N, E, C)$ where N is the set of nodes, E is the set of edges or arcs, and C is the arc traversal cost matrix. For each arc $(i, j) \in E$ we define an arc demand $q_{ij} > 0$. Given a fleet of vehicles of capacity W, the CCPP is then to find a set of cycles, each of which passes through the domicile (node 1), which satisfies all the arc demands at a minimal total distance traveled. (Note that the terms *cost* and *distance* are used interchangeably throughout this paper.)

The CCPP is a generalization of the CPP where each arc has unit demand and vehicle capacity $W \geq |E|$. That is, the CPP requires the determination of a minimal length cycle that traverses each arc in E at least once. The solution to the CPP may be obtained efficiently, i.e. in polynomial time, using an algorithm due to Edmonds and Johnson[7] (see Minieka[11] for an overview). If all the nodes in the CPP network are of even degree, then the CPP may be solved by tracing an Euler cycle. A procedure for identifying an Euler cycle in a graph is given by Nijenhuis and Wilf[12]. If some of the nodes in the CPP network are of odd degree, a matching algorithm is used to identify the arcs of E of minimal total length which when duplicated in G make all nodes of even degree. Once all nodes are of even degree, the CPP is solved by obtaining an Euler cycle for the expanded graph.

To illustrate the nature of the CCPP, consider the network in Fig. 1 in which node 1 is the domicile, $W = 4$, and arcs are labeled by ordered pairs (c_{ij}, q_{ij}).

One feasible solution is the set of tours

$$\underline{1\ 2\ 4}\ 3\ 1$$

$$\underline{1\ 4\ 3\ 1}$$

(underlines indicate the servicing of arc demands) with a total distance of 15 units. All arcs in the network are undirected and may be traversed and/or serviced in either direction.

In this simple example, we've postponed the question of how to construct tours in order to *minimize* total distance. At this point, we present the example merely as an illustration of the CCPP.

The CCPP is of such computational complexity (Golden and Wong[9] show it to be NP-hard) that we turn our attention to the development and testing of heuristic algorithms. These procedures provide upper bounds on the optimal solution. In order to assess deviations from optimality, tight lower bounds are desired as well.

In an early paper, Christofides[3] presents a lower-bounding procedure and uses it to evaluate the accuracy of his proposed CCPP heuristic. Golden and Wong[9] prove the lower-bounding procedure to be incorrect, provide a counter-example, and then describe a valid lower-bounding procedure which is based on an associated matching problem.

HEURISTIC ALGORITHMS

In this section, we present three classes of algorithms for the CCPP. The two types that are new, path-scanning and augment-merge, are presented in detail. The construct and strike algorithm, proposed by Christofides[3], is described briefly.

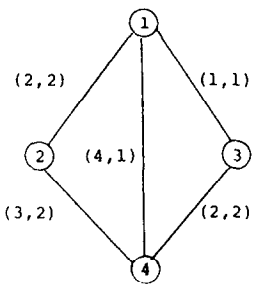

Fig. 1.

Path-scanning algorithms

The basic idea here is to construct one cycle at a time based on a certain myopic optimization criterion. In forming each cycle, a path is extended by joining the arc that looks most promising until vehicle capacity is exhausted. Then, the shortest return path to the domicile is followed. We refer to such a procedure as a path-scanning algorithm.

In this investigation, five path-scanning optimization criteria are considered. Each of these criteria is used to generate a complete CCPP solution. The final solution from this approach is the best solution of the five. The arc-selection criteria are described next.

Given a path from the domicile to node i, we choose, subject to capacity constraints, the arc (i, j) such that: (1) the distance, c_{ij}, per unit remaining demand is minimized; (2) the distance, c_{ij}, per unit remaining demand is maximized; (3) the distance from node j back to the domicile is minimized; (4) the distance from node j back to the domicile is maximized; (5) if the vehicle is less than half-full, maximize the distance from node j back to the domicile, otherwise minimize this distance.

The rationale for each of these criteria is as follows. With criteria 1 and 2, we look at a cost/demand ratio for each arc. This ratio can be viewed as a cost per unit demand of servicing an arc. With criterion 1, we minimize this ratio at each step and look for a large and quick payoff. In contrast, using criterion 2, we maximize the ratio, seeking to incur larger expenses early on and thereby get them *out of the way*. With criterion 3, we tend to obtain shorter cycles whereas criterion 4, in general, yields longer cycles. We point out that a smaller number of longer cycles *may* have shorter total distance. Criterion 5 represents a hybrid approach. Under each of the five criteria, once a vehicle capacity is fully utilized, the remainder of the cycle is given by the shortest return path to the domicile. The smallest total distance set of cycles is selected as the outcome of the algorithm.

The path-scanning algorithm has the nice feature of simplicity. It is coded in fewer than two hundred lines of FORTRAN. (The construct and strike algorithm and the augment–merge algorithm, presented subsequently, each require about one thousand lines of FORTRAN code. Note that this comparison is fairly crude, as the coding was obviously not done with the explicit goal of minimizing lines of code.) As will be discussed later, the path-scanning algorithm produces good results and requires a relatively small amount of computer execution time as well.

The construct and strike algorithm

In 1973, Christofides [3] introduced the CCPP as a generalization of the CPP and presented a heuristic algorithm (based upon an optimal algorithm for the CPP) for obtaining approximately optimal solutions to the CCPP. We will refer to this procedure as the construct and strike algorithm since it repeatedly constructs feasible cycles and then strikes or removes them.

The construct and strike algorithm requires the repetition of three steps. The first step attempts to remove cycles with the following two properties from the current graph: (i) the removal of the demand arcs serviced on the cycle must not disconnect the current graph; (ii) the sum of the arc demands serviced on the cycle must not exceed vehicle capacity. In general, cycles possessing property (i) are formed from arcs in one particular area of the graph. Property (ii) is necessary to ensure the feasibility of the solutions produced.

The determination of desirable cycles in Step 1 of the algorithm is a very important component in the procedure's implementation. Christofides does not offer a particular method for identifying such cycles, however, he notes that they must be formed with the knowledge of vehicle capacity requirements and arc demands on both the outward and inward portions of the route. Although Christofides obtains these cycles by visual inspection in the example he works through, the path-scanning criteria suggested in the previous section could be used for this purpose.

The second step in Christofides' heuristic records the demand arcs serviced by the cycles created in Step 1 and deletes these arcs from the current graph. Additionally, artificial arcs, which may have been added to the graph in Step 3, are deleted such that the graph passed on to Step 3 of the algorithm contains only nodes and unserviced demand arcs. (In the first pass no artificial arcs will exist.)

The third and final step of the algorithm accepts the graph from Step 2 and adds a minimum cost set of artificial arcs to the graph to ensure that the domicile (node 1) is of positive and even degree and that all other nodes in the current graph are of even degree. The algorithm accomplishes this by finding a minimum weighted matching on the odd degree nodes of the current graph. Artificial arcs and artificial arc costs are added based on the minimum cost paths between the odd degree nodes. When the domicile is of zero degree it is replaced by two artificial nodes that are connected by an arc with a prohibitively high cost to ensure that it is not used in the matching. The current graph including the artificial arcs determined by the matching solution is then presented to Step 1 of the algorithm. The procedure iterates through Steps 1–3 until no demand arcs remain unserviced.

In Christofides' paper[3], a 12-node, 22-arc problem is solved by hand in order to illustrate the procedure. His solution is of length 323. As shown in a later section of this paper, our computerized implementation of the construct and strike algorithm could not equal his hand-derived result. Instead, we obtained a solution of 331. One possible reason for this difference might be that an automated implementation simple cannot, at least for small networks, recognize good cycles as easily as can the human eye.

We implemented Christofides' procedure in FORTRAN. The programming was done in a straightforward manner, avoiding obvious inefficiencies and using a very fast minimal cost matching subroutine originally developed by Derigs and Kazakidis[6]. In our implementation, we encountered several minor ambiguities in Christofides' procedure, but we resolved these reasonably quickly; we do not go into details here.

The augment–merge algorithm

The heuristic algorithm discussed in this section will be referred to as the augment–merge procedure. Originally proposed by Golden and Wong[9], we have since implemented, modified, and thoroughly tested the procedure. The *basic procedure* (taken from [9]) is outlined below.

Step 1. INITIALIZE—all demand arcs are serviced by a separate cycle.

Step 2. AUGMENT—starting with largest cycle available, see if a demand arc on a smaller cycle can be serviced on a larger cycle.

Step 3. MERGE—subject to capacity constraints, evaluate the merging of any two cycles (possibly subject to additional restrictions). Merge the two cycles which yield the largest positive savings.

Step 4. ITERATE—repeat Step 3 until finished.

At this point, it seems appropriate to discuss each of the above four steps in more detail. In Step 1, we construct initial cycles such that each one services exactly one arc. We do this by finding the shortest path from each endpoint of an arc to the domicile.

In Step 2, we order available cycles from longest to shortest with respect to distance. The longest cycle receives the number 1, etc. Next, we attempt to service the demand arc from each shorter (i.e. higher-numbered) cycle on a longer (lower-numbered) one. For example, if the demand arc in cycle $i+1$ can be serviced on cycle i, we do so and then discard cycle $i+1$. Similarly, if the demand arc on cycle $i+2$ can be serviced on cycle i, we do so and then discard cycle $i+2$, and so on. If cycle i reaches capacity, it is set aside as completed, and the augmentation begins on the next available, higher-numbered cycle. We continue until all cycles have been considered.

Step 2 results in the formation of a number of cycles that fall into one of three classes: (1) cycles filled to capacity and set aside (no further processing required); (2) cycles discarded because their demand arcs could be serviced by longer cycles; (3) cycles with one or more demand arcs but not filled to capacity.

In Step 3, we seek to perform cost-effective pairwise concatenations or mergers of the cycles falling into the third category at a node common to both cycles. The merged cycle must obey the capacity limitation and must service the same set of arcs as the original two cycles.

For example, consider the following hypothetical cycles:

cycle 1—1 2 9 10 3 1
cycle 2—1 2 3 5 6 12 1
cycle 3—1 12 6 7 8 11 1.

Cycles 1 and 2 each include node 2 and can be merged to form the new cycle

$$1\ \ 3\ \ \underline{10\ \ 9}\ \ \underline{2\ \ 3\ \ 5}\ \ 6\ \ 12\ \ 1.$$

Cycles 2 and 3 each include node 12 and can be merged to form the new cycle

$$1\ \ \underline{2\ \ 3\ \ 5}\ \ 6\ \ 12\ \ 6\ \ 7\ \ \underline{8\ \ 11}\ \ 1.$$

Cycles 2 and 3 also have node 6 in common. Such a merger would result in the cycle

$$1\ \ \underline{2\ \ 3\ \ 5}\ \ 6\ \ 7\ \ \underline{8\ \ 11}\ \ 1,$$

which is necessarily shorter than the last merger. Cycles 1 and 3 have no nodes in common, except for the domicile, and so no merger is possible.

Note that mergers are legal only if the combined demand does not exceed the vehicle capacity. Also note that even though cycles 1 and 2 each include node 3, the implied merger would be illegal since it would interrupt the serviced segment $\underline{2\ \ 3\ \ 5}$ in cycle 2. In a related instance, if cycle 2 were

$$1\ \ \underline{2\ \ 3\ \ 5}\ \ \underline{6\ \ 12}\ \ 1,$$

node 6 could no longer be considered a common node for merging cycles 2 and 3. We force the procedure to maintain the current servicing of demand arcs. That is, since the proposed merger using node 6,

$$1\ \ \underline{2\ \ 3\ \ 5}\ \ 6\ \ 7\ \ \underline{8\ \ 11}\ \ 1,$$

would not service $\underline{6\ \ 12}$, it is viewed by the procedure as an illegal merger.

After pairwise merging of all partially-filled cycles has been considered, the savings associated with each merger is evaluated by the expression

$$S_{ij} = l_i + l_j - m_{ij}$$

where S_{ij} = savings attributable to merger

l_k = length of kth cycle

m_{ij} = length of cycle resulting from merging cycles i and j.

The merger with the largest savings is selected.

Suppose the merger t_{ij} of cycles results in the largest savings S_{ij}. All other mergers which use cycle i or j must be updated to reflect the newly-created merged cycle t_{ij}. Once the cycle t_{ij} is created, we discard cycle j and replace the old cycle i with the merger t_{ij} assuming, without loss of generality, that $i < j$.

In the 3-cycle example discussed earlier in this section, we found that the following two mergers were possible:

$$\begin{array}{ll} t_{12}\text{—} & 1\ \ 3\ \ \underline{10\ \ 9}\ \ \underline{2\ \ 3\ \ 5}\ \ 6\ \ 12\ \ 1 \\ t_{23}\text{—} & 1\ \ \underline{2\ \ 3\ \ 5}\ \ 6\ \ 7\ \ \underline{8\ \ 11}\ \ 1. \end{array}$$

Assume that $S_{12} > S_{23}$. Then cycle 2 is discarded, merger t_{12} becomes the new cycle 1, and t_{23} must be re-evaluated as the merger of cycles 1 and 3 listed below:

$$\begin{array}{llllllllll} 1 & 3 & \underline{10} & \underline{9} & \underline{2} & \underline{3} & \underline{5} & 6 & 12 & 1 \\ 1 & 12 & 6 & 7 & \underline{8} & \underline{11} & 1. & & & \end{array}$$

Merger t_{13} would then become

1 11 8 7 6 5 3 2 9 10 3 1.

The process of selecting the largest savings among feasible mergers, discarding the higher-numbered source cycle, replacing the lower-numbered source cycle with the selected merger, updating other mergers, and scanning again for the best savings merger continues until no further mergers are possible.

The following problem will help to illustrate the procedure. We seek the CCPP solution to the network in Fig. 2 where the domicile is node 1, arc demands are all unity, vehicle capacity is 4, and arc lengths are as indicated.

Step 1 produces the nine cycles below ordered by length.

Cycle number	Cycle length	Cycle
1	16	1 4 6 1
2	15	1 5 6 1
3	14	1 4 1
4	13	1 3 6 1
5	10	1 5 1
6	10	1 2 6 1
7	8	1 6 1
8	8	1 3 1
9	6	1 2 1

Step 2 first attempts to augment cycle 1 with the demand arc from cycle 2, then cycle 3, etc. The augmentation attempts continue for higher-numbered cycles until we obtain the results given below.

Cycle number	Cycle
1	1 4 6 1
2	1 5 6 1
4	1 3 6 1
6	1 2 6 1

At this point, no cycle has yet reached capacity, so all are still candidates for merging. Step 3 produces the following initial mergers:

Merge name	Savings	Merger
t_{24}	8	1 5 6 3 1
t_{26}	8	1 5 6 2 1
t_{46}	8	1 3 6 2 1

The savings for the three above mergers are each 8 units. We arbitrarily select t_{24}. Cycle 4 is then discarded, cycle 2 becomes merger t_{24}, and we attempt to update mergers t_{26} and t_{46}. As is easily seen, no additional mergers are possible, so the algorithm stops. The final cycles are

1 4 6 1
1 5 6 3 1
1 2 6 1

for a total length of $16 + 20 + 10 = 46$. This happens to be an optimal solution.

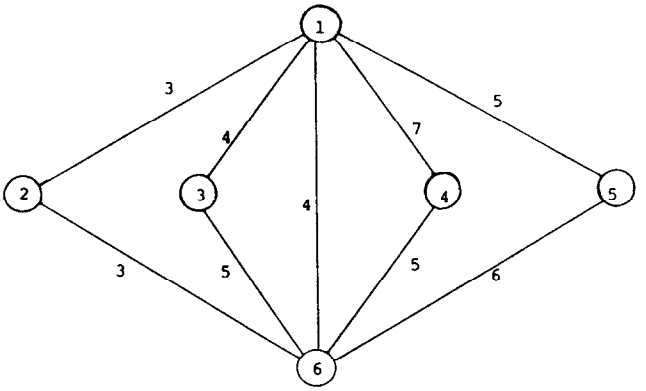

Fig. 2.

Variants of the augment–merge algorithm

In this section, we describe several modifications of the basic augment–merge approach and provide some preliminary computational results.

First of all, in experiments carried out by hand, we noticed that the cardinality of the shortest paths generated in the first step affects the final set of cycles. Between the endpoint of a demand arc and the domicile there may be several shortest paths. Since the arcs in these paths are the only ones utilized in subsequent steps of the algorithm, it seems reasonable that the paths themselves would influence solution values quite heavily. With this in mind, two versions of the algorithm have been implemented. The first generates minimal-cardinality shortest paths, and the second generates maximal-cardinality shortest paths. Both versions are applied to test problems to see if one version dominates the other one with respect to accuracy. Preliminary results are displayed in Table 1. Test problem 1 is Christofides' 12-node, 22-arc problem [3]. Test problems 2–7 were devised by DeArmon [5]. In only a few cases does the algorithm get close to the lower bound. In the remainder of this section, we describe ideas for improving the performance of the algorithm.

In a second experiment, we focus on the merge step and seek to improve the augment–merge algorithm by introducing some variability into the merging process. Instead of choosing the merger with the largest savings at each step, one of the top three mergers is chosen randomly. Since test problem five displays the greatest disparity in solution values obtained using the min- and max-cardinality versions of the augment–merge algorithm, we use the min-cardinality version of the algorithm on this problem in these experiments (this version gives the poor solution value of 429 in Table 1, 16% above the lower bound).

As shown in Table 2, three different sets of probabilities are tried. Each set is run for 100 repetitions. In column one, for example, where probabilities are 0.5, 0.3 and 0.2 for choosing the first, second and third best savings, respectively, a solution of 429 is obtained 40 times, a solution of 433 is obtained 13 times, etc., out of the 100 trials. Recall that the solution using the original merger-selection criterion is 429.

Unfortunately, this approach yields little improvement. In only one trial out of six hundred is an improved solution obtained. Note that as the probabilities place diminishing emphasis on the best merger, solution quality steadily declines. Test problem one was also examined in this context and the results were quite similar.

Table 1. Preliminary results with augment–merge algorithm

Test Problem Number	Heuristic Solutions min. card.	max. card.	Lower Bound	Solution/Lower Bound min. card.	max. card.
1	351	349	310	1.13	1.13
2	394	394	339	1.16	1.16
3	316	316	275	1.15	1.15
4	316	318	274	1.15	1.15
5	429	383	370	1.16	1.04
6	340	348	295	1.15	1.18
7	325	325	312	1.04	1.04

Table 2. Randomized selection among mergers, test problem 5 (unaltered algorithm yields 429)

	Probabilities Associated with Choice Among Top 3 Mergers		
Obtained Solution	Strategy A .5, .3, .2	Strategy B .33,.33,.33	Strategy C .25,.25,.5
427	0	0	1
429	40	22	3
433	13	12	4
435	2	2	6
437	8	14	9
439	3	4	2
441	11	15	22
443	11	4	5
445	4	4	7
447	6	3	9
449	2	5	12
451	0	4	1
453	0	0	5
455	0	2	4
457	0	2	3
459	0	0	2
463	0	1	0
465	0	3	2
467	0	0	2
471	0	3	0
479	0	0	1
Total No. Solution Trials	100	100	100

In a further attempt to improve the results of the merge step, we alter the savings function slightly. Recall that the savings available from a feasible merger (i.e. one not violating vehicle capacity) is evaluated by

$$S_{ij} = l_i + l_j - m_{ij}$$

and that we choose the largest S_{ij} at each iteration. The first modified savings function (MS1) that we propose is defined by

$$S^1_{ij} = l_i + l_j - m_{ij} + \lambda p^1$$

where λ = a scalar multiplier (to be varied over a range of values)

$$p^1 = \frac{\text{number of demand arcs serviced in merged tour}}{\text{total number of arcs in merged tour}}.$$

A second modified savings function (MS2) that we investigate is

$$S^2_{ij} = l_i + l_j - m_{ij} + \lambda p^2$$

where

$$p^2 = \frac{\text{total demand covered in merged tour}}{\text{vehicle capacity}}.$$

We know that distance saved is an important component of a merger since the problem objective function is to minimize total distance. The motivation behind the savings functions MS1 and MS2 is to characterize a second important component. In particular, we thought that it

might be beneficial to include in a savings function a measure of the degree of *packing* associated with the proposed merger. The scalar multiplier λ is necessary to adjust p^1 and p^2 so that they are approximately of the same size as l_k and m_{ij}, which are distance values. We tested approximately fifteen values of λ between 0 and 300 on the first test problem; the five values used between 30 and 150 seemed to provide reasonable results. Obviously, if $\lambda = 0$ we would have the original savings function.

We used six values of λ on the seven test problems, applying both the min-cardinality and max-cardinality versions of the algorithm. Tables 3 and 4 display the results for each of the two modified savings functions. The two entries in each cell represent the min- and max-cardinality solutions.

Minor improvements were obtained with each of the two modified savings functions. For example, total length for test problem six decreased from 340 to 328 under MS1 and from 340 to 324 under MS2.

As an additional modification to the augment–merge algorithm, we added a constant value h to each arc length. Since all arcs available to the solution process are defined in the shortest path step of the procedure, we felt that such a modification would force us to consider a different subset of arcs. That is, we thought that adding a constant to each arc length would perturb the shortest paths sufficiently to introduce new arcs and, hence, a broader set of possible solutions. The added constants h would have to, of course, be subtracted out at the end of the algorithm.

Table 3. Results using modified savings 1, $S_{ij}^1 = l_i + l_j - m_{ij} + \lambda p^1$

	Test Problem Number						
	1	2	3	4	5	6	7
λ							
0	351 349	394 394	316 316	316 316	429 383	340 348	325 325
30	357 349	412 394	316 316	316 316	441 409	328* 328*	325 325
50	357 349	412 394	354 354	316 316	441 421	328* 328*	325 325
70	357 375	408 384*	354 354	316 316	441 421	328* 328*	325 325
100	383 375	408 384*	368 368	316 316	441 421	328* 328*	325 325
150	383 375	408 384*	368 368	316 316	441 451	328* 328*	325 325

* Improved solutions over $\lambda = 0$.

Table 4. Results using modified savings 2, $S_{ij}^2 = l_i + l_j - m_{ij} + \lambda p^2$

	Test Problem Number						
	1	2	3	4	5	6	7
λ							
0	351 349	394 394	316 316	316 316	429 383	340 348	325 325
30	399 349	380* 394	354 316	316 316	441 409	358 328	325 325
50	399 349	380* 394	368 354	316 316	479 421	358 328	325 325
70	399 375	380* 384*	368 354	316 316	479 421	394 328	359 325
100	399 375	380* 384*	368 368	316 316	455 421	394 328	359 325
150	326* 326*	394 394	343 343	290* 290*	405 439	324* 324*	366 366

*Improved solutions over $\lambda = 0$.

Table 5 shows the results of varying h from -2 to 30 over the seven test problems. The smallest h value should be the negative of the smallest arc length—when added to arcs, the modified arc lengths are then guaranteed to be non-negative. We do this to preclude the formation of negative cycles. The largest value of h should be approximately equal to the largest original arc length. Between these extremes we select several intermediate values. The improvement in the results is dramatic. Four of the seven problems have improved solutions when h is non-zero.

In Table 6, the following information is presented: (i) the best min-cardinality solutions, varying h; (ii) the best max-cardinality solutions, varying h; (iii) problem lower bounds; (iv) the ratio of min-cardinality solution to lower bound; (v) the ratio of max-cardinality solution to lower bound; (vi) solutions when $h = 0$; (vii) the ratios of "$h = 0$" solutions to the lower bounds.

To appreciate the enhancements of the augment–merge procedure, we examine Table 6 closely. Comparing rows 5 and 7, one can see that the modified solution technique reduced the solution to lower bound ratio to 1.10 or less in problems 1, 2, 4 and 6. Problems 5 and 7, already displaying respectable ratios of only 1.04, showed no improvement. Problem 3, with a rather poor ratio of 1.15, also showed no improvement.

In a further attempt at improving results from the augment–merge procedure, we combined the two previous strategies, i.e. varying λ in the packing component of MS1 or MS2 and varying h to alter arc lengths. Results from varying λ over the values 30, 50, 70 and 100, using modified savings 1 and 2, and varying h over the values -2, -1, 0, 1, 2, 5, 10, 13 and 20 were obtained. Although there was no improvement over the case where $\lambda = 0$, there were a few replications of the best results from the "$\lambda = 0$" case, and quite a few near-best solutions. Results were fairly stable, without much variation.

Table 5. Results from adding h to arc lengths

	Test Problem Number						
	1	2	3	4	5	6	7
h							
-2	337 337	379 381	346 346	322 322	413 403	343 343	359 359
-1	373 373	381 381	316 316	316 316	419 383	331 331	359 359
0	351 349	394 394	316* 316*	316 316	429 383*	340 348	325* 325*
1	352 351	395 394	316 316	316 316	424 383	356 348	325 325
2	333 333	367* 367*	329 329	316 316	405* 405	356 324*	325 325
3	333 333	367 367	329 329	316 316	405 413	356 324	325 325
7	333 333	375 375	343 343	316 316	413 413	356 324	325 325
10	333 333	375 375	343 343	290* 290*	413 413	356 324	325 325
13	344 344	394 394	343 343	290 290	405 439	324* 324	366 366
17	326* 326*	394 394	343 343	290 290	405 439	324 324	366 366
20	326 326	394 394	343 343	290 290	405 439	324 324	366 366
25	326 326	394 394	343 343	290 290	405 439	324 324	366 366
30	326 326	394 394	343 343	290 290	405 439	324 324	366 366

*Best solutions: min- and max-cardinality versions

Table 6. Comparing solutions from Tables 1 and 5

	Test Problem Number						
	1	2	3	4	5	6	7
1) Best min-cardinality solution, varying h	326	367	316	290	405	324	325
2) Best max-cardinality solution, varying h	326	367	316	290	383	324	325
3) Problem lower bound	310	339	275	274	370	295	312
4) Min-card. solution / lower bound	1.05	1.08	1.15	1.06	1.09	1.10	1.04
5) Max-card. solution / lower bound	1.05*	1.08*	1.15	1.06*	1.04	1.10*	1.04
6) Solution when h = 0	349	394	316	316	383	340	325
7) "h = 0" solution / lower bound	1.13	1.16	1.15	1.15	1.04	1.15	1.04

* Solution improved from h = 0 case

Computational comparison of algorithms

In order to assess the accuracy of the path-scanning, augment–merge, and construct and strike algorithms more conclusively, we expanded the set of test problems from seven to twenty-five in total. These twenty-five networks are diverse in terms of sparsity and topological structure. Some networks have all arc demands of the same value, others have arc demands varying between 1 and 4, 1 and 9, or 1 and 16. The networks are displayed in DeArmon's thesis [5].

In testing the augment–merge algorithm, we initially ran each of the 25 problems one hundred and eighty times, using each combination of min- and max-cardinality shortest paths, nine values of the arc addition constant h, five values of λ, and the two modified savings functions. Since we are comparing solution accuracy *and* computer run time, we next sought to limit the number of runs required by the augment–merge procedure. After numerous computational experiments (see [5] for details) we decided to limit the number of augment–merge program executions to 24. The solutions were generated using both the min- and max-cardinality shortest paths, six values of h (-2, -1, 0, 1, 2 and 20), two values of λ (0 and 30), and MS1.

We now compare the augment–merge solutions to those obtained using the path-scanning and the construct and strike algorithms, relying upon the lower bound of each problem as a convenient reference point. Table 7 presents these computational results. The path-scanning entries represents 5 program executions and the augment–merge entries represent 24 program executions.

We note that each algorithm seems to obtain its worst solutions on test problems 8–14. These problems, constructed to be perverse CCPP's, either have a domicile node that is stranded (has small degree) or are rather spread-out (remote arcs require a large number of arc traversals to be reached). Test problems 15 through 20, 24 and 25 are very dense networks (all are complete except for problem 24 which is 80% complete) and, in general, each algorithm performs extremely well on these problems. An interesting case is problem 17 on which all algorithms find the lower bound, which must be the optimal solution.

Which algorithm yields the most accurate results? In order to answer this question we look at average percentage above lower bound, average rank, and worst case performance as displayed in Table 8.

Table 7. Comparison of solutions and ratios for 25 test problems

Test Problem Number	Lower Bound	Path-Scanning		Augment-Merge		Construct and Strike	
1	310	316	1.02	326	1.05	331	1.07
2	339	367	1.08	367	1.08	418	1.23
3	275	289	1.05	316	1.15	313	1.14
4	274	320	1.17	290	1.06	350	1.28
5	370	417	1.13	383	1.04	475	1.28
6	295	316	1.07	324	1.10	356	1.21
7	312	357	1.14	325	1.04	355	1.14
8	144	171	1.19	164	1.14	186	1.39
9	150	172	1.15	180	1.20	194	1.29
10	258	416	1.61	356	1.40	407	1.58
11	253	355	1.40	339	1.34	364	1.44
12	275	302	1.10	302	1.10	364	1.32
13	395	424	1.07	443	1.12	501	1.27
14	424	560	1.32	573	1.35	655	1.54
15	544	592	1.09	560	1.03	560	1.03
16	100	102	1.02	102	1.02	112	1.12
17	58	58	1.00	58	1.00	58	1.00
18	127	131	1.03	131	1.03	149	1.17
19	91	93	1.02	91	1.00	91	1.00
20	164	168	1.02	170	1.04	174	1.06
21	55	57	1.04	63	1.15	63	1.15
22	121	125	1.03	123	1.02	125	1.03
23	156	168	1.08	158	1.01	165	1.06
24	200	207	1.04	204	1.02	204	1.02
25	233	241	1.03	237	1.02	237	1.02

Table 8. Comparison of performance of the algorithms

	Path-Scanning	Augment-Merge	Construct and Strike
Average Percentage above Lower Bound	11.60	10.04	18.96
Average Rank Among the Three Algorithms	1.64	1.32	2.20
Worst Solution Among 25 Test Problems in Terms of Percentage Above Lower Bound	1.61	1.40	1.58

The average percentage above the problem lower bound is computed from the twenty-five ratios from Table 7. The average rank is computed using the assignment of numbers 1, 2, or 3 for the best, the second-best, and the third-best algorithm on each problem. For example, on problem 1, path-scanning obtains 316, augment–merge obtains 326, and construct and strike gets 331, so the rankings are 1, 2 and 3, respectively. (As an aside, the solution value 316 for problem 1 is an improvement on the previous best known length which was conjectured by Christofides[3] to be optimal. The associated cycles are listed below

1 12 6 7 1
1 2 9 10 1
1 4 2 3 5 12 1
1 12 7 8 10 11 5 12 1
1 12 6 5 3 4 2 9 11 8 7 6 12 1.)

The average rank is the sum of ranks for an algorithm divided by 25. If there is a tie, duplicate

ranks are assigned. To illustrate, in problem 18, lengths are 131, 131 and 149 and corresponding ranks are 1, 1 and 2. The worst case performance is the largest ratio of length to lower bound obtained over the 25 test problems.

Computer run times for the three algorithms vary quite extensively. Below run times are displayed for three test problems in CPU seconds on the Univac 1108. These times are fairly typical.

Test problem	Path-scanning	Augment–merge	Construct and strike
1	1.6	38.4	1.6
10	12.3	91.2	11.0
25	2.3	165.6	4.6

None of the three codes was written with the intent of achieving great efficiencies. Roughly speaking, the path-scanning and the construct and strike algorithms are comparable in run time. The augment–merge algorithm takes considerably more computer time than either of the two other procedures.

In practical terms, the greater accuracy of the augment–merge algorithm over the path-scanning algorithm may more than compensate for the associated increase in computer run time, especially if the potential savings are in the millions of dollars. On the other hand, if computing time and cost is a primary concern, the path-scanning procedure is the one that should probably be applied.

CONCLUSIONS AND FURTHER REMARKS

The capacitated Chinese postman problem is an interesting combinatorial problem with many real-world applications. In contrast to node routing problems which have been the subject of much research over the last several decades, arc routing problems have received comparatively little attention.

In this paper, two new algorithms, path-scanning and augment–merge, have been developed and compared with an existing algorithm. In addition, a valid lower-bounding procedure has been automated which provides a standard against which to judge results. Notwithstanding fairly substantial run time requirements, the augment–merge algorithm obtains better solutions than the other two procedures—approximately 10% above problem lower bounds.

Possible directions for future research are many and varied. There is room for more experimentation in order to fine-tune the augment–merge algorithm and improve its accuracy. Also, since we found our algorithms not to work well on more general capacitated arc routing problems (in which arc demands can be zero) research and development of procedures for handling these problems should be encouraged.

REFERENCES

1. E. Beltrami and L. Bodin, Networks and vehicle routing for municipal waste collection. *Networks* **4**(1), 65–94 (1974).
2. L. Bodin and B. Golden, Classification in vehicle routing and scheduling. *Networks* **11**(2), 97–108 (1981).
3. N. Christofides, The optimum traversal of a graph. *Omega* **1**(6), 719–732 (1973).
4. N. Christofides, A. Mingozzi and P. Toth, Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. *Math. Prog.* **20**(3), 255–282 (1981).
5. J. DeArmon, *A comparison of heuristics for the capacitated Chinese postman problem.* Masters Thesis, University of Maryland at College Park (1981).
6. U. Derigs and G. Kazakidis, *On two methods for solving minimal perfect matching problems*, Technical Report. University of Cologne, Cologne, West Germany (May 1979).
7. J. Edmonds and E. Johnson, Matching, Euler tours and the Chinese postman. *Math. Prog.* **4**, 88–124 (1973).
8. B. Golden, T. Magnanti and H. Nguyen, Implementing vehicle routing algorithms. *Networks* **7**(2), 113–148 (1977).
9. B. Golden and R. Wong, Capacitated arc routing problems. *Networks* **11**(3), 305–315 (1981).
10. T. Magnanti, Combinatorial optimization and vehicle fleet planning: perspectives and prospects. *Networks* **11**(2), 179–213 (1981).
11. E. Minieka, The Chinese postman problem for mixed networks. *Mngmt Sci.* **25**, 643–648 (1979).
12. A. Nijenhuis and H. Wilf, *Combinatorial Algorithms.* Academic Press, New York (1975).