Efficient heuristics for the Rural Postman Problem

GW Groves*   JH van Vuuren†

Received: 29 July 2004; Revised: 16 December 2004; Accepted: 9 February 2005

Abstract

A local search framework for the (undirected) Rural Postman Problem (RPP) is presented in this paper. The framework allows local search approaches that have been applied successfully to the well-known Travelling Salesman Problem also to be applied to the RPP. New heuristics for the RPP, based on this framework, are introduced and these are capable of solving significantly larger instances of the RPP than have been reported in the literature. Test results are presented for a number of benchmark RPP instances in a bid to compare efficiency and solution quality against known methods.

Key words: Chinese Postman Problem, Rural Postman Problem, Travelling Salesman Problem.

1 Introduction

Consider a weighted graph $G = (V, E)$, with vertex set $V = \{v_1, \ldots, v_p\}$, edge set $E$, and edge weights denoted by $c(i, j)$ for all $v_i v_j \in E$. The well-known Chinese Postman Problem (CPP) is the problem of determining a minimum-weight closed route traversing each edge $v_i v_j \in E$ at least once (Guan, 1962). The CPP is tractable and may be solved in $O(|V|^3)$ time (Edmonds and Johnson, 1973). The Rural Postman Problem (RPP) is a generalisation of the CPP in which a subset of the edges $E_r \subseteq E$ (called required edges) have to be traversed. It is the problem of determining a minimum-weight closed route traversing each edge in $E_r$ at least once. The RPP is NP–Hard (Lenstra and Rinnooy Kan, 1976), except when $E_r = E$, in which case the problem reduces to the CPP. The above CPP and RPP definitions for undirected graphs have been generalised in many ways, and algorithms catering for directed and mixed graphs, for example, have been introduced — see Ball et al. (1995), Dror (2000) and Eiselt et al. (1995a, 1995b) for an overview. These versions of the RPP have many applications — see, for example, Angel et al. (1972), Beltrami and Bodin (1974), Bennet and Gazis (1972), Bodin and Berman (1979), Bodin et al. (1989), Bodin and Kursh (1979), Braca et al. (1993), Desrosiers et al. (1986), Eglese (1994), Eglese and Murdock (1991), Gelders and Cattrysse (1991), Ghiani

*Department of Industrial Engineering, University of Stellenbosch, Private Bag X1, Matieland, 7602, South Africa
†Corresponding author: Department of Applied Mathematics, University of Stellenbosch, Private Bag X1, Matieland, 7602, South Africa, email: vuuren@sun.ac.za

In this paper, local search heuristics are presented for the (undirected) RPP. Heuristic methods stand in contrast to exact methods (i.e., algorithms that yield optimal solutions) in that they are designed with the aim of finding good (though not necessarily optimal) solutions without expending much computational execution time. The class of so-called local search heuristics is a family of methods that operates by iteratively performing transformations (referred to as moves) to existing solutions to an optimisation problem in a way that tends to (but is not guaranteed to) improve the solution as the search progresses. Typically, a local search heuristic operates by considering a number of candidate moves during each iteration, and selects the best one to perform on the solution. During the next iteration, the process is repeated on the transformed solution, and so on.

Perhaps the best known heuristic for the undirected RPP is Frederickson’s heuristic (Frederickson, 1979). This heuristic is similar to Christofides’ heuristic for the Travelling Salesman Problem\(^1\) (TSP), and operates by adding artificial edges (representing shortest paths between the relevant vertices in G) to the subgraph induced by \(E_r\) in a way that yields a connected, Eulerian graph (i.e., one in which it is possible to find a closed route traversing each edge exactly once). Hertz et al. (1999) introduced a family of local search heuristics for the RPP, while Fernández de Córdoba et al. (1998) employed a heuristic based on Monte Carlo principles. Exact algorithms have been proposed by Christofides et al. (1986), Corberan and Sanchis (1991), Ghiani and Laporte (2000), and Letchford (1996). The algorithm by Ghiani and Laporte has been used to solve instances with 350 vertices to optimality, which seem to constitute the largest instances previously addressed by either an exact or heuristic method in the literature. The heuristic presented in this paper is capable of (approximately) solving considerably larger instances of the RPP, and the quality of its solutions compare very favourably to those documented in the literature for benchmark problems.

This paper is structured as follows. A local search framework is described in §2, which allows local search moves that have traditionally been applied successfully to Vertex Routing Problems (VRPs), such as the TSP, also to be applied to Arc Routing Problems (ARPs), such as the RPP and the Capacitated Arc Routing Problem\(^2\). This transition is achieved by the introduction of a complexity reduction method (in §2.3) in which both aspects of the order (in §2.1) and direction (in §2.2) of edge traversals are accommodated. The section is concluded with a worked example, before we report results obtained by the heuristics for a number of benchmark RPP instances in §3. The paper closes with a short conclusion in §4.

\(^1\)The problem of finding a minimum–weight closed route, containing every vertex of a weighted graph.

\(^2\)The problem of finding a set of closed routes of minimum total weight in a weighted graph, satisfying the following conditions: (i) each route starts and ends at a specified vertex, representing a depot, (ii) each edge in \(E_r\) is traversed at least once by some route, and (iii) the sum of demands of each route does not exceed vehicle capacity.
2 Local Search Framework

Denote a solution to the RPP by the sequence \( S = \langle (v_{s_1}, v_{t_1}), (v_{s_2}, v_{t_2}), \ldots, (v_{s_n}, v_{t_n}) \rangle \) of required edges in the order in which they are traversed. Traversals taking place between the required traversals are omitted from the sequence and are assumed to take place along routes corresponding to shortest distances between the required edges of the sequence. The total weight of the route is therefore given by

\[
C(S) = \sum_{i=1}^{n} c(s_i, t_i) + \sum_{j=1}^{n-1} d(t_j, s_{j+1}) + d(t_n, s_1),
\]

where \( d(k, \ell) \) denotes the shortest distance between two vertices \( v_k, v_\ell \in V(G) \), and \( c(i, j) \) is the cost weight associated with the edge \( v_i v_j \), as before.

2.1 Applying Local Search Moves

In a local search framework moves are performed on candidate solutions to the RPP that directly specify the order in which required edges are traversed in the transformed solution. An example of such a move is one that simply exchanges the order in which two required edges are traversed. Given, for example, the route

\[
S_1 = \langle (3, 4), (4, 1), (5, 6), (5, 4), (6, 8) \rangle,
\]

the edges (3, 4) and (5, 4) might be exchanged, to yield the transformed route

\[
S_1^* = \langle (5, 4), (4, 1), (5, 6), (3, 4), (6, 8) \rangle.
\]

Typically, one would consider all pairs of these exchanges during a single iteration of the search and then perform one that yields a route of minimum overall cost. In the above example, the traversal order of required edges was altered, but not their traversal directions. However, it may be better to traverse the edge (3, 4) in \( S_1^* \), for example, in the direction (4, 3) instead of in the direction (3, 4). Consequently, it is necessary to determine the optimal directions of traversals of required edges in \( S_1^* \) after performing an exchange.

Applying a move therefore involves altering the order of the required edges in the route, and then determining their directions of traversal. This requirement for determining traversal directions results in an increased time complexity, when compared to applying the same type of move to a VRP. However, by using a complexity reduction method presented later, it is, in fact, possible to determine the cost of a route without redetermining the traversal directions of all of its required edges. This allows for the development of comparatively efficient procedures for many ARPs.

The moves considered in this paper for the RPP are slightly more involved than the above example, and are based on the well-known Two–Opt (Flood, 1956 and Croes, 1958) and Three–Opt (Bock, 1958 and Lin, 1965) procedures for the TSP. The application of the Two–Opt method is explained with the aid of Figure 1(a), depicting a solution for a small, hypothetical instance of the RPP. The vertices in the figure represent the required edges of the problem, and the solid arcs represent shortest paths between vertices incident to
required edges. The Two–Opt procedure is applied by deleting two of the edges, and re–
connecting the route so that it corresponds to the dotted arcs, again using shortest paths
between the relevant vertices. Two–Opt is typically implemented as a post–optimisation
procedure by performing the best Two–Opt move out of all possible moves, and then
repeating this process until no improving solution can be found.

The Three–Opt procedure, depicted in Figure 1(b), is similar to Two–Opt, but three
dges are deleted instead and are replaced with shortest paths in a differently connected
traversal order. Note that there are eight possible ways of connecting three route segments
in this way. Of course the Three–Opt procedure includes Two–Opt as a special case. The
vertex 0 in Figure 1 represents a domicile vertex, from which a route is considered to start
and end, and may represent a depot in a practical context. However, for the RPP, in
which a specific domicile vertex is not specified, the vertex 0 may be taken as any vertex
incident to a required edge.

Returning to our previous small example sequence for the RPP, the solution sequence

\[ S^2 = \{ (3, 4), (5, 4), (5, 6), (4, 1), (6, 8) \} \]
is found upon applying the Two–Opt procedure to $S^1$, deleting the shortest path between the first and second required edges, and between the fourth and the fifth required edges in $S^1$ and assuming that the traversal directions of the edges are not also reversed. The Three–Opt procedure for the RPP may be applied in a similar manner.

However, as mentioned earlier, the optimal traversal directions for the edges also have to be determined after performing the standard Two–Opt or Three–Opt operation.

### 2.2 Determining Traversal Directions

The algorithm described in this section may be used to compute optimal traversal directions for the edges in a solution sequence $S$ for the RPP, given a fixed order of the edges in the sequence. The algorithm may be applied after each local search move performed in order to yield the smallest cost weight for the new ordering of $S$.

Consider a routing sequence $S = \langle (v_{s_1}, v_{t_1}), (v_{s_2}, v_{t_2}), \ldots, (v_{s_n}, v_{t_n}) \rangle$ and construct, from the solution sequence, a directed, layered auxiliary graph $L$ with vertex set $V(L) = \{b, s_1t_1, s_1s_2t_2, s_2t_2, \ldots, s_nt_n, t_n, e\}$. The first and last layer of the auxiliary graph each consists of a single vertex, and the other layers each consist of two vertices. Each internal layer of the graph represents the two possible active traversal directions for an edge in $S$. The labels $b$ and $e$ represent the vertices at which the route is to begin and end. For a closed route $v_b = v_e$, and in the RPP $v_b = v_e$ is incident to an edge in $E_r$.

![Diagram](image)

**Figure 2:** The layered graph $L$, corresponding to the solution sequence $S = \langle (v_{s_1}, v_{t_1}), (v_{s_2}, v_{t_2}), \ldots, (v_{s_n}, v_{t_n}) \rangle$.

For each $i$ ($0 < i < n$) construct edges in $L$ directed from $s_it_i$ to $s_{i+1}t_{i+1}$ & $t_{i+1}s_{i+1}$, and from $t_is_i$ to $s_{i+1}t_{i+1}$ & $t_{i+1}s_{i+1}$. Also add edges directed from $b$ to $s_1t_1$ & $t_1s_1$ and from $s_nt_n$ & $t_ns_n$ to $e$. Assign a weight of $d(t_i, s_{i+1})$ [$d(s_i, t_{i+1})$, respectively] to the edge in $L$ between $s_it_i$ and $s_{i+1}t_{i+1}$ [$t_is_i$ and $t_{i+1}s_{i+1}$, respectively] for every $0 < i < n$, where $d(i, j)$ denotes the cost of a shortest path from $v_i$ to $v_j$, as before. Similarly assign a weight of $d(t_i, t_{i+1})$ [$d(s_i, s_{i+1})$, respectively] to the edge in $L$ between $s_it_i$ and $t_{i+1}s_{i+1}$ [$t_is_i$ and $s_{i+1}t_{i+1}$, respectively] for every $0 < i < n$. Assign a weight of $d(b, s_1)$ [$d(b, t_1)$...
respectively] to the edge in \( L \) between \( b \) and \( s_1t_1 \) \([t_1s_1 \text{ respectively}] \) and a weight \( d(t_n, e) \) \([d(s_n, e) \text{ respectively}] \) to the edge between \( s_nt_n \) \([t_ns_n \text{ respectively}] \) and \( e \).

Each route from \( b \) to \( e \) in \( L \) represents one way of arranging the traversal directions of edges within \( S \), and a shortest path from \( b \) to \( e \) in \( L \) represents a set of optimal directions by which to traverse the edges of \( S \). For example, if vertex \( t_2s_2 \) is on a calculated shortest path, then the second edge of \( S \) should be traversed from \( v_{t_2} \) to \( v_{s_2} \), and hence coded as \((v_{t_2}, v_{s_2}) \) in \( S \), instead of \((v_{s_2}, v_{t_2}) \). Note that the total weight of a route may be found by adding the sum of the weights of the required edges to the weight of the shortest path.

The computational complexity of finding a shortest path in a directed, acyclic graph, such as \( L \), is \( O(E(L) + V(L)) \) (Mehlhorn and Näher, 1999), because no updating of information, such as occurs in Dijkstra’s method (Dijkstra, 1959), is necessary during the algorithm execution. However, because no earlier layer of \( L \) can be reached from a later layer, and because each layer consists of a predetermined number of vertices, and is connected to the other layers in the particular manner shown, this complexity may be reduced further to \( O(V(L)) \).

The algorithm for computing a shortest path between \( b \) and \( e \) in \( L \) is a straightforward one, and is given by the pseudo–code listing in Algorithm 1. Let \( d_L(i, j) \) be the distance between vertices \( i \) and \( j \) in the auxilliary graph \( L \), and let \( w_L(i, j) \) be the weight of the edge between \( i \) and \( j \) in \( L \). In the algorithm \( pred(i) \) is a variable that stores the predecessor vertex of vertex \( i \) in a shortest path from \( b \) to \( e \) in \( L \).

Algorithm 1 (Shortest path through the layered graph \( L \))

**Input:** A layered graph \( L \), as described earlier, with edge weights \( w_L(i, j) \) for each edge \( ij \in E(L) \).

**Outputs:** (1) The shortest path from \( b \) to \( e \) in \( L \), stored in the variables \( pred(i) \), for all \( i \in V(L) \), (2) The shortest distance, \( d_L(b, e) \), from \( b \) to \( e \) in \( L \).

1. \( d_L(b, s_1t_1) \leftarrow w_L(b, s_1t_1) \), \( d_L(b, t_1s_1) \leftarrow w_L(b, t_1s_1) \), \( pred(s_1t_1) \leftarrow b \), \( pred(t_1s_1) \leftarrow b \).

2. For \( i \leftarrow 2, \ldots, n \):

   2a. \( val1 \leftarrow d_L(b, s_{i-1}t_{i-1}) + w_L(s_{i-1}t_{i-1}, s_it_i) \), \( val2 \leftarrow d_L(b, t_{i-1}s_{i-1}) + w_L(t_{i-1}s_{i-1}, s_it_i) \).

   2b. if \( val1 < val2 \) then \( \{d_L(b, s_it_i) \leftarrow val1, pred(s_it_i) \leftarrow s_{i-1}t_{i-1}\} \)

       else \( \{d_L(b, s_it_i) \leftarrow val2, pred(s_it_i) \leftarrow t_{i-1}s_{i-1}\} \).

   2c. \( val1 \leftarrow d_L(b, s_{i-1}t_{i-1}) + w_L(s_{i-1}t_{i-1}, t_is_i) \), \( val2 \leftarrow d_L(b, t_{i-1}s_{i-1}) + w_L(t_{i-1}s_{i-1}, t_is_i) \).

   2d. if \( val1 < val2 \) then \( \{d_L(b, t_is_i) \leftarrow val1, pred(t_is_i) \leftarrow s_{i-1}t_{i-1}\} \)

       else \( \{d_L(b, t_is_i) \leftarrow val2, pred(t_is_i) \leftarrow t_{i-1}s_{i-1}\} \).

3. \( val1 \leftarrow d_L(b, s_n t_n) + w_L(s_n t_n, e) \), \( val2 \leftarrow d_L(b, t_n s_n) + w_L(t_n s_n, e) \)

   if \( val1 < val2 \) then \( \{d_L(b, e) \leftarrow val1, pred(e) \leftarrow s_n t_n\} \)

   else \( \{d_L(b, e) \leftarrow val2, pred(e) \leftarrow t_n s_n\} \).
Every vertex of $L$ (except $b$) is considered once by the algorithm, and its complexity is therefore $O(V(L))$. The operation of assigning the computed distance of a vertex (from some other vertex) is referred to as labelling in the literature on shortest path algorithms. Each step of the algorithm performs a labelling operation on one or two vertices with respect to vertex $b$. In the next section, it is argued that the above procedure does not necessarily need to be applied each time that a move is evaluated.

2.3 Reducing Computational Complexity

The task of computing a shortest path in the layered graph in §2.2 is of linear computational complexity. Hence, if the shortest path is computed each time that a move is evaluated, it adds an order of magnitude to the computational complexity of the search heuristic. However, an approach is presented in this section that allows a move to be evaluated without having to re-compute the shortest path through the layered graph, provided that certain shortest path information is known about the untransformed solution (i.e., the solution before the move was implemented).

For the purposes of describing the method of computational complexity reduction, a move performed in a candidate solution $S$ to obtain a transformed candidate solution $S^*$ may be viewed as a sequence of $z$ subsequences, containing pairs, given by

$$\text{TRANSFORM}(S \rightarrow S^*) = \langle \langle (o_1, n_1), \ldots, (o_{l_1}, n_{l_1}) \rangle_1 \rangle, \langle \langle (o_{l_1+1}, n_{l_1+1}), \ldots, (o_{l_2}, n_{l_2}) \rangle_2 \rangle, \ldots, \langle \langle (o_{l_{z-1}+1}, n_{l_{z-1}+1}), \ldots, (o_{l_z}, n_{l_z}) \rangle_z \rangle,$$

where each subsequence represents adjacent elements in $S$ involved in the move. In each pair, $o$ and $n$ represent respectively the old position (in $S$) and the new position (in $S^*$) of an edge in $S$ that had its position changed in the sequence during the move. The edges of each subsequence are adjacent both before and after the move (else they are placed in separate subsequences), and the subsequences are listed in increasing order of the positions of their edges after the move, i.e. no subsequence placed later than another will contain a smaller $n$ value than the earlier subsequence, and the pairs within the subsequences themselves are arranged in increasing order of $n$. To illustrate this notation, consider again our hypothetical solution sequence $S^1$ in §2.1. Assuming that a move involves moving the second edge to the end of the sequence (which incidentally is one of the moves considered by a Three–Opt procedure), the resulting sequence is

$$S^3 = \langle (3, 4), (5, 6), (5, 4), (6, 8), (4, 1) \rangle,$$

and the move may be expressed as

$$\text{TRANSFORM}(S^1 \rightarrow S^3) = \langle \langle (3, 2), (4, 3), (5, 4) \rangle, \langle (2, 5) \rangle \rangle,$$

where the second subsequence, consisting of a single pair, represents the edge that is moved from position 2 to position 5, as reflected by its entry $(2, 5)$. The first subsequence, consisting of 3 pairs, represents the edges that move to the left to make place for the edge that moved to the end.

The complexity reduction method is based on the observation that edges of the same subsequence will retain their distances with respect to each other during a move. Note, however,
that when a subsequence describes edges that are reversed in the transformed sequence, the shortest paths take place along routes that are perhaps not as intuitive as in the case where the edges are not reversed. Consider, for example, the layered graph $L^4$, corresponding to a hypothetical solution sequence $S^4 = \langle (2, 1), (1, 3), (3, 2), (5, 6), (8, 7), (7, 2) \rangle$, shown in Figure 3. The bold edges in the figure indicate a shortest path (and optimal traversal directions) of $S^4$. Consider the Two–Opt move \( \text{TRANSFORM}(S^4 \rightarrow S^4^*) = \langle \langle (5, 2), (4, 3), (3, 4), (2, 5) \rangle \rangle \), which reverses the segment between (and including) the second and fifth edges in $S^4$. The layered graph of the transformed solution, $L^{4*}$, is shown in Figure 4. A shortest path between vertices 5 and 11 of $L^{4*}$ (shown as bold edges) has the same length as the shortest path from vertex 4 to 10 in $L^4$. Similarly, the distance from 4 to 10 in $L^{4*}$ equals the distance from 5 to 11 in $L^4$. The distance from 4 to 11 [5 to 10, respectively] in $L^{4*}$ equals the distance from 4 to 11 [5 to 10, respectively] in $L^4$.

Figure 3: The layered graph, $L^4$, corresponding to the solution sequence $S^4 = \langle (2, 1), (1, 3), (3, 2), (5, 6), (8, 7), (7, 2) \rangle$. A shortest path from the first to the last layer is shown using bold edges.

Figure 4: The layered graph $L^{4*}$ of the transformed solution sequence $S^{4*}$. The path indicated by the bold edges, between vertices 5 and 11, represents the same path as the path between vertices 4 and 10 in Figure 3.

The complexity reduction method is now described in general, using the alternative nota-
Efficient heuristics for the Rural Postman Problem

Algorithm 2 (Local search iteration using the complexity reduction method)

**Input:** A solution sequence $S$ with corresponding layered graph $L$, as described in §2.2.

**Output:** A transformed solution sequence, with the minimum total cost out of all the moves considered.

1. Calculate, in $L$, the shortest distance from all vertices to $e$ (the distances from each of these vertices to all vertices in later layers becomes known).

2. For each possible move (denote the transformed solution $S^*$, and its auxiliary graph $L^*$):
   - 2.1 Set $d_{L^*}(b, s_{n_{i-1}+1}t_{n_{i-1}}) = d_L(b, s_{n_{i-1}+1}t_{n_{i-1}})$, $d_{L^*}(s, t_{n_{i-1}+1}s_{n_{i-1}}) = d_L(b, t_{n_{i-1}+1}s_{n_{i-1}})$
   - 2.2 For each subsequence $k ← 1, \ldots, n$ in TRANSFORM($S \rightarrow S^*$):
     - 2.2.1 Assume $l_0 = 0$, $n_0 = 0$; $\text{label}(L^*, b, s_{n_i+1}, t_{n_i+1})$, $\text{label}(L^*, b, t_{n_i+1}, s_{n_i+1})$
     - 2.2.2 If $n_k > n_{k-1}$ (i.e., more than one pair in subsequence) calculate $d_{L^*}(b, s_{n_k}t_{n_k})$ and $d_{L^*}(b, t_{n_k}s_{n_k})$ as follows: (else they are already known)

   If $o_k > o_{k-1}$ (i.e., if the edges of subsequence are not reversed):
   - $d_{L^*}(b, s_{n_k}t_{n_k}) \leftarrow \min \left\{ \begin{array}{l}
   d_{L^*}(b, s_{n_{k-1}+1}t_{n_{k-1}+1}) + d_{L^*}(s_{n_{k-1}+1}t_{n_{k-1}+1}, s_{n_k}t_{n_k}) \\
   d_{L^*}(b, t_{n_{k-1}+1}s_{n_{k-1}+1}) + d_{L^*}(t_{n_{k-1}+1}s_{n_{k-1}+1}, s_{n_k}t_{n_k})
   \end{array} \right.$
   - $d_{L^*}(b, t_{n_k}s_{n_k}) \leftarrow \min \left\{ \begin{array}{l}
   d_{L^*}(b, s_{n_{k-1}+1}t_{n_{k-1}+1}) + d_{L^*}(s_{n_{k-1}+1}t_{n_{k-1}+1}, t_{n_k}s_{n_k}) \\
   d_{L^*}(b, t_{n_{k-1}+1}s_{n_{k-1}+1}) + d_{L^*}(t_{n_{k-1}+1}s_{n_{k-1}+1}, t_{n_k}s_{n_k})
   \end{array} \right.$
   else (i.e., edges of subsequence are reversed):
   - $d_{L^*}(b, s_{n_k}t_{n_k}) \leftarrow \min \left\{ \begin{array}{l}
   d_{L^*}(b, s_{n_{k-1}+1}t_{n_{k-1}+1}) + d_{L^*}(s_{n_{k-1}+1}t_{n_{k-1}+1}, s_{n_k}t_{n_k}) \\
   d_{L^*}(b, t_{n_{k-1}+1}s_{n_{k-1}+1}) + d_{L^*}(t_{n_{k-1}+1}s_{n_{k-1}+1}, s_{n_k}t_{n_k})
   \end{array} \right.$
   - $d_{L^*}(b, t_{n_k}s_{n_k}) \leftarrow \min \left\{ \begin{array}{l}
   d_{L^*}(b, s_{n_{k-1}+1}t_{n_{k-1}+1}) + d_{L^*}(s_{n_{k-1}+1}t_{n_{k-1}+1}, t_{n_k}s_{n_k}) \\
   d_{L^*}(b, t_{n_{k-1}+1}s_{n_{k-1}+1}) + d_{L^*}(t_{n_{k-1}+1}s_{n_{k-1}+1}, t_{n_k}s_{n_k})
   \end{array} \right.$

2.2.3 If $(k < z AND n_k + 1 - m_k > 1) OR (k = z AND n_z < |S|)$:
   - $\text{label}(L^*, b, s_{n_k+1}t_{n_k+1})$, $\text{label}(L^*, b, t_{n_k+1}s_{n_k+1})$

2.3 If $n_z < |S|$
   - $d_{L^*}(b, e) \leftarrow \min \left\{ \begin{array}{l}
   d_{L^*}(b, s_{n_{z-1}+1}t_{n_{z-1}+1} + d_{L}(s_{n_{z-1}+1}t_{n_{z-1}+1}, t_{n_{z}+1}s_{n_{z}+1}) \\
   d_{L^*}(b, t_{n_{z}+1}s_{n_{z}+1} + d_{L}(t_{n_{z}+1}s_{n_{z}+1}, s_{n_{z}+1}t_{n_{z}+1})
   \end{array} \right.$
   else: $\text{label}(L^*, b, e)$.

3 Select the move for which $d_{L^*}(b, e)$ is a minimum and perform this move on $S$.

Note that Algorithm 2 does not apply any of the moves during step 2, but computes what the total route cost would be if they were to be applied. The sequence $S^*$ is therefore not seen as an existing sequence, but rather as one that may be inferred from $\text{TRANSFORM}(S \rightarrow S^*)$. 

The computational complexity of step 1 is $O(|V(L)|^2)$. The complexity of step 2 depends on two factors, the first of which is the number of moves evaluated and the second, the time taken to evaluate a move. Using Two–Opt and Three–Opt, for example, there are respectively $O(|S|^2)$ and $O(|S|^3)$ moves to be evaluated. The method takes $O(z)$ time to evaluate a move, and the complexity of step 3 is $O(|S|)$. Therefore, the worst–case computational complexity of using Two–Opt and Three–Opt in the pseudo–code algorithm shown, is respectively $O(|S|^2)$ and $O(|S|^3)$. The average–case execution time of the procedures are of the same orders of magnitude.

Finally, although it was stated earlier that TRANSFORM($S \rightarrow S^*$) describes only edges that have their position changed in the transformed solution, in one particular case an edge that does not have its positions changed may be included in the notation to simplify the execution of the complexity reduction method. Consider the Two–Opt move TRANSFORM($S^5 \rightarrow S^{5*}$) = $\langle \langle (5, 1), (4, 2) \rangle, \langle (2, 4), (1, 5) \rangle \rangle$ performed on a hypothetical solution sequence, $S^5$. This move could also be encoded as TRANSFORM($S^5 \rightarrow S^{5*}$) = $\langle \langle (5, 1), (4, 2), (3, 3), (2, 4), (1, 5) \rangle \rangle$. The inclusion of the pair (3, 3) does not influence the result of the complexity reduction method, but improves its execution time slightly, because the method evaluates fewer subsequences. Therefore, in a move where a sequence consisting of an odd number of edges is reversed, without otherwise moving their positions in the sequence, it is more efficient to include, in the notation TRANSFORM($S \rightarrow S^*$), the edge that does not have its position changed. Nevertheless, in a computer implementation, this is not a practical concern, because a move would not be encoded as in TRANSFORM($S \rightarrow S^*$) during execution, since this would take $O(|S|)$ time to encode. Rather, the computer program would keep track of the untransformed and transformed positions of the endpoints of each subsequence, and in doing so, would automatically ensure this benefit. The operation of the complexity reduction method is now illustrated by means of example.

**Figure 5:** An example RPP problem instance. Edge cost weights are shown next to each edge, and required edges are denoted in bold face.

**Example 1** The complexity reduction method is demonstrated in this example by applying it to the graph in Figure 5, in which the required edges are denoted in bold face. Assume that an initial solution to this instance has been determined, and is given by
$S_0 = \{(1,3),(4,8),(8,5),(6,9),(10,6),(2,1)\}.$

The corresponding layered graph, denoted $L_0$, is depicted in Figure 6. Its vertex set is given by $V(L_0) = \{1, \ldots, 2 \times 6 + 2 = 14\}$. The cost of $S_0$ may be determined by adding the sum of the cost weights of the required edges (i.e., the value 11) to $d_{L_0}(1,14) = 15$ to yield a total cost of 26.

The evaluation of a single move by the complexity reduction method is described next. The move under consideration is the Two–Opt move $\text{TRANSFORM}(S_0 \to S_0^*) = \langle \langle 5,2 \rangle, \langle 4,3 \rangle, \langle 3,4 \rangle, \langle 2,5 \rangle \rangle$, in which the sequence of edges between the first and the last edges in $S_0$ are reversed. The resulting layered graph, as it would appear if the move were applied, is shown in Figure 7. Note that the edges of this graph are omitted for the sake of clarity.

![Figure 6: The layered graph, $L_0$, corresponding to the initial solution sequence $S_0 = \{(1,3),(4,8),(8,5),(6,9),(10,6),(2,1)\}$ of the RPP instance in Example 1.](image)

The relevant cost labels required to compute the route weight are shown on the graph, as computed by the complexity reduction method. The computations performed in step 2 of the complexity reduction method proceed as follows:

2.1 Set $d_{L_0^*}(1,2) \leftarrow d_{L_0}(1,2) = 0$, $d_{L_0^*}(1,3) \leftarrow d_{L_0}(1,3) = 1$.

2.2 For the only subsequence $k \leftarrow 1$ in $\text{TRANSFORM}(S_0)$:

2.2.1 label($L_0^*,1,4$), label($L_0^*,1,5$) (yielding $d_{L_0^*}(1,4) = 3$, $d_{L_0^*}(1,5) = 2$).

2.2.2 $n_{l_1} = 5 > 2 = n_{l_0+1}$, therefore $d_{L_0^*}(1,10)$ and $d_{L_0^*}(1,11)$ are computed in this step. $o_{l_1} = 2 < 3 = o_{l_1-1}$ (i.e. edges of subsequence reversed):

$$d_{L_0^*}(1,10) \leftarrow \min \begin{cases} d_{L_0^*}(1,4) + d_{L_0}(5,11) = 3 + 10 \end{cases} = 11$$

$$d_{L_0^*}(1,11) \leftarrow \min \begin{cases} d_{L_0^*}(1,4) + d_{L_0}(4,11) = 3 + 8 \end{cases} = 9.$$
2.2.3 \( k = z = 1 \) AND \( n_L = 5 \), therefore:
\[
\text{label}(\mathcal{L}_0^*, 1, 12), \text{label}(\mathcal{L}_0^*, 1, 13) \quad (\text{yielding } d_{\mathcal{L}_0^*}(1, 12) = 13, d_{\mathcal{L}_0^*}(1, 13) = 11).
\]

2.3 \( n_L = 5 < 6 = |S| \), therefore:
\[
d_{\mathcal{L}_0^*}(1, 14) \leftarrow \min \left\{ d_{\mathcal{L}_0^*}(1, 12) + d_{\mathcal{L}_0^*}(12, 14) = 13 + 0, \quad d_{\mathcal{L}_0^*}(1, 13) + d_{\mathcal{L}_0^*}(13, 14) = 11 + 2 \right\} = 13.
\]

The total cost of the solution after performing the move would be 24 (i.e., the sum of the cost weights of the required edges, 11, added to \( d_{\mathcal{L}_0^*}(1, 14) = 13 \)). The example illustrates how the total cost of the route may be found without actually performing the move or computing an entire shortest path from vertex 1 to 14. Note also that the optimal traversal directions need not be redetermined when the complexity reduction method is used.

\[\begin{array}{c|c|c|c|c|c|c}
\text{edge} & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
\text{label} & 3 & 4 & 6 & 8 & 10 & 12 \\
\end{array}\]

\[\begin{array}{c|c|c|c|c|c|c}
\text{vertex} & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
\text{label} & 5 & 7 & 9 & 11 & 13 & 14 \\
\end{array}\]

**Figure 7:** The layered graph \( \mathcal{L}_0^* \) obtained by performing the move TRANSFORM(\( S_0 \rightarrow S_0^* \)) = \( \langle \langle (5, 2), (4, 3), (3, 4), (2, 5) \rangle \rangle \). The labels calculated by the complexity reduction procedure are shown as bold numbers beside the relevant vertices.

## 3 Computational Results for the RPP

The results obtained from applying the Two–Opt and Three–Opt heuristics (described in §2) to benchmark problems and new test data are presented in this section and contrasted with results previously obtained by others.

### 3.1 Details of Heuristics

A Two–Opt and Three–Opt procedures, described in §2.1, were used to arrive at the results presented in this section. The heuristic of Frederickson (1979), of complexity \( O(|V|^3) \), was used to generate starting solutions for the Two–Opt and Three–Opt procedures, and the method described in §2.3 was used to reduce the complexity of each iteration of Two–Opt procedure [Three–Opt procedure, respectively] to \( O(|E_r|^2) \) \( O(|E_r|^3) \), respectively. A one–dimensional array of integer values was used to represent the order in which the edges of the route are traversed, and the Two–Opt and Three–Opt procedures were performed on this array.
3.2 Benchmark Graph Instances

The results obtained from applying the Two–Opt and Three–Opt heuristics to a set of benchmark test instances are shown in Table 1. The instances numbered 1 to 24 in Table 1 are the instances used by Christofides et al. (1986), and the other two instances are two of the so–called Albaida instances used by Corberán and Sanchis (1991). Hertz et al. (1999) and Fernández de Córdoba et al. (1998) present results for the same data. Note that the execution times listed in the columns for the Two–Opt and Three–Opt procedures of Table 1 are the total execution time, including the time taken to generate an initial solution.

| Instance | | | | | | | | | | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
|          | | | | | | | | | | | | | | | | |
| 1        | 11 13 7  | 76       | 76 0     | 76 0     | 76 0     |
| 2        | 14 33 12 | 152      | 155 0    | 153 0    | 152 0    |
| 3        | 28 58 26 | 102      | 105 0    | 103 0    | 103 0    |
| 4        | 17 35 22 | 84       | 84 0     | 84 0     | 84 0     |
| 5        | 20 35 16 | 124      | 130 0    | 124 0    | 124 0    |
| 6        | 24 46 20 | 102      | 107 0    | 107 0    | 102 0    |
| 7        | 23 47 24 | 130      | 130 0    | 130 0    | 130 0    |
| 8        | 17 40 24 | 122      | 122 0    | 122 0    | 122 0    |
| 9        | 14 26 14 | 83       | 83 0     | 83 0     | 83 0     |
| 10       | 12 20 10 | 80       | 80 0     | 80 0     | 80 0     |
| 11       | 9 14 7   | 23       | 26 0     | 23 0     | 23 0     |
| 12       | 7 18 5   | 19       | 22 0     | 19 0     | 19 0     |
| 13       | 7 10 4   | 35       | 35 0     | 35 0     | 35 0     |
| 14       | 28 79 31 | 202      | 207 0    | 204 0    | 202 0    |
| 15       | 26 37 19 | 441      | 445 1    | 441 0    | 441 0    |
| 16       | 31 94 34 | 203      | 215 0    | 205 0    | 203 0    |
| 17       | 19 44 17 | 112      | 116 0    | 112 0    | 112 0    |
| 18       | 33 55 29 | 257      | 274 0    | 271 0    | 266 0    |
| 19       | 50 98 63 | 398      | 402 0    | 400 0    | 400 1    |
| 20       | 50 184 74| 366      | 372 0    | 372 0    | 372 0    |
| 21       | 50 158 78| 475      | 479 0    | 477 0    | 477 2    |
| 22       | 50 144 88| 8629     | 8629 0   | 8629 0   | 8629 1   |

Table 1: Results for RPP benchmark test instances. Execution times are measured in seconds and were achieved on an Intel Pentium IV processor (2.8 GHz) with 512 Mb memory.

Note also that instance 18 of the Christofides problems is omitted from this analysis due to inconsistencies in its reported optimal value. Hertz et al. (1999) report that the optimal solution for this graph instance has a cost weight of 147 and that a solution of this cost is found by their heuristics, while Fernández de Córdoba et al. (1998) report that it is 148 and that their procedure also finds a solution of this cost. However, the heuristics described in §2.1 and §2.3 found a solution of cost 146. A manual examination of the route confirms that a feasible solution of cost 146 exists.

Albaida is a Spanish town, on whose streets the graphs are based.
Frederickson’s heuristic found an optimal solution 9 times out of the 25 instances considered. When the Two–Opt procedure was also applied, this number increased to 15. When the Three–Opt procedure was used in conjunction with Frederickson’s heuristic instead, an optimal solution was found 19 times out of 25. The heuristic of Fernández de Córdoba et al. (1998) found an optimal solution 11 times out of 25, while the Two–Opt based heuristic of Hertz et al. (1999) found an optimal solution 22 times out of 25. However, the high computational complexity of the Hertz et al. (1999) Two–Opt heuristic ($O(|E|^5)$ per iteration) and the fact that actual running times closely match theoretical worst–case times in exchange–based moves (because all exchanges are considered), limits its applicability to small problem instances, in our opinion.

3.3 Graph Instances with Euclidean and Random Edge Weights

The graph instances of Table 1 are clearly well within the capabilities of all the heuristics considered, and two larger sets of data were therefore generated, in order to establish bounds of practical feasibility for the heuristics, in terms of problem instance sizes that could be considered. The results of applying the heuristics to these larger instances are shown in Table 2.

The instances of the first data set in Table 2(a) were generated with Euclidean edge weights\(^4\), and the instances of the data set in Table 2(b) with random edge weights\(^5\). The dimensions of the graphs were chosen uniformly in the ranges $500 \leq |V| \leq 1000$ and $2500 \leq |E| \leq 3500$, and the number of required edges in each graph (i.e., $|E_r|$) was set to approximately 10% of the size of the graph. The edges designated as \textit{required} were selected iteratively — a random as yet \textit{non–required} edge was chosen and then designated as \textit{required} only if the subgraph induced by the required edges would be disconnected if it were to be chosen. The process was repeated until the desired number of required edges had been selected. The graphs were generated according to the procedure outlined in Algorithm 3.

\begin{algorithm}
\textbf{Algorithm 3 (Generate Random Graph Instance)}
\begin{enumerate}
\item Initialise the graph $G$ to contain $p$ vertices (and no edges).
\item Select two different components of $G$ at random, and add an edge between a randomly chosen vertex from each component. Repeat this step until $G$ is connected.
\item While $G$ has fewer than $q$ edges, add an edge between two randomly chosen vertices not yet joined by an edge.
\item Output $G$.
\end{enumerate}
\end{algorithm}

The algorithm used to determine the lower bound values in Table 2 operates in a similar way to the well–known Edmonds & Johnson algorithm for the CPP. A minimum cost maximum cardinality matching is performed on the odd–degree vertices of the subgraph induced by the required edges (Edmonds and Johnson, 1973).

\(^4\)The vertices of each graph were each randomly assigned a position in a unit square (according to a uniform distribution) and the edge weights were set equal to the value obtained by multiplying the Euclidean distance between vertices by 1000 and rounding the result to the nearest integer.

\(^5\)The edge weights of the random instances were uniformly chosen integer values between 1 and 1000.
### Table 2: Test results for two new RPP data sets, one consisting of Euclidean edge weights, and the other with random edge weights. Execution times are measured in seconds and were achieved on an Intel Pentium IV processor (2.8 GHz) with 512 Mb memory.
Data Set | Frederickson | Two–Opt | Three–Opt
---|---|---|---
Euclidean | 5.9% | 3.8% | 3.0%
Random | 4.1% | 2.5% | 2.0%

Table 3: Average percentage–gap–over–lower bound values obtained by the RPP heuristics on the test problem instances of Table 2.

The performance of the heuristics, in relation to the lower bound values, are summarised in Table 3. The Two–Opt procedure is typically able to reduce the gap over lower bound values obtained by Frederickson’s heuristic by a further $1.5 - 2.0\%$. The Three–Opt procedure is able to reduce this gap to approximately $2 - 3\%$.

### 3.4 Large Problem Instances

Computational results for a set of large test problems are shown in Table 4. These results were obtained using the Two–Opt heuristic. The instances of this table were generated in the same manner as those in Table 2, and were assigned random Euclidean edge weights. The order and size of each graph were chosen uniformly from the ranges $3000 \leq |V| \leq 5000$ and $20000 \leq |E| \leq 30000$ respectively.

| Instance | $|V|$ | $|E|$ | $|E_r|$ | Lower bound | Frederickson | Two–Opt |
|---|---|---|---|---|---|---|
| 1 | 4462 | 27506 | 2751 | 1729797 | 1764908 | 3718 | 1752799 | 4339 |
| 2 | 3130 | 22342 | 2234 | 1363165 | 1374875 | 1270 | 1369920 | 1518 |
| 3 | 4555 | 24398 | 2440 | 1577885 | 1628666 | 3859 | 1610481 | 4622 |
| 4 | 4592 | 22492 | 2249 | 1502920 | 1564479 | 3945 | 1539635 | 4852 |
| 5 | 4740 | 28547 | 2855 | 1809672 | 1846532 | 4342 | 1833930 | 5079 |
| 6 | 4927 | 25007 | 2501 | 1632439 | 1692323 | 4836 | 1669227 | 5901 |
| 7 | 3260 | 29835 | 2984 | 1743054 | 1749300 | 1508 | 1747615 | 1641 |
| 8 | 4221 | 23490 | 2349 | 1497199 | 1536811 | 3014 | 1553667 | 3568 |
| 9 | 4519 | 20839 | 2084 | 1413368 | 1480931 | 3678 | 1458962 | 4288 |
| 10 | 3908 | 20660 | 2066 | 1326576 | 1373923 | 3365 | 1353635 | 4538 |

Table 4: Test results for an RPP data set consisting of large graphs. Execution times are measured in seconds and were achieved on an Intel Pentium IV processor (2.8 GHz) with 512 Mb memory.

The instances of Table 4 are the largest known RPP instances to which either a heuristic or exact method have been applied successfully. The largest graphs previously reported seem to be those of Ghiani and Laporte (2000), consisting of 350 vertices and an average of 1370 edges (of which approximately 140 are required edges).

### 4 Conclusion

New heuristics for the Rural Postman Problem (RPP) were introduced in this paper. The heuristics are based on a local search framework that can also be used to optimise routes in other types of arc routing problems (e.g., the Capacitated Arc Routing Problem), and seem to constitute the best heuristic approach currently available for the RPP.
Acknowledgements

The authors are grateful to Prof Angel Corberán of the University of Valencia for providing the benchmark RPP graph instances used in this paper. Work towards this paper was supported by the South African National Research Foundation under grant number GUN 2053755 and Research Sub-Committee B at the University of Stellenbosch.

References


Efficient heuristics for the Rural Postman Problem


Letchford AN, 1996, Polyhedral results for some constrained arc routing problems, PhD Dissertation, Lancaster University, Lancaster.


Letchford AN, 1996, Polyhedral results for some constrained arc routing problems, PhD Dissertation, Lancaster University, Lancaster.


Letchford AN, 1996, Polyhedral results for some constrained arc routing problems, PhD Dissertation, Lancaster University, Lancaster.


Letchford AN, 1996, Polyhedral results for some constrained arc routing problems, PhD Dissertation, Lancaster University, Lancaster.


