Investigating Edge-Reordering Procedures in a Tabu Search Algorithm for the Capacitated Arc Routing Problem

Wasin Padungwech, Jonathan Thompson, and Rhyd Lewis

School of Mathematics, Cardiff University, Cardiff, UK
{padungwechw,thompsonjm1,lewisr9}@cardiff.ac.uk

Abstract. This paper presents two ideas to guide a tabu search algorithm for the Capacitated Arc Routing Problem to a promising region of the solution space. Both ideas involve edge-reordering, although they work in different ways. One of them aims to directly tackle deadheading cycles, and the other tries to reorder edges with the aim of extending a scope of solutions that can be reached from a given solution. Experiments were performed on 134 benchmark instances of various sizes, and the two ideas were shown to have an ability to guide the search to good solutions. Possible issues that may arise when implementing these ideas are also discussed.

1 Introduction

The Capacitated Arc Routing Problem (CARP) is a combinatorial optimisation problem that can be defined as follows: Given a graph with one of its vertices called *the depot*, a cost and a demand for each edge, and a vehicle capacity, the objective of the CARP is to find a minimum-cost set of routes (one route for each vehicle) such that (i) each route contains the depot, (ii) all edges with non-zero demands (called *required edges*) are serviced in precisely one of the routes, and (iii) the total demand in each route does not exceed the capacity. The CARP can be used to model and solve various real-life situations such as rubbish collection, street sweeping, and snow ploughing. It was originally introduced and proved to be NP-hard by Golden and Wong [8].

A wide variety of algorithms have been proposed to solve the CARP, possibly as a result of its real-world applicability. Metaheuristics have been popular choices, and include guided local search [1], scatter search [9], variable neighbourhood search [13], ant colony optimisation [14], memetic algorithms [6, 15], and tabu search [2, 9, 10].

Despite a wide variety of proposed algorithms in the literature, there are still some CARP benchmark instances that remain unsolved, especially those with a relatively large number (347 to 375) of required edges.

This suggests that more efficient algorithms for the CARP are still to be found. One key idea could be to find a way to explore a space of solutions efficiently. Traditional neighbourhood moves for the CARP such as removing or inserting edges or swapping edges between routes usually affect only a small number of edges and leave other edges untouched (apart from perhaps shifting their orders). It could be beneficial to integrate such moves with a method that can extend the scope of solutions that can be reached from a given solution, thereby increasing the connectivity of the solution space. One possible way to achieve that is to allow edges in a route to be reordered when receiving a new edge from another route. This could better accommodate the new edge and lead to an improvement which might have otherwise required several traditional moves.

Note that allowing edges to be reordered can greatly enlarge a neighbourhood of a given solution. In this paper, we therefore present two ideas that can help a search head towards a promising region of the solution space. The first idea is based on an investigation into deadheading edges, i.e. edges that are not serviced by a vehicle but are used to travel from one serviced edge to another. A route usually contains not only serviced edges but also deadheading edges. If deadheading edges form a cycle, such cycle should be removed provided that the route does not get disconnected as a result. This potentially reorders the edges in the route while definitely improving a solution (assuming non-zero edge costs). This could be particularly useful for large instances with a small ratio of capacity to total demand, where vehicles tend to fill the capacity quickly and have to return to the depot early, resulting in a significant amount of deadheading cost.

The second idea is to reconstruct a route with a given set of required edges. This can be achieved by means of a heuristic algorithm for the Rural Postman Problem, a special case of the CARP in which a single vehicle has large enough capacity to service all edges (see, for example, [4]). This idea was also utilised by Brandão and Eglese in [2]. In their paper, the heuristic is applied to routes that are changed by the best neighbourhood move in each iteration. In this paper, by contrast, the heuristic is integrated with each neighbourhood move, so it is also taken into account when finding the best neighbourhood move.

This paper is organised as follows: A formal definition of the CARP is given in Section 2. Section 3 explains how deadheading cycles can occur in a route and introduces a procedure for removing such cycles. Section 4 describes a tabu search algorithm with edge-reordering procedures.

Performances of the algorithm are presented and discussed in Section 5. Finally, Section 6 gives a conclusion and suggestions for future work.

2 Problem Definition and Notation

Given an undirected graph $G = (V, E)$ with a vertex set V and an edge set E, a cost $c(e) \in \mathbb{Z}^+$ and a demand $d(e) \in \mathbb{Z}^+ \cup \{0\}$ for each edge $e \in E$, a vehicle capacity $Q \in \mathbb{Z}^+$, and one of the vertices $v_0 \in V$ regarded as the *depot*, the objective of the CARP is to find a minimum-cost set of routes such that

- each route contains the depot,
- each edge with non-zero demand (called a *required edge*) is serviced in precisely one of the routes, and
- the total demand of serviced edges in each route does not exceed the vehicle capacity.

Here, the number of routes is treated as a variable. Note that the orientation of each edge in a route needs to be specified even if the underlying graph is undirected because this can affect the cost of travelling from one required edge to another. An edge $e = \{i, j\}$ can be traversed in two possible ways: from i to j or from j to i, denoted by directed edges, or *arcs*, (i, j) and (j, i), respectively. A route R can then be represented as a sequence of arcs $(a_1, a_2, \ldots, a_n)$, where n is the number of arcs that are serviced by R and $a_1, a_2, \ldots, a_n$ are the serviced arcs. This is possible as it is clear that a vehicle should travel between serviced arcs via a shortest path in order to minimise the overall cost. This representation of a route is similar to that in [1].

Let $t(a)$ and $h(a)$ denote the tail and the head of an arc a. For example, if $a = (i, j)$, then $t(a) = i$ and $h(a) = j$. The total cost of the route $R = (a_1, a_2, \ldots, a_n)$ is given by

$$\mathcal{C}(R) = d(v_0, t(a_1)) + \sum_{i=1}^{n-1} d(h(a_i), t(a_{i+1})) + d(h(a_n), v_0) + \sum_{i=1}^{n} c(a_i) \ , \quad (1)$$

where $d(u, v)$ is the cost of a shortest path between vertices u and v. The total demand of R is given by

$$\mathcal{D}(R) = \sum_{i=1}^{n} d(a_i) \ . \quad (2)$$

Using the above notations, the CARP can be presented more formally. Let $x(e, R)$ be a binary variable such that $x(e, R) = 1$ if an edge e is serviced in a route R, and $x(e, R) = 0$ otherwise. Let E_R be the set of required edges. The objective of the CARP is to find a set of routes S that minimises

$$f(S) = \sum_{R \in S} \mathcal{C}(R) \tag{3}$$

while satisfying the following constraints:

$$\sum_{R \in S} x(e, R) = 1 \qquad \forall e \in E_R \ , \tag{4}$$

$$\mathcal{D}(R) \leq Q \qquad \forall R \in S \ . \tag{5}$$

The equality (4) means that each required edge must be serviced in precisely one of the routes, and the inequality (5) is the capacity constraint. Note that the CARP can also be formulated as an integer linear programming problem. Interested readers are referred to [4] or [8].

3 Deadheading Cycles

Even though a route can be represented by a sequence of serviced edges (with specified orientation), in reality a vehicle must travel in a continuous route, and so it may need to traverse some edges without servicing them when travelling between two required edges that are not physically adjacent. Such edges are called *deadheading edges*. In some cases, several deadheading edges may form a cycle, called a *deadheading cycle*. Indeed, provided it does not disconnect a route, a deadheading cycle can be removed without affecting feasibility of the route for two reasons: (i) the capacity constraint is still satisfied because serviced edges are unaffected (apart from potential reordering or reorientation), and (ii) with a route regarded as an Eulerian (multi)graph, removing a cycle preserves the parity of the degree of each vertex, so the Eulerian graph remains Eulerian after the removal.

It can be difficult to find a deadheading cycle while viewing a route as a sequence of serviced arcs because a deadheading cycle may be composed of deadheading edges which are traversed between different pairs of serviced arcs. So, for the purpose of detecting deadheading cycles, we view a route as an Eulerian graph (or multigraph if some edges are traversed more than once). Given a route $R = (a_1, a_2, \ldots, a_n)$, first we need to find shortest paths between v_0 and a_1, between a_i and a_{i+1} (for $i = 1, \ldots, n-1$), and

between a_n and v_0. This can be achieved by Dijkstra's algorithm [3]. Then, let G_{mult} be a multigraph such that the multiplicity of each edge in G_{mult} is equal to the number of times the edge is traversed (in any direction, with or without servicing) in R. When an edge is traversed three times or more, at least two such traversals are deadheading because, by definition, an edge can be serviced at most once. Every two deadheading traversals on the same edge correspond to a cycle in G_{mult}, which can be removed without disconnecting it as long as the number of traversals does not drop below 2. For an edge that is traversed just twice, careful consideration is needed before removing the corresponding cycle. In Fig. 1 for example, we can see that removing a deadheading cycle may or may not disconnect a route. Thus, to ensure continuity of a route, here we remove a deadheading cycle until the multiplicity of the corresponding edge reduces to either 1 or 2, depending on its parity.

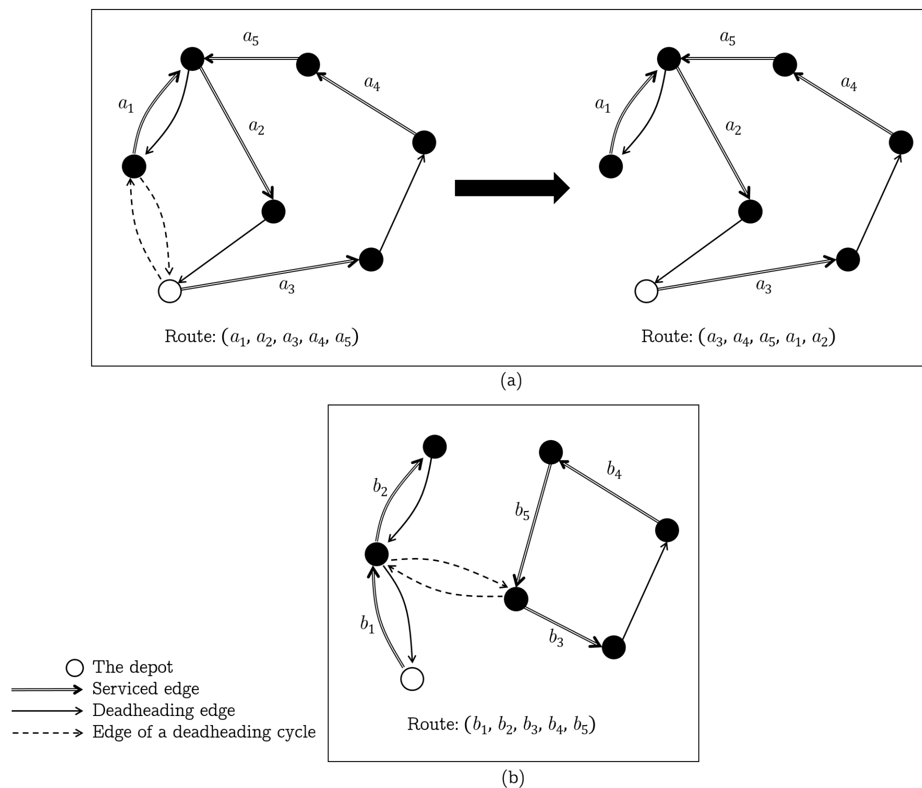

Fig. 1. Examples of deadheading cycles that are (a) removable and (b) not removable.

After removing deadheading cycles, an updated sequence of edges traversed in the route R can be determined by finding an Eulerian cycle in the consequent G_{mult}. In order to obtain a representation of R as described in Section 2, if an edge to be serviced is traversed more than once, it is assumed that the edge is serviced at its first occurrence in the Eulerian cycle. Notice that the ordering and orientation of some serviced edges may change after the removal—see Fig. 1 (a).

4 Description of the Tabu Search Algorithm

In this section, components of our tabu search algorithm are described. Our algorithm starts with an initial solution generated by the Path-Scanning algorithm [7]. This algorithm has been shown to produce better solutions on average than several other constructive algorithms [2].

4.1 Neighbourhood Moves

Recall that a route is viewed as a sequence of required edges with specified directions that are serviced by the route. Four common neighbourhood moves are used:

1. *Single Insertion*: A required edge is removed from one route and then inserted at any position into another route. Both directions are examined when inserting it into a route.
2. *Double Insertion*: Two required edges are removed from one route and then inserted at any positions into another route. The edges may be inserted at the same position (i.e. between the same required edges, or between the depot and the first or last required edge). In the case where the edges are inserted at the same position, both possible permutations of the edges are considered ("edge 1 before edge 2" and "edge 2 before edge 1"). Both directions of each edge are examined when inserting it into a route. Thus, there are 8 possibilities of inserting two edges at a given position.
3. *Swap*: A required edge from each of two given routes is removed and inserted at any position into the other route, including the position of the removed edge. Both directions of each edge are examined when inserting it into a route.
4. *Two-Opt*: Each of two given routes is divided into two parts. Note that each part must have at least one required edge. Then, a part from one route is joined to a part from the other to create a new route. The remaining parts are also joined to create another route. Two possible ways of joining are examined. For clarity, this is illustrated in Fig. 2.

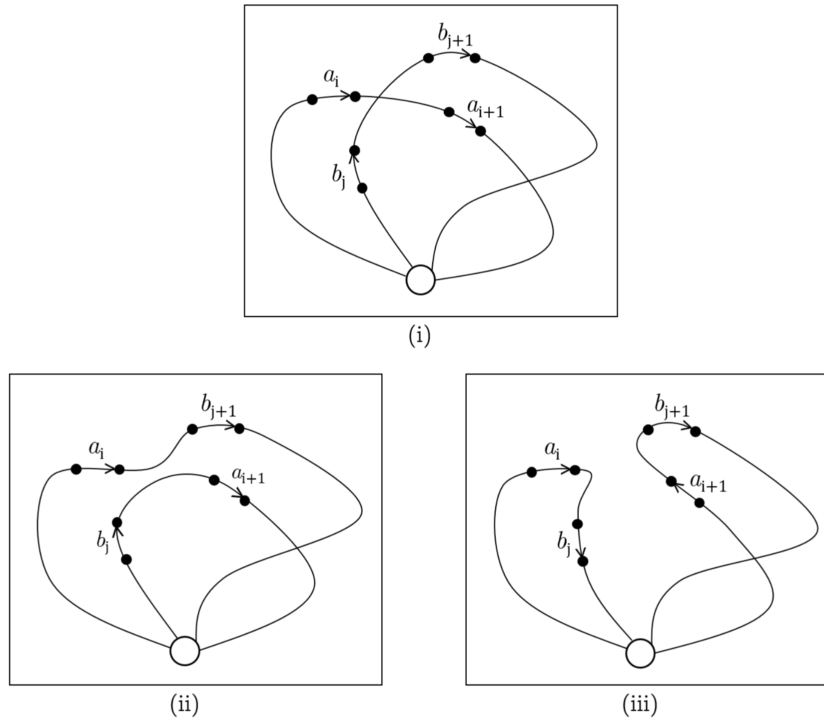

Fig. 2. An example of a Two-Opt move. In (i), after one route is cut between a_i and a_{i+1} and the other route between b_j and b_{j+1}, they can become either (ii) or (iii).

The first three neighbourhood moves were used in, for example, [2, 9, 12, 15], and the Two-Opt move was utilised in, for example, [1, 12, 15].

4.2 Rural Postman Heuristic

When a Single Insertion, Double Insertion, or Swap move is implemented, an edge is inserted into a route without affecting the order of required edges already in the route. However, it is possible that reordering some of those edges might better accommodate the new edge and potentially lead to a better solution. Consider Single Insertion for example. Assume the move is feasible. Instead of specifying where to insert an edge into a route R, we may simply add it to the set of edges serviced by R and then attempt to construct a "promising" route that services this set of edges without having to keep the original order of any of the edges. In fact, we are attempting to solve a special (though still NP-hard) case of the CARP, namely the Rural Postman Problem (RPP), where the capacity is no less than the sum of required edges under consideration.

In our case, this reordering is achieved by means of a heuristic for the RPP proposed by Frederickson [5]. Given a set of required edges E_R in an underlying graph G, let G_R be a subgraph of G generated by E_R. The heuristic consists of two main steps:

1. *Connecting components*: If G_R has more than one connected component, they will be joined to make one connected component. This is achieved by solving the minimum spanning tree problem: Let G_S be a complete graph having as many vertices as the components of G_R. Let the cost of an edge $\{i, j\}$ in G_S be equal to the shortest distance between components C_i and C_j, that is,

$$\text{the cost of an edge } \{i, j\} \text{ in } G_S = \min_{u \in C_i, v \in C_j} d(u, v) \ ,$$

   where $d(u, v)$ is the cost of the shortest path in G between vertices u and v. Let T be a minimum spanning tree in G_S. Add to G_R the shortest path corresponding to each edge in T. Now G_R is connected.
2. *Matching odd-degree vertices*: If G_R has odd-degree vertices, paths will be added to G_R to render the graph Eulerian. To achieve this, let G_M be a complete graph whose vertices are precisely the odd-degree vertices of G_R. Let the cost of an edge $\{i, j\}$ in G_M be equal to $d(i, j)$. Let M be a perfect matching in G_M. Add to G_R the shortest path corresponding to each edge in M.

Note that finding a minimum-cost perfect matching in Step 2 does not guarantee an optimal solution for the RPP. In fact, Brandão and Eglese [2] have noted that using a minimum-cost perfect matching could give a worse solution in some cases. Instead of using an exact approach (such as the blossom algorithm [11]), we therefore opt to use a cheaper greedy method which operates by selecting the cheapest edge which is not adjacent to any previously selected edges.

Now G_R is an Eulerian (multi)graph. A new route corresponding to G_R can be obtained in the same way as we did for G_{mult} at the end of Section 3.

4.3 Tabu Record and Tabu Tenure

In our case, information on tabu moves is stored in arrays (as opposed to lists). The Single Insertion, Double Insertion and Swap moves share the same tabu array T_1. An entry $T_1(e, R)$ denotes the iteration number until which the insertion of the edge e into the route R is declared tabu (i.e. forbidden). Whenever a required edge e is removed from a route R, the entry

$T_1(e, R)$ is updated so that $T_1(e, R)$ is set to the current iteration number plus the tabu tenure. Note that Double Insertion and Swap involve two insertions, so two entries in T_1 are updated. A Single Insertion/Double Insertion/Swap move is tabu if and only if all the insertions involved in the move are tabu. Note that the above procedure for updating T_1 still applies no matter whether the RPP heuristic is used because T_1 does not concern the position of an edge in a route.

Two-Opt uses a separate tabu array T_2. An entry $T_2(a, b)$ denotes the iteration number until which a cut (or, equivalently, a deadheading path) between required arcs a and b is declared tabu. (Recall that an arc is a directed edge.) Suppose a Two-Opt move involves a cut between required arcs a and b. Then, the entry $T_2(a, b)$ is updated so that $T_2(a, b)$ is set to the current iteration number plus the tabu tenure. It should be noted that a pair of arcs here must be treated as an ordered pair because different orders may correspond to different deadheading paths (see Fig. 3). In contrast, a pair (a, b) should be treated as identical to a pair $(-b, -a)$, where a minus sign denotes the opposite direction, because of symmetry of a route (as an undirected cycle in a graph). This can help save memory required for this tabu record.

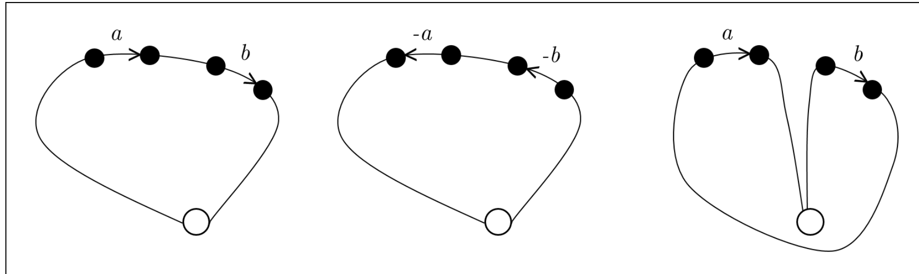

Fig. 3. Routes that contain the same required edges may be the same or different, depending on the direction and the order in which they are traversed.

For a tabu tenure, we follow a policy used in a previous tabu search algorithm in the literature [2]: the tenure is set to $n/2$, where n is the number of required edges. This remains fixed throughout the algorithm.

4.4 Admissibility of Moves

A common aspiration criterion is used here: In a given iteration, a neighbourhood move is considered if and only if it is feasible (the capacity

constraint is satisfied) and either (1) it is non-tabu or (2) it is tabu but leads to a better solution than the current best solution.

5 Computational Results

The tabu search algorithm described above was coded using C++ and all experiments were performed on Intel Core i3-2120 3.30GHz with 8GB RAM using benchmark datasets EGL, BMCV, and EGL-Large.[1] To simulate a real-life scenario where speed is preferable to optimality, a time limit of 300 seconds was introduced for the EGL and BMCV sets. Due to a larger number of required edges, a longer time limit of 1,200 seconds was introduced for the EGL-Large set.

To investigate how the edge-reordering procedures might help to improve the search, three versions of the algorithms were tested: the first one does not consider deadheading cycles nor the RPP heuristic ("No Reordering"), and each of the other two implements one of the procedures:

1. attempting to remove deadheading cycles from an initial solution and from routes that are affected by a neighbourhood move in each iteration ("RDC"),
2. integrating the RPP heuristic within the neighbourhood moves ("RPP").

For each version, 20 independent runs were conducted on each instance.

Tables 1, 2 and 3 present the features of the instances together with the mean and the coefficient of variation of the best solution costs from 20 independent runs of each version of our algorithm as described above. Due to a large number (100) of instances in the BMCV dataset, each row of Table 2 shows the average results on a subset of 25 instances

Tables 1 and 2 suggest that both procedures improve the quality of solutions produced compared to the "No Reordering" version (here, the ** symbol indicates statistical significance according to a Wilcoxon Signed Rank test with $p < 0.01$). However, for the EGL-Large dataset in particular, their performances are noticeably different. Table 3 shows that attempting to remove deadheading cycles can improve a solution on all EGL-Large instances, while the RPP heuristic does not seem to give any improvement. This is very likely because the RPP heuristic requires a long computational time for large instances, as can be seen in Table 4.

Table 4 shows the mean and coefficient of variation of the time taken by each edge-reordering procedure in the respective version of the algorithm. As no optimal solutions are known for this dataset, the algorithm

[1] These datasets, as well as best known solutions, are available at http://logistik.bwl.uni-mainz.de/benchmarks.php.

Table 1. The mean and the coefficient of variation (CV) of solution costs for 20 independent runs on the EGL dataset

Instance	$\|V\|$	$\|E\|$	$\|E_R\|$	Best known solution	No Reordering		RDC		RPP	
					Mean	CV	Mean	CV	Mean	CV
E1-A	77	51	98	3548	3548.0	0.0%	3548.0	0.0%	3548.0	0.0%
E1-B	77	51	98	4498	4531.9	0.3%	4525.4	0.2%	4529.3	0.2%
E1-C	77	51	98	5595	5748.3	1.2%	5736.3	0.9%	5634.1	0.9%
E2-A	77	72	98	5018	5149.4	0.6%	5059.8	0.4%	5028.1	0.2%
E2-B	77	72	98	6317	6347.5	0.1%	6345.8	0.1%	6339.3	0.1%
E2-C	77	72	98	8335	8620.4	1.7%	8683.6	1.8%	8559.9	1.3%
E3-A	77	87	98	5898	5916.1	0.3%	5918.1	0.5%	5937.4	0.7%
E3-B	77	87	98	7775	8015.3	1.3%	7912.9	1.2%	7915.8	1.0%
E3-C	77	87	98	10292	10392.1	0.6%	10398.2	0.4%	10371.4	0.3%
E4-A	77	98	98	6444	6521.9	0.6%	6492.8	0.4%	6480.6	0.3%
E4-B	77	98	98	8961	9135.3	1.1%	9086.0	0.7%	9029.7	0.3%
E4-C	77	98	98	11550	11800.7	0.6%	11795.6	0.6%	11775.6	0.3%
S1-A	140	75	190	5018	5132.7	0.9%	5069.4	1.0%	5077.5	1.1%
S1-B	140	75	190	6388	6503.1	0.6%	6480.1	1.0%	6511.7	0.7%
S1-C	140	75	190	8518	8662.8	0.9%	8646.9	0.7%	8623.6	0.8%
S2-A	140	147	190	9884	10041.5	1.0%	10089.1	1.0%	10152.7	0.9%
S2-B	140	147	190	13100	13531.4	1.1%	13479.0	1.4%	13366.8	0.9%
S2-C	140	147	190	16425	16865.3	0.6%	16888.1	0.7%	16773.9	0.7%
S3-A	140	159	190	10220	10414.8	0.8%	10335.8	0.3%	10460.9	0.9%
S3-B	140	159	190	13682	14126.6	1.5%	13935.6	0.4%	13959.0	0.8%
S3-C	140	159	190	17188	17634.8	0.7%	17490.8	0.5%	17456.6	0.4%
S4-A	140	190	190	12268	12527.3	0.7%	12555.5	0.7%	12775.3	0.5%
S4-B	140	190	190	16283	16613.7	0.7%	16534.9	0.5%	16641.5	0.5%
S4-C	140	190	190	20481	21012.5	0.5%	20980.8	0.7%	21059.6	0.4%
Average				9736.9	9949.7		9916.2**		9917.0	

Table 2. The mean and the coefficient of variation (CV) of solution costs for 20 independent runs on the BMCV dataset

Subset of instances	Average over 25 instances in the subset						
	Best known Solution	No Reordering		RDC		RPP	
		Mean	CV	Mean	CV	Mean	CV
C (C1 - C25)	3683.4	3781.1	1.5%	3760.9	1.2%	3747.1	1.1%
D (D1 - D25)	2872.4	2936.4	1.0%	2896.8	0.9%	2892.7	0.8%
E (E1 - E25)	3698.0	3795.9	1.5%	3782.0	1.6%	3774.8	1.2%
F (F1 - F25)	3003.0	3058.4	0.9%	3035.8	1.0%	3026.7	0.7%
Average	3314.2	3392.9		3368.9**		3360.3**	

Table 3. The mean and the coefficient of variation (CV) of solution costs for 20 independent runs on the EGL-Large dataset

Instance	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert E_R\rvert$	Best known solution	No Reordering		RDC		RPP	
					Mean	CV	Mean	CV	Mean	CV
G1-A	255	347	375	1004864	1025967.0	0.6%	1024128.5	0.6%	1046700.8	0.8%
G1-B	255	347	375	1129937	1135307.6	0.4%	1134844.3	0.4%	1154564.5	0.7%
G1-C	255	347	375	1262888	1282483.9	0.6%	1279901.8	0.7%	1303527.5	0.5%
G1-D	255	347	375	1398958	1410589.4	0.7%	1408335.1	0.6%	1423174.4	0.5%
G1-E	255	347	375	1543804	1551973.2	0.9%	1550111.0	0.8%	1581999.2	0.7%
G2-A	255	375	375	1115339	1120506.3	0.5%	1120201.9	0.6%	1142626.4	0.6%
G2-B	255	375	375	1226645	1237201.4	0.7%	1235783.5	0.7%	1281057.4	0.5%
G2-C	255	375	375	1371004	1381432.9	0.6%	1378903.3	0.7%	1426369.6	0.8%
G2-D	255	375	375	1509990	1515175.8	0.4%	1511281.5	0.4%	1557018.8	0.8%
G2-E	255	375	375	1659217	1667802.9	0.6%	1663952.9	0.6%	1696421.8	0.5%
Average				1322264.6	1332844.0		1330744.3**		1361346.0**	

Table 4. The mean and the coefficient of variation (CV) of the time taken (seconds) by the edge-reordering procedures for 20 independent runs on the EGL-Large dataset

Instance	RDC		RPP	
	Mean	CV	Mean	CV
G1-A	12.8	2.6%	1171.9	0.2%
G1-B	17.2	2.5%	1165.7	0.1%
G1-C	19.4	2.4%	1161.0	0.1%
G1-D	20.7	3.2%	1152.6	0.2%
G1-E	22.0	1.5%	1139.2	0.2%
G2-A	14.4	1.9%	1164.9	0.1%
G2-B	15.3	4.2%	1165.9	0.1%
G2-C	17.6	4.3%	1158.2	0.2%
G2-D	18.7	3.6%	1149.4	0.5%
G2-E	19.0	1.9%	1143.1	0.5%
Average	17.7		1157.2	

halts once the time limit of 1,200 seconds has elapsed. Table 4 shows that the RPP heuristic uses the vast majority of computation time (about 96%). This could be because the RPP heuristic is integrated with all Single Insertion, Double Insertion and Swap moves, comprising a huge set of neighbour solutions to consider with the heuristic. Using a large amount of time to explore a neighbourhood means this version of the algorithm performed considerably fewer iterations. As a result, the search may have not moved "far" from the initial solution in the solution space.

It is clear from Tables 1, 2 and 3 that solutions from all versions of the algorithm in this work are still far from the best known solutions. Nevertheless, we have seen that re-ordering edges has potential to guide the search to a promising region of the solution space and obtain better solutions within the same amount of time.

6 Conclusion and Future Work

This paper brought to attention two ideas to help guide a tabu search or, in fact, any local search method to a promising region of the solution space for the Capacitated Arc Routing Problem. The first idea is to investigate deadheading cycles and attempt to remove them after generating an initial solution and when they appear as a result of neighbourhood moves. Removing deadheading cycles guarantees an improvement, provided all edge costs are non-zero. Nevertheless, it is important to note that some deadheading cycles might not be removable as doing so may disconnect a route.

One way to ensure the continuity of a route is to remove a deadheading cycle only if the multiplicity of the corresponding edge does not drop below 2. However, we might try removing each possible deadheading cycle and directly checking if the route still remains connected. This allows us to detect more removable deadheading cycles, but it is important to find an efficient algorithm to do so.

Moreover, this work considered only deadheading cycles that result from a single edge traversed repeatedly. There can also be a cycle composed of several deadheading edges that are traversed precisely once. Still, an efficient algorithm is required for detecting the removability of such cycle.

This paper also investigated a combination of a heuristic for the Rural Postman problem with neighbourhood moves. This allows edges in a route to be re-ordered and potentially gives a better solution that might normally require several traditional neighbourhood moves. This can in-

crease connectivity of the solution space and, given excess time, increase the probability of reaching good solutions.

Experimental results showed that both ideas have potential to improve a search. However, it can be time-consuming to try the Rural Postman heuristic with all Single Insertion, Double Insertion, and Swap moves. It would therefore be interesting to find a balance between increasing connectivity of the solution space and taking time to evaluate all neighbours. Moreover, these two ideas may not be effective alone as can be seen from a comparison between solutions from the algorithm in this work and best known solutions. One may try to combine these two ideas together rather than use them separately, or even combine them with traditional tabu search techniques such as intensification and diversification. Such a good combination is still to be researched.

References

1. Beullens, P., Muyldermans, L., Cattrysse, D., Van Oudheusden, D.: A guided local search heuristic for the capacitated arc routing problem. European Journal of Operational Research 147(3), 629–643 (2003)
2. Brandão, J., Eglese, R.: A deterministic tabu search algorithm for the capacitated arc routing problem. Computers & Operations Research 35(4), 1112–1126 (2008)
3. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische mathematik 1(1), 269–271 (1959)
4. Eiselt, H.A., Gendreau, M., Laporte, G.: Arc routing problems, part ii: The rural postman problem. Operations research 43(3), 399–414 (1995)
5. Frederickson, G.N.: Approximation algorithms for some postman problems. Journal of the ACM (JACM) 26(3), 538–554 (1979)
6. Fu, H., Mei, Y., Tang, K., Zhu, Y.: Memetic algorithm with heuristic candidate list strategy for capacitated arc routing problem. In: Evolutionary Computation (CEC), 2010 IEEE Congress on. pp. 1–8. IEEE (2010)
7. Golden, B.L., DeArmon, J.S., Baker, E.K.: Computational experiments with algorithms for a class of routing problems. Computers & Operations Research 10(1), 47–59 (1983)
8. Golden, B.L., Wong, R.T.: Capacitated arc routing problems. Networks 11(3), 305–315 (1981)
9. Greistorfer, P.: A tabu scatter search metaheuristic for the arc routing problem. Computers & Industrial Engineering 44(2), 249–266 (2003)
10. Hertz, A., Laporte, G., Mittaz, M.: A tabu search heuristic for the capacitated arc routing problem. Operations research 48(1), 129–135 (2000)
11. Kolmogorov, V.: Blossom v: a new implementation of a minimum cost perfect matching algorithm. Mathematical Programming Computation 1(1), 43–67 (2009)
12. Lacomme, P., Prins, C., Ramdane-Cherif, W.: Competitive memetic algorithms for arc routing problems. Annals of Operations Research 131(1-4), 159–185 (2004)
13. Polacek, M., Doerner, K.F., Hartl, R.F., Maniezzo, V.: A variable neighborhood search for the capacitated arc routing problem with intermediate facilities. Journal of Heuristics 14(5), 405–423 (2008)

14. Santos, L., Coutinho-Rodrigues, J., Current, J.R.: An improved ant colony optimization based algorithm for the capacitated arc routing problem. Transportation Research Part B: Methodological 44(2), 246–266 (2010)
15. Tang, K., Mei, Y., Yao, X.: Memetic algorithm with extended neighborhood search for capacitated arc routing problems. Evolutionary Computation, IEEE Transactions on 13(5), 1151–1166 (2009)