Math. Prog. Comp. (2010) 2:79–101
DOI 10.1007/s12532-010-0013-5

FULL LENGTH PAPER

A library of local search heuristics for the vehicle routing problem

Chris Groër · Bruce Golden · Edward Wasil

Received: 31 December 2008 / Accepted: 16 March 2010 / Published online: 8 April 2010

Abstract The vehicle routing problem (VRP) is a difficult and well-studied combinatorial optimization problem. Real-world instances of the VRP can contain hundreds and even thousands of customer locations and can involve many complicating constraints, necessitating the use of heuristic methods. We present a software library of local search heuristics that allows one to quickly generate solutions to VRP instances. The code has a logical, object-oriented design and uses efficient data structures to store and modify solutions. The core of the library is the implementation of seven local search operators that share a similar interface and are designed to be extended to handle additional options with minimal code change. The code is well-documented, straightforward to compile, and is freely available online. The code contains several applications that can be used to generate solutions to the capacitated VRP. Computational results indicate that these applications are able to generate solutions that are within about one percent of the best-known solution on benchmark problems.

Edward Wasil was supported in part by a Kogod Research Professorship at American University.

C. Groër (✉)
Oak Ridge National Laboratory, 1 Bethel Valley Rd, Oak Ridge, TN 37831, USA
e-mail: cgroer@gmail.com

B. Golden
R.H. Smith School of Business, University of Maryland, College Park, MD 20742, USA
e-mail: bgolden@rhsmith.umd.edu

E. Wasil
Kogod School of Business, American University, Washington, DC 20016, USA
e-mail: ewasil@american.edu

Keywords Vehicle routing · Optimization · Heuristics · Metaheuristics

1 Introduction

In the classical capacitated Vehicle Routing Problem (VRP), a minimum cost set of routes is constructed for a fleet of identical vehicles. These routes must satisfy the demands of all customers, and the vehicles traversing these routes have a fixed capacity. The VRP is a well-known *NP*-hard problem and computational experience indicates that the VRP is difficult to solve to optimality. Most real-world vehicle routing problems are solved using heuristic methods.

There are several open source software packages for generating solutions to the traveling salesman problem (TSP), some of which are listed in [25]. We describe two of the best-known options here. The freely available and widely used *Concorde* package is an exact solver for the TSP that has been used to produce optimal solutions for problems with nearly 100,000 nodes [1,2]. The C source code for *Concorde* was developed by a team of seven authors and can be downloaded for academic research. A second open source code for solving the TSP is the Lin-Kernighan-Helsgaun (*LKH*) package developed by Helsgaun [17,18]. The author provides an optimized implementation of the original Lin-Kernighan heuristic [24] and enhances it in several ways. Although the *LKH* code does not implement exact methods, it has generated optimal tours to problems with up to 85,000 nodes. *LKH* is responsible for the current best-known solution to the World TSP with nearly 2 million nodes. The *LKH* code is freely available for academic purposes.

We are aware of only two freely available, non-commercial software packages for generating solutions to VRP instances. Both of these are focused on generating provably optimal solutions to smaller instances of the VRP. The first is the VRP code of Ralphs et al. [33,35] that is included with the distribution of the open source mixed-integer programming package *SYMPHONY* [34]. This VRP solver uses the branch, cut, and price capabilities of the *SYMPHONY* solver and is also able to interface with the *Concorde* TSP solver to add additional cuts to the formulation. The second open source package is the *CVRPSEP* software of Lysgaard [26]. Given a fractional solution to a linear programming formulation of the VRP, this software is designed to generate valid inequalities that are violated by the fractional solution. This software was used as part of the recent effort to generate optimal solutions to VRP instances containing as many as 135 nodes [9].

In this paper, we present an open source software library of heuristics for generating solutions to instances of the VRP. Our library of Vehicle Routing Problem Heuristics (called *VRPH*) is written in C/C++ and uses efficient data structures to implement methods for constructing an initial feasible solution and several local search heuristics that can be used to improve solutions. This paper is organized as follows. We begin by providing a review of VRP heuristic algorithms from the literature that use local search methods. We then discuss the design of the *VRPH* library and its data structures, and describe various functions contained in the library. We describe several applications

that we have built using the library and conclude by providing computational results for standard benchmark problems. *VRPH* is available for download at [15].

2 Local search for the VRP

Local search is a commonly used technique in combinatorial optimization. We are given a general minimization problem,

$$\min f(x) \text{ subject to } x \in \mathcal{S},$$

where $\mathcal{S}$ is a discrete solution space and $f : \mathcal{S} \mapsto \mathbb{R}$ is the objective function. Given a feasible solution $x \in \mathcal{S}$, we construct a neighborhood of feasible solutions, $N(x) \subset \mathcal{S}$, and then search this neighborhood for a new solution $x' \in N(x)$. When selecting x', we may require $f(x') < f(x)$ if we wish to consider only improving moves, or we may use a probabilistic acceptance strategy such as simulated annealing [20] where we may accept the new solution when $f(x') > f(x)$. More complicated metaheuristics such as tabu search [11] can be seen as a kind of local search where the acceptance of a new solution $x' \in N(x)$ depends on short- and long-term memories that are developed during the search.

Local search has proven to be an effective technique for generating good solutions to instances of the classical VRP as well as variants that involve additional constraints such as time windows and multiple depots. One typically defines a search neighborhood that consists of all feasible solutions that can be obtained from an existing solution through well-defined operations such as relocating a customer to a new position in a route or by removing edges from the solution and replacing them with new edges. *VRPH* contains implementations of seven different local search operators that are illustrated and described in Fig. 1.

There are dozens of successful algorithms for the VRP and its variants that incorporate some form of local search. We discuss a few of the better-known algorithms that employ local search, but our discussion is not intended to be exhaustive. The reader should consult the recent literature reviews [3,4,10] for additional examples of successful local search implementations.

The tabu search algorithm described in [38] applies a general local search operator that exchanges the positions of μ customers from one route with π customers from another. Due to the exponential growth of the resulting search neighborhood, μ and π are generally less than three. Note that the one-point, two-point, three-point, and two-opt moves are special cases of this operator. These local search operators were used as part of the well-known adaptive memory algorithm of Rochat and Taillard [37] that produced the best solutions to many benchmark instances more than 15 years ago. More recently, the guided evolutionary algorithms for the VRP and VRP with Time Windows (VRPTW) of Mester and Bräysy [27,28] use six of the local search operators available in *VRPH*. When the algorithm becomes trapped in a local minimum, sets of edges are penalized and the search continues. These algorithms are responsible for

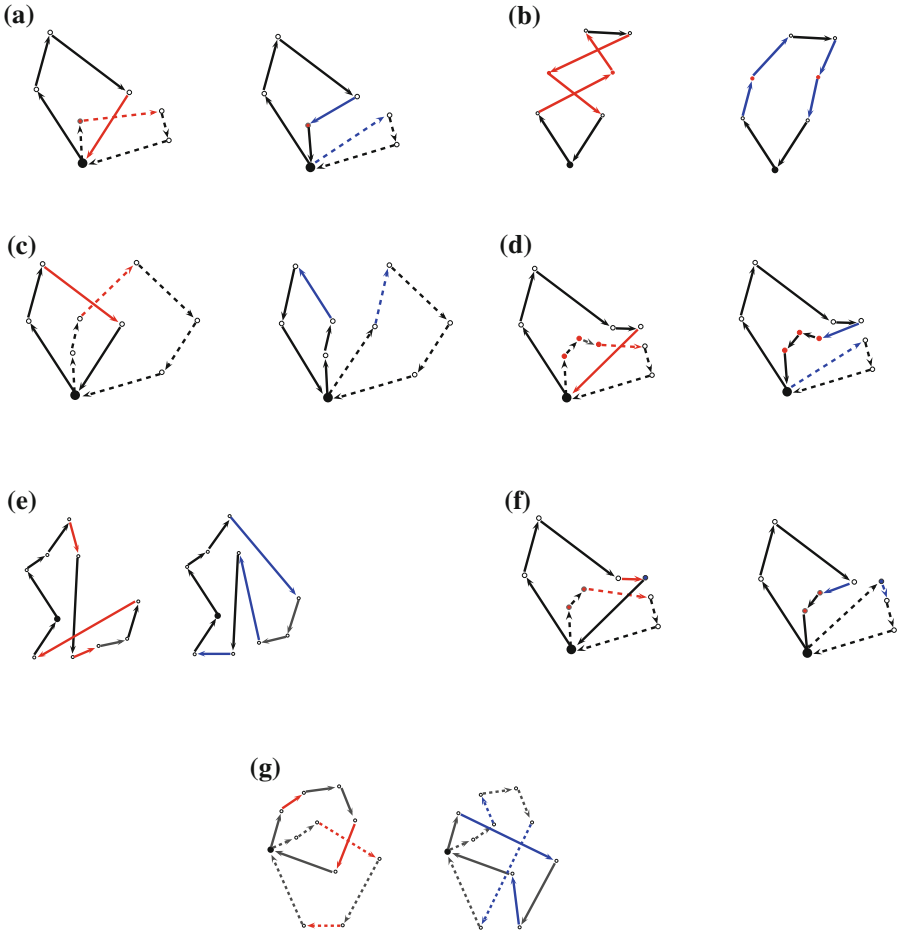

Fig. 1 Local search operators in *VRPH*. **a** One-point move: relocate an existing node into a new position; **b** Two-point move: swap the position of two nodes; **c** Two-opt move: remove two edges from the solution and replace them with two new edges; **d** Or-opt move: remove a string of two, three, or four nodes and insert the string into a new position; **e** Three-opt move: remove three edges from a route and replace them with three new edges; **f** Three-point move: swap the position of a pair of adjacent nodes with the position of a third node; **g** Cross-exchange move: remove four edges from two different routes and replace them with four new edges

many of the best-known solutions to benchmark instances. The record-to-record travel algorithm proposed by Li et al. [23] uses a subset of these local search operators and allows worsening moves according to a deterministic annealing approach. The *VRPH* source code includes an implementation of this algorithm. Finally, we mention the algorithm of Kytöjoki et al. [21]. This algorithm uses all seven local search operators described in Fig. 1 and penalizes edges and modifies the objective function as the search progresses. Their implementation of this algorithm has extremely fast running times for instances with thousands of nodes and some of the implementation details described in this paper are incorporated into *VRPH*.

3 The design of the *VRPH* library

In this section, we describe the motivation behind the design of *VRPH*, including the implementation of the local search operators, data structures, and some of its additional capabilities. We begin with an overview of the object-oriented structure, discuss the uniform behavior and implementation of the various local search operators, and then discuss some technical details such as file input and output.

3.1 Object-oriented structure

VRPH is a completely self-contained library and requires no additional software dependencies. It has an object-oriented design that is easy to understand and can be embedded in other applications with relative ease.

The main class used in *VRPH* is the `VRP` class. Along with a number of public methods to read and write input files and solutions, this class contains many methods that modify the problem instance and solution in different ways. The current solution and problem information are stored in private members of the class and can be modified only by other class members or friend classes. When any solution modification is made using the routines within *VRPH*, the ordering of the customers, the route lengths, loads, service times, and so on are all modified accordingly. All of this is transparent to the caller of the function and it ensures internal consistency of the representations. Additionally, this prevents the user from ever having to perform tedious bookkeeping that is required when one modifies an existing feasible solution.

One novel feature of the library is that the user can maintain a memory of the distinct routes and the distinct solutions discovered during the search in the `VRPRouteWarehouse` and `VRPSolutionWarehouse` classes (the name is borrowed from [22]). These classes both use hash tables to speed up the task of determining uniqueness (the details are discussed in Sect. 3.7).

3.2 Implementation of the local search operators

VRPH can generate high-quality solutions to vehicle routing problems by applying the seven different local search operators. Each operator is implemented in a similar, object-oriented manner and each is designed so that the user can add capabilities to them with minimal effort and code modification.

Each operator is its own class and implements one or both of the following methods, *search(k, rules)* and *route_search(a,b,rules)* where k is the index of a customer or node and a and b refer to indices of individual routes. The other parameter, *rules*, is created by combining different options listed in Table 1. The idea is that the user can combine different requirements for the search into a single parameter that governs how the search for an acceptable move is conducted. These *rules* determine things such as how the search neighborhood is constructed, what type of moves are acceptable (i.e., only improving moves, moves that worsen the solution by less than a fixed amount, etc.), whether to search the entire neighborhood for a move or make the first acceptable move, whether to ensure that the moves must keep certain edges fixed in the solution, and so on.

Table 1 Different options allowed in the *rules*

Option	Meaning
DOWNHILL	Accept only those moves that decrease the total route length
RECORD_TO_RECORD	Accept moves according to the record-to-record acceptance strategy [8]
SIMULATED_ANNEALING	Accept moves according to the simulated annealing metaheuristic [20]
FIRST_ACCEPT	Make the first solution modification that is found that meets the other specifications given in the current *rules*
BEST_ACCEPT	Evaluate all moves in the neighborhood and make the move that leads to a feasible solution with minimum total route length (this is the *best* move)
LI_ACCEPT	If any improving move is found, make this move. If no improving move is found, then make the move that increases the total route length by the smallest amount (a similar acceptance strategy is used in [23])
INTER_ROUTE_ONLY	Search for moves that involve the modification of more than one route
INTRA_ROUTE_ONLY	Search for moves that involve only a single route
USE_NEIGHBOR_LIST	Limit the search to those moves that only involve a particular node's neighbor list
FORWARD	Create an ordered solution buffer by concatenating all routes and search for moves that involve nodes that are found when moving forward in the solution buffer (see [28])
BACKWARD	Create an ordered solution buffer by concatenating all the routes and search for moves that involve nodes that are found when moving backward in the solution buffer (see [28])
RANDOMIZED	When examining the search neighborhood for moves, evaluate these moves in a random order
SAVINGS_ONLY	When comparing two moves, evaluate them by considering only the savings or improvement offered by the moves
MINIMIZE_NUM_ROUTES	When comparing two moves that involve more than a single route, attempt to minimize the number of routes by trying to maximize the sum of the squares of the number of nodes in the routes involved in the move
FIXED_EDGES	Forbid those moves that disrupt any edges that are currently set as *fixed*
TABU	Forbid those moves that result in routes that are *tabu* according to the solution's current memory

Each entry in Table 1 is defined to be a power of two so that there is a single bit in the value of *rules* that corresponds to each of the possible options. This allows one to easily add more options to the value of *rules*. We now describe how we designed the code so that new options could be handled by the various operators with relative ease.

Whenever an operator calls the *search* method, then *create_search_neighborhood* is called in order to create a valid list of other nodes involved in the solution modification, taking into account any neighborhood-related values in the *rules*. If the

route_search method is called then the default behavior defines the search neighborhood as the set of all possible moves involving nodes in the two routes. Thus, if one wished to experiment with a new type of neighborhood restriction, this could easily be done by augmenting the *create_search_neighborhood* method. Having defined a search neighborhood, each operator then calls its own *evaluate* method to determine the validity of the solution modifications encountered in the search neighborhood. If the FIXED_EDGES bit is set in the value of *rules*, then each operator checks to see that no fixed edges are disturbed (the `VRP` class contains a boolean matrix to keep track of fixed edges). If all these checks pass, then the *evaluate* method returns true and places all information about the move in a `VRPMove` data structure.

Once a particular operator's *evaluate* method determines that a move is feasible and has created a valid `VRPMove`, the value of *rules* and the `VRPMove` data structure are passed to the *check_savings* and *check_move* methods which verify that all requirements of the *rules* are satisfied. Thus, in order to add a new kind of requirement to the available possibilities, one would have to ensure that each operator can check this requirement in its *evaluate* method, populate the `VRPMove` fields appropriately, and then add a few lines of code to the *check_savings* and *check_move* functions.

We illustrate the flexibility of this approach with a few examples. We assume that we have a properly instantiated `VRP` object, V, and that we have loaded a 100-node problem with a current solution containing 8 routes.

```
rules = FIRST_ACCEPT+RECORD_TO_RECORD+RANDOMIZED
       +USE_NEIGHBOR_LIST; two_opt.search(V,37,rules);
```

In this example, we search the nearest neighbors of node 37 for a valid two-opt move. We construct the search neighborhood consisting of the nearest nodes (the actual number is an internal parameter that can be modified) and then randomly permute the search neighborhood before searching for moves. Since our *rules* include FIRST_ACCEPT, we make the first feasible move that results in a new total route length that is allowed when using the RECORD_TO_RECORD travel acceptance strategy.

```
rules = FIXED+BEST_ACCEPT+DOWNHILL+INTRA_ROUTE_ONLY;
one_point_move.route_search(V,5,7,rules);
```

In this example, we search for the best one-point move involving routes five and seven. *VRPH* allows the user to fix any edge $u - v$ by setting the u, v entry to true in an internal boolean matrix. In this example, since FIXED is part of the *rules*, any move that removes fixed edges from the existing solution is forbidden. Since the *rules* include BEST_ACCEPT, we search all moves in the neighborhood before selecting the best one, and accept only those moves that reduce the current total route length since the DOWNHILL bit is set.

3.3 Creating an initial solution

After loading a properly formatted file, *VRPH* can create an initial feasible solution with two well-known procedures: the Clarke–Wright algorithm and the sweep algorithm.

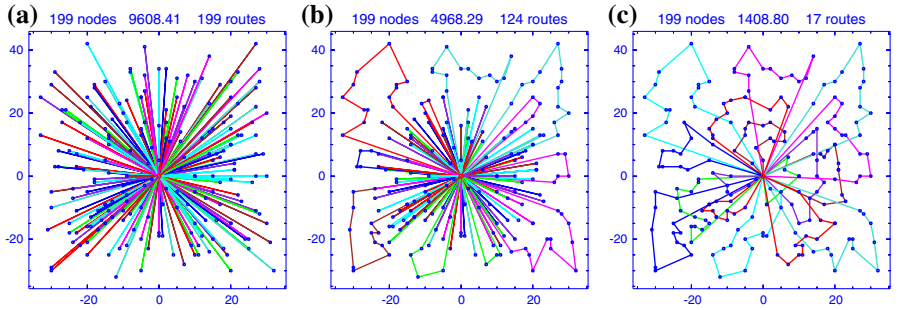

Fig. 2 Progress of the Clarke–Wright algorithm. **a** Initial solution; **b** After 100 merges; **c** Final Clarke–Wright solution

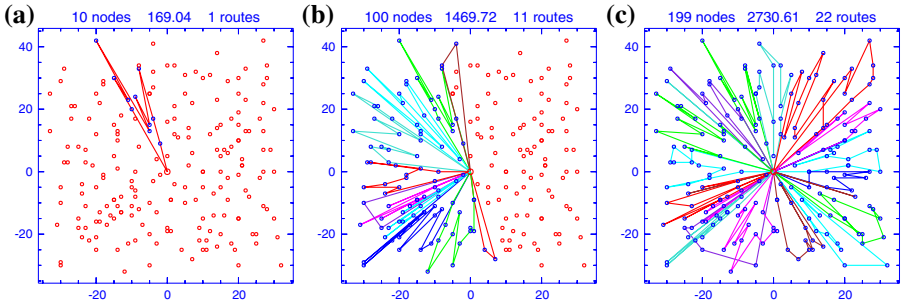

Fig. 3 Progress of the sweep algorithm. **a** 10 nodes added; **b** 100 nodes added; **c** Final sweep solution

The Clarke–Wright algorithm initially assigns each customer to a single route and then successively merges routes together, selecting routes based on the savings or cost reduction of the merge. In *VRPH*, we implement a variant of the Clarke–Wright algorithm [40] that incorporates a shape parameter into the savings calculation, providing a simple and fast way to produce many different initial feasible solutions to a problem by varying only a single parameter. Figure 2 illustrates the progress of the procedure.

The sweep algorithm works only with Euclidean VRP instances. To construct an initial solution using this procedure, we select a random *seed* customer and draw a ray connecting this node to the depot. This seed customer is the first node on the first route. We then sweep this ray around the depot until the ray touches another node which we then add to the current route. When the route is about to become infeasible due to a violation of a capacity constraint or a route length constraint, we start a new route and repeat the process until all customers are routed. In Fig. 3, we show the routes generated by the sweep algorithm.

3.4 Removing and adding customers

When solving a VRP with n customers, it may be useful to consider a smaller problem with fewer nodes. For example, if one is satisfied with how an existing solution services some subset of the n customers, then they can be removed from the solution.

This allows improvement procedures to focus on the remaining customers in the VRP instance.

The `VRP` class contains an array *routed*[ ] where *routed*[*i*] is true if customer *i* is in the current solution, and false otherwise. All heuristic operators discussed in the previous section handle partial problems and unrouted nodes by checking this array when evaluating proposed moves, returning false whenever a proposed move involves an unrouted node. *VRPH* offers several routines that allow the user to take a solution and eject and insert nodes and sets of nodes. The routines are described below and one of the example applications discussed in Sect. 4 provides additional details.

- *eject_node(j)* Remove node j from the solution, replacing the old edges $i - j$ and $j - k$ with the new edge $i - k$.
- *eject_route(r)* Remove all nodes contained in route r from the solution.
- *inject_node(j)* Insert the unrouted node j into the solution so that the increase in total route length is minimal and all routes remain feasible.
- *inject_set*(a[ ], k) Insert all k nodes from the array a[ ] into the solution so that the total route length increase is minimal and so that the resulting routes are feasible. *VRPH* uses several different heuristic approaches to search for an efficient insertion of the set.
- *eject_neighborhood(j)* Select a random set of nodes near j and remove them from the solution.

Other methods within the VRP class allow the user to split a Euclidean instance into two parts via the *split* and *split_routes* methods. Finally, one can create a smaller instance from two existing solutions S_1 and S_2 by removing all nodes belonging to routes that are in both S_1 and S_2. If many solutions share a particular set of routes, then it is reasonable to believe that these routes belong in the final solution. By removing the customers in these routes from the solution, the heuristics can focus on the resulting smaller subproblem.

3.5 Data structures

For an n-node problem, the current solution at any stage in the solution procedure is stored in a doubly linked list contained in two arrays of length $(n + 1)$: *next_array* and *pred_array*. We use the method suggested in Kytöjoki et al. [21] where negative indices in these arrays indicate the beginning of a new route. This permits a very compact way of storing the solution information that exists in a contiguous block of memory, allowing for potentially improved cache performance over a more traditional pointer-based linked list implementation.

In any VRP solution, each customer is assigned to a single route (we do not allow for split deliveries). *VRPH* stores this information in the *route_num* array and also maintains an array of `VRPRoute` objects that represents the individual route and contains all the relevant information such as the length, load, start and end nodes, number of customers, and the ordering of the customers in the route if desired (this information is already contained in the *next_array* and *pred_array*). All of these internal data structures are protected members of the class, but there are public accessor

Table 2 A 10-node VRP example solution and our method for exporting it

Route	Ordering
1	0-2-5-8-0
2	0-1-3-4-0
3	0-6-7-9-10-0
Solution buffer	
10 -2 5 8 -1 3 4 -6 7 9 10 0	

methods such as *get_total_route_length* that allow an application read-only access to these data.

3.6 Input and output

The TSPLIB format [36] is a widely used file format for representing TSP instances that offers several options for representing the VRP. The file format used by *VRPH* adheres closely to the TSPLIB format, and the sample problems and README file included with *VRPH* illustrate the various options in detail. *VRPH* allows double precision distances by setting the EDGE_WEIGHT_FORMAT to EXACT_2D and an arbitrary distance matrix can be used by setting EDGE_WEIGHT_TYPE to EXPLICIT. Finally, we note that *VRPH* can handle problems where the planning horizon has multiple days and there is some interaction among the visit times across the different days (otherwise one could of course just solve the problems independently). This is done by adding an additional line NUM_DAYS: *D* for a *D*-day problem, and by then providing *D* days worth of demands and service times in the appropriate sections.

VRPH allows the user to store solutions to a problem in buffers and to write solutions to files in a simple format using the commands *export_solution_buff* and *write_solution_file*. The formats of the solution are identical for these two commands. The first entry in the solution buffer or file is the number of non-depot nodes in the solution. Following this entry, we list the order in which customers are visited, using a negative index to indicate the first node in each route. The buffer or file produced by these functions can then be imported into a properly initialized VRP object by calling either *import_solution_buff* or *read_solution_file*. When these functions are called, the current solution stored by *VRPH* is discarded and all internal data structures are updated to reflect this new solution. A simple example of this storage method is shown in Table 2.

3.7 Hashing

In addition to providing several ways to export and import solutions, *VRPH* stores a large pool of the best *s* solutions (*s* is an internal parameter that can be varied). When a potentially new solution is encountered, we need to quickly determine whether or not the solution is already in the pool. We do this by using an *h*-bit hash table and

associate an h-bit integer with each solution. We are unaware of any other uses of hashing in the VRP literature, and so we describe the details of our procedure.

The idea behind our hash function is to transform the solution into a standard form, export the standardized solution into a buffer, and then associate an h-bit integer with this buffer. We are given an R-route solution to an n-node VRP. For each j from 1 to R, we represent the j-th route as an ordered list, r_j. In particular, we write $r_j = \{a_j, \ldots, z_j\}$ where a_j is the index of the first customer visited in route j and z_j is the index associated with the last customer visited in route j. Additionally, we store a list of n random 32-bit integers, $Y_0, Y_1, \ldots, Y_{n-1}$. Given this notation and this list Y, we associate a positive integer with each solution by performing the steps given in Algorithm 1.

Algorithm 1 Hashing a VRP Solution

1: **for** $j = 1$ to R **do**
2: **if** $a_j > z_j$ **then** Reverse the ordering of route j
3: **end if**
4: **end for**
5: Create a list L containing the resulting, possibly reversed routes, and sort this list in terms of the index of the first node visited in the route
6: With $L = \{r_1, r_2, \ldots, r_R\}$, concatenate these routes into a single list $\{m_1, m_2, \ldots, m_n\}$
7: Return $H = \bigoplus_{i=1}^{n-1} Y_{(m_i + m_{i+1}) \bmod n} \bmod 2^h$

In Step 2, we ensure that each route is oriented so that the index of the first node in each route is smaller than the index of the last node visited in each route (this step is omitted for asymmetric instances). These routes are then concatenated to form a single list in Steps 5 and 6. In Step 7, we associate a 32-bit integer $Y_{(m_i+m_{i+1}) \bmod n}$ with each pair of nodes in the ordered list and then sum them together (when computing this sum, we use the bitwise XOR operation rather than normal addition). We produce the hash value H by taking the low h bits.

We illustrate this procedure using the example given in Table 2. Since the index of the start node is less than the index of the end node in each of the three routes, no reversals are required, and so we sort these routes by the start node in Step 5 and concatenate them together to form the list $\{1, 3, 4, 2, 5, 8, 6, 7, 9, 10\}$. We then associate one of our 10 random numbers $Y[0], Y[1], \ldots, Y[9]$ with each pair of consecutive numbers in this list and XOR them together to compute an h-bit hash value:

$$H = (Y[4] \oplus Y[7] \oplus Y[6] \oplus Y[7] \oplus Y[3] \oplus Y[4] \oplus Y[3] \oplus Y[6] \oplus Y[9]) \bmod 2^h.$$

This provides a fast method of associating an h-bit integer with a given solution. When a potentially new solution is encountered, *VRPH* hashes the solution, producing an integer H. Then we check the entry in position H in the hash table. If the location is empty, then we instantly know that the solution is not currently one of the s best solutions. If entry H in the hash table is not empty, then we perform additional checks to determine whether or not this is a new solution.

Our code includes similar routines to compute hash values for individual routes as well. This is useful for applications that require a large number of feasible routes

for a VRP instance as it allows one to quickly determine whether or not a particular route has been previously generated. As mentioned earlier, the `VRP` class includes a `VRPRouteWarehouse` subclass that allows one to store these routes. The *VRPH* code includes an example application that illustrates this capability in the context of a set partitioning formulation of the VRP.

4 Applications of *VRPH*

In this section, we describe several applications of the library that are included with the source code. We begin by describing a local metaheuristic algorithm that we have implemented using the capabilities of *VRPH*. This provides a method for producing high-quality solutions for instances of the classical VRP. We then describe three applications built with the *VRPH* library that illustrate additional capabilities. The first illustrates how the library can be used to interface with the open source VRP solver included with *SYMPHONY*, providing a quick upper bound to accelerate the search for provably optimal solutions. The second application demonstrates how to interface *VRPH* with a mixed-integer programming solver. In order to insulate ourselves from a particular solver, this code is written using the COIN-OR Open Solver Interface (OSI) [7]. The third application illustrates the node ejection and injection routines and how they can be used to improve good solutions generated by a metaheuristic.

4.1 A local search metaheuristic

We implement a variant of the fast and flexible record-to-record travel (RTR) algorithm presented in [23]. This algorithm alternates between two phases, a diversification phase and an improvement phase. In the diversification phase, the goal is to accept some worsening moves and explore new parts of the solution space. This is followed by an improvement phase where only improving moves are allowed as we seek a local minimum in the search space. The algorithm terminates after it has been unable to escape a local minimum after several attempts. The algorithm requires the following parameters:

- D: Controls the size of the main loop in the diversification phase. A value between 25 and 50 is reasonable. Default value is 30.
- δ: Controls how much deterioration of the objective function is allowed in the diversification phase. Defaults to $\delta = 0.01$.
- K: The number of local minima that must be reached before perturbing the solution or terminating the algorithm. Default value is $K = 5$.
- N: The size of the neighbor list that is searched when running the local search operators. A larger value will slow the algorithm down as there are more possibilities to consider. The default value is $N = 30$.
- P: The number of times the solution is perturbed once the search is stuck in a local minimum. Default value is $P = 2$.

Algorithm 2 An algorithm for the VRP based on record-to-record travel

```
1: Generate a random λ ∈ (0.5, 2) and generate an initial feasible solution using the parameterized Clarke–
   Wright algorithm
2: Select a set of local search operators 𝒰
3: Set the record R equal to the current total route length, set the threshold T = (1+δ)R, and set k = p = 0
4: while p < P do
5:     for i = 1 to D do
6:         for all Operators u ∈ 𝒰 do
7:             for all Nodes j in the solution do
8:                 Apply operator u to node j using record-to-record travel
9:             end for
10:        end for
11:    end for
12:    while Improving moves can be found do
13:        for all Operators u ∈ 𝒰 do
14:            for all Nodes j in the solution do
15:                Apply operator u to node j accepting only improving moves
16:            end for
17:        end for
18:    end while
19:    if The current solution is a new record then
20:        Update R and T and set k = 0
21:    end if
22:    k = k + 1
23:    if k == K then
24:        Perturb the solution
25:        p = p + 1
26:    end if
27: end while
28: Return the best solution found
```

After compiling *VRPH*, Algorithm 2 is implemented in the binary executable file *vrp_rtr*. There are a number of options for this implementation, some of which are illustrated by the following command.

```
vrp_rtr -f Christofides_10.vrp -D 50 -L 5 -N 30 -h
ONE_POINT_MOVE -h TWO_OPT -h THREE_OPT -out C10.sol
```

The above command will run the RTR solver on the Christofides_10.vrp benchmark problem using five different initial solutions, three heuristic operations (one-point move, two-opt, and three-opt), a diversification loop of 50 iterations, neighbor lists of size 30, with the best solution written to the file C10.sol.

4.2 Integrating with *SYMPHONY*'s exact VRP solver

The distribution of the open source mixed-integer programming package *SYMPHONY* includes a solver especially designed for generating optimal solutions to VRP instances. The process of finding and proving the optimality of a solution can be accelerated by using *VRPH* to provide an upper bound on the optimal solution at the beginning of the procedure. As *VRPH* and *SYMPHONY*'s VRP solver both read TSPLIB-formatted files, it turns out that incorporating *VRPH* into this solver is surprisingly straightforward. About 20 lines of code need to be added to *SYMPHONY*'s

existing *main* routine. After *SYMPHONY* processes the input file, we instantiate a `VRP` object, load the same input file, and generate 10 solutions to the problem. This requires only a few seconds of running time since the instances are relatively small. We determine the best objective function value achieved during these runs, set *SYMPHONY*'s internal upper bound value, and allow *SYMPHONY* to solve the problem as usual. If the solution found by *VRPH* is indeed optimal, then when *SYMPHONY* eventually discovers this solution, it reports an infeasibility, effectively proving that the *VRPH* solution is optimal. In our testing of this interface using small problems with 50 nodes or less, the solution found by *VRPH* was optimal in most cases.

4.3 Integrating with mixed-integer programming solvers

The VRP can be formulated as a set partitioning problem. Given an n-node VRP instance and the set of all feasible routes, $\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$ where route R_j has cost c_j, for every customer i and every route j, we let $a_{ij} = 1$ if customer i is contained in route j and $a_{ij} = 0$ otherwise. Using decision variables $y_j = 1$ if route j is selected and $y_j = 0$ otherwise, the VRP can be formulated as follows:

$$\text{Minimize} \quad \sum_{j=1}^{m} c_j y_j \tag{1}$$

$$\text{s.t.} \quad \sum_{j=1}^{m} a_{ij} y_j = 1, \quad \forall i = 1, 2, \ldots, n \tag{2}$$

$$y_j \in \{0, 1\}, \quad \forall j = 1, 2, \ldots, m. \tag{3}$$

Of course, the set of all feasible routes $\mathcal{R}$ is too large to enumerate, even for very small problems. Nevertheless, one can use a heuristic algorithm to generate many feasible solutions to a VRP instance and use these solutions to construct subsets $\mathcal{S} \subset \mathcal{R}$. By solving the set partitioning problem using the smaller set $\mathcal{S}$, it is often possible to find new solutions that are better than any of the other solutions. This technique was used as a post-processor in [37] and was the basis for a parallel algorithm described in [14].

We include an example that demonstrates how to use *VRPH* as part of this approach. We use *VRPH*'s implementation of the RTR metaheuristic to generate many solutions to a VRP instance. We use the hashing capability discussed in Sect. 3.7 to keep track of unique routes encountered in the search and then add these routes as variables (columns) to a set partitioning problem. This is accomplished through an interface between *VRPH* and the COIN-OR Open Solver Interface [7] that allows the code to be insulated from a particular choice of commercial or open source mixed-integer programming solvers. We then use the mixed-integer programming solver to generate the optimal solution to this set partitioning problem. This process can be repeated any number of times where we begin each iteration with a new random solution. To ensure that the set partitioning problems can be solved in a reasonable amount of time, we set a parameter M to be the maximum number of variables (columns) allowed in the set

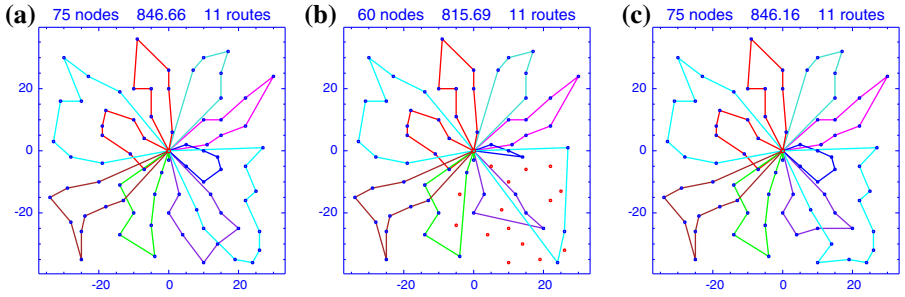

Fig. 4 An example of the ejection and injection routine. **a** Original solution; **b** Solution after ejecting 15 nodes; **c** Improved solution

partitioning problem. Once the problem has M columns, we delete D non-basic variables at random and continue the procedure. This application also illustrates the use of *VRPH*'s internal routines to store unique routes and solutions through its internal hash tables. Pseudocode is given in Algorithm 3.

Algorithm 3 An algorithm for the VRP combining RTR with a set partitioning formulation

```
Set M to be the maximum number of columns in the set partitioning problem
Set D to be the number of columns removed from the set partitioning problem
Set S to be the size of VRPH's internal solution warehouse
Set k to be the total number of iterations
for i = 1 to k do
    Generate a random λ ∈ (0.5, 2) and generate an initial feasible solution using the parameterized
        Clarke–Wright algorithm
    Improve the solution to the VRP instance using the RTR algorithm with default parameters
    for j = 1 to S do
        Run intra-route improvements on all routes in solution j in the solution warehouse
        for all Routes r in the solution do
            if r is not in the set partitioning problem then
                Add a new variable (column) for the route
            end if
            if There are M columns in the set partitioning problem then
                Delete D non-basic random columns from the set partitioning problem
            end if
        end for
    end for
    Solve the current set partitioning problem to optimality
end for
Return the current solution to the set partitioning problem
```

4.4 Neighborhood ejection

The final example application that we describe demonstrates the ability of *VRPH* to remove and insert customers from an existing solution. This application begins by either reading in an existing solution to a VRP instance or generating a set of new

solutions with the RTR algorithm. For each solution, the application then performs t iterations of the following neighborhood ejection and injection routine. First, a customer c_0 and a set of $m - 1$ additional customers that are located near c_0 are selected at random. This neighborhood of m customers is then removed or ejected from the solution using the *eject_neighborhood* function. This gives a partial solution of the original VRP instance. These customers are then re-inserted back into this partial solution using two different methods. The first method uses a greedy algorithm where we repeatedly select one of the removed customers at random and add the customer to the existing partial solution using cheapest insertion so that the increase in total route length is minimized. This is a greedy procedure as we select the cheapest insertion at each stage. The second method also utilizes cheapest insertion, but allows previous insertions to be undone if this removal allows for a cheaper insertion of the current customer under consideration. This is similar to the notion of *regret* described in [16]. After all customers have been re-inserted back into the solution, if the resulting solution is better than the original solution, then the next iteration continues with the new solution. Otherwise, the original solution is re-loaded and the next iteration continues. At the end of the run, the best solution found during the search is reported. An example of this procedure is given in Fig. 4 where 15 nodes are removed from a 75-node instance.

The source code includes three additional applications, one that improves solutions by running a simulated annealing algorithm [20], one that demonstrates the generation of initial solutions by running either the sweep or Clarke–Wright algorithms, and a third that creates PostScript plots of existing solutions by linking with the open source *PLPlot* library [31]. Additional details about these applications can be found in the source files themselves which contain many comments.

5 Computational results

We present computational results produced by *VRPH* on several sets of standard benchmark problems from the literature. The computational results reported in this section use the applications described in Sect. 4. A user should be able to produce very similar results by running the scripts accompanying the code. We note that reproducing the results exactly is unlikely. We have observed that even using different versions of the same compiler on the same machine yield slightly different results due to internal code optimization. All of the results reported here were obtained by compiling with *g++*, version 4.1.2, and running on a quad-core AMD 2.3 GHz processor with 16 GB of main memory. As *VRPH* is single-threaded, only a single core was used in these experiments.

We begin by showing the improvement in running time that results from providing *SYMPHONY*'s exact VRP solver with an upper bound prior to beginning the branch and bound process. We ran the application described in Sect. 4 on 15 well-known problems from the literature with 40 or more nodes. We generated 10 initial solutions to each instance by using the parameterized Clarke–Wright algorithm and then ran the RTR metaheuristic on each solution. We report the best solution found by *VRPH* over these 10 runs, the optimal solution to the problem, and the amount of time spent

Table 3 Results of incorporating *VRPH* with *SYMPHONY*'s exact solver

Problem data			Optimal solution	*VRPH* solution	*VRPH* time	*SYMPHONY* time (sec)	
Name	Nodes	Routes				(with *VRPH*)	(no *VRPH*)
A-n44-k6	44	6	937	941	2.89	126.95	174.27
A-n45-k6	45	6	944	948	2.27	18.26	34.58
A-n46-k7	46	7	914	914	2.72	2.51	9.05
A-n48-k7	48	7	1073	1073	3.29	158.27	200.84
A-n53-k7	53	7	1010	1010	3.79	141.8	282.3
B-n41-k6	41	6	829	839	2.44	1.3	2.39
B-n43-k6	43	6	742	742	2.63	17.14	35.21
B-n44-k7	44	7	909	909	2.93	0.88	1.61
B-n50-k7	50	7	741	741	3.35	0.47	0.98
B-n51-k7	51	7	1016	1016	3.98	0.78	19.11
B-n52-k7	52	7	747	747	4.09	1.69	3.02
B-n56-k7	56	7	707	707	4.82	4.88	9.21
B-n64-k9	64	9	861	884	6.89	27.18	21.03
att-n48-k4	48	4	40002	40013	3.31	7.23	7.57
E-n51-k5	51	5	521	521	3.29	3.04	8.46
Average % above optimal				0.3%			
Average running time (sec)					3.51	34.16	53.98

generating these 10 solutions. Finally, we report the time spent running *SYMPHONY*'s branch-and-bound procedure, both with and without the upper bound provided by the *VRPH* solution. These results are reported in Table 3.

The upper bound from *VRPH* allows the branch-and-bound procedure to complete in less time for 14 of the 15 instances. The only exception is *B-n64-k9* where the *VRPH* solution with length 884 requires 10 routes while the optimal solution of 861 requires only nine routes. Ten runs of *VRPH*'s implementation of the RTR metaheuristic requires 3.5 s on average and finds the optimal solution for 10 of 15 problems. Finally, we note that for problem *A-n54-k7* (not in Table 3), *VRPH* found the optimal solution and *SYMPHONY* proved the optimality after about 15 h of computing time. However, when we ran *SYMPHONY* on this instance without providing an upper bound, it did not complete in 24 h and we aborted the run at this point.

The remaining computational results involve running the RTR metaheuristic, the set partitioning algorithm, and the neighborhood ejection algorithm described in Sect. 4. Each of these applications uses 10 solutions obtained by running *VRPH*'s implementation of the RTR metaheuristic. For each application, we used a fixed seed of *VRPH*'s internal random number generator so that the same 10 solutions were produced by each application. This allows us to measure the amount of improvement obtained by using either the set partitioning approach or the ejection and injection approach. We used the default parameters for the RTR metaheuristic. For the set partitioning-based algorithm described in Algorithm 3, we use the open source *GLPK* [12] mixed-integer

Table 4 Solutions for 14 VRPs from [5] and [6]

Problem data			Best known	RTR		Set partitioning	Ejection	
Number	Nodes	Routes		1 run	Best of 10		Random	Regret
1	50	5	524.61[a]	524.61	524.61	524.61	524.61	524.61
2	75	10	835.26[a]	846.18	845.05	842.37	835.77	836.18
3	100	8	826.14[a]	829.44	827.39	827.39	827.39	827.39
4	150	12	1028.42[a]	1045.77	1034.94	1032.12	1031.16	1031.16
5	199	16	1291.29[b]	1311.89	1311.89	1307.36	1307.13	1307.13
6	50	6	555.43[a]	555.43	555.43	555.43	555.43	555.43
7	75	11	909.68[a]	927.76	912.89	909.68	912.89	911.76
8	100	9	865.94[a]	865.94	865.94	865.94	865.94	865.94
9	150	14	1162.55[a]	1169.10	1169.10	1164.12	1169.10	1164.13
10	199	18	1395.85[a]	1417.96	1414.35	1410.97	1414.35	1408.152
11	120	7	1042.11[a]	1043.18	1042.11	1042.11	1042.11	1042.11
12	100	10	819.56[a]	819.56	819.56	819.56	819.56	819.56
13	120	11	1541.14[a]	1572.69	1546.60	1542.86	1546.60	1543.35
14	100	11	866.37[a]	866.97	866.37	866.37	866.37	866.37
Average % above best known				0.81	0.44	0.28	0.31	0.22
Total CPU time (sec)				18.03	181.22	128.83	306.75	441.02
Average CPU time (sec)				1.29	12.94	9.20	21.91	31.50

[a] Rochat and Taillard [37]; [b] Mester and Bräysy [28]

programming solver and limited the set partitioning problem to 500 routes (columns). For the ejection application described in Sect. 4.4, we generate 1000 random neighborhoods containing 15 customers each and report the best solution found during this search.

Tables 4, 5, and 6 show the results of these computations and compare them with the best-known solutions from the literature. The final row in each table summarizes the results of the different algorithms and provides the running times.

Based on the results given in Tables 4, 5, and 6, on average, a single run of the RTR implementation produces solutions that are generally between 0 and 2.5% above the best-known solution with an average running time of less than 5 s. Selecting the best solution from 10 runs of RTR produces solutions that are typically within 1.5% of the best-known solution. By incorporating the set partitioning formulation, the solutions improve by a few tenths of a percent. The neighborhood ejection routines appear to offer a similar amount of improvement but require more computing time. Even though the use of *regret* in the neighborhood injection routines requires about 50% more CPU time than the *random* injection routines, there appears to be very little (if any) improvement in solution quality. Finally, we note that we experimented with increasing the number of columns in the set partitioning application. However, when we increased the relevant parameter to 1000 or so, we experienced a dramatic slowdown in the solution times of the resulting integer programming problems. We

Table 5 Solutions for 20 VRPs from [13]

Problem data			Best known	RTR		Set partitioning	Ejection	
Number	Nodes	Routes		1 run	Best of 10		Random	Regret
1	240	9	5623.47[a]	5766.16	5724.69	5715.10	5662.23	5662.34
2	320	10	8431.66[b]	8679.04	8523.41	8466.92	8466.92	8466.92
3	400	10	11036.22[b]	11396.92	11104.80	11047.01	11078.85	11078.85
4	480	10	13592.88[b]	13691.80	13690.86	13635.31	13634.11	13634.11
5	200	5	6460.98[b]	6661.69	6472.39	6460.98	6472.38	6466.68
6	280	7	8404.26[b]	8468.49	8468.49	8414.75	8415.21	8415.21
7	360	9	10156.58[b]	10325.81	10252.16	10195.59	10195.59	10195.59
8	440	10	11643.90[c]	12051.11	11923.20	11910.43	11869.61	11869.61
9	255	14	580.02[a]	607.25	588.31	588.31	588.31	588.13
10	323	16	738.44[a]	763.22	751.217	747.95	747.76	747.63
11	399	18	914.03[a]	948.26	940.39	938.64	934.85	935.37
12	483	19	1104.84[a]	1147.40	1138.61	1136.87	1133.01	1132.32
13	252	26	857.19[d]	877.35	871.75	867.87	867.11	868.13
14	320	30	1080.55[d]	1115.67	1098.42	1088.01	1093.69	1093.69
15	396	33	1340.24[a]	1375.25	1372.43	1362.47	1360.68	1360.68
16	480	37	1616.33[a]	1667.01	1650.08	1641.91	1639.24	1639.22
17	240	22	707.76[d]	711.25	710.38	707.79	709.56	709.56
18	300	27	995.13[d]	1020.03	1015.72	1013.85	1012.66	1011.87
19	360	33	1365.97[a]	1391.03	1384.42	1382.17	1381.01	1381.01
20	420	38	1819.99[a]	1850.07	1850.07	1844.93	1842.57	1842.63
Average % above best known				2.60	1.54	1.16	1.08	1.08
Total CPU time (sec)				98.08	989.14	252.25	11099.43	1514.59
Average CPU time (sec)				4.90	49.46	12.61	54.97	75.73

[a] Groër [14]; [b] Nagata and Bräysy [30]; [c] Prins [32]; [d] Nagata and Bräysy [29]

suspect that switching to a faster commercial solver such as *CPLEX* or *Gurobi* would allow one to increase this parameter with a much less noticeable slowdown.

6 Extending *VRPH*

We discuss several ways that our code could be extended to handle variants of the classical VRP. Currently, *VRPH* handles arbitrary, possibly asymmetric distance matrices by using the *TSPLIB* FULL_MATRIX option. Although *VRPH* is able to produce reasonably good solutions to the few asymmetric benchmark instances we are aware of, we have not made any effort to improve its performance on these types of problems.

VRPH has the ability to read in and store time windows for customers, but does not use this information at all and is, therefore, not able to generate good solutions to instances of the *VRPTW*. In order for the local search operators to account for time

Table 6 Solutions for nine VRPs from [39]

Problem data			Best known	RTR		Set partitioning	Ejection	
Number	Nodes	Routes		1 run	Best of 10		Random	Regret
100A	100	11	2041.34[a]	2078.04	2077.12	2071.31	2071.93	2071.93
100B	100	11	1939.90[a]	1951.27	1944.26	1940.61	1940.61	1940.61
100C	100	11	1406.20[a]	1411.66	1406.20	1406.20	1406.20	1406.20
100D	100	11	1580.46[a]	1598.45	1597.60	1593.30	1597.60	1597.60
150A	150	15	3055.23[a]	3253.18	3057.73	3055.88	3056.64	3056.64
150B	150	14	2727.20[b]	2809.27	2808.01	2801.34	2797.23	2797.23
150C	150	15	2341.84[c]	2420.34	2382.26	2382.26	2382.26	2382.26
150D	150	14	2645.40[c]	2678.87	2663.15	2661.84	2661.91	2661.91
385	385	47	24366.69[d]	24785.51	24602.26	24525.03	24536.89	24536.89
Average % above best known				2.19	1.05	0.89	0.92	0.92
Total CPU time (sec)				15.01	154.98	228.27	255.74	369.26
Average CPU time (sec)				1.67	17.22	25.36	28.42	41.03

[a] Mester and Bräysy [28]; [b] Nagata-Bräysy [29]; [c] Taillard [38]; [d] Groër [14]

windows, one would have to first add a new time window option to the *rules* parameter as discussed in Sect. 2. The *evaluate* method of each local search operator would then need to be able to verify the feasibility of a solution by taking into account the changes in arrival times for the customers affected by a solution modification.

Although *VRPH* was designed with single-depot instances in mind, one could use *VRPH* for a multiple depot instance by instantiating multiple `VRP` classes, one for each depot. Given $k > 1$ depots, it would be simple to use the node injection and ejection routines to re-assign customers to different depots. In particular, given a customer c currently assigned to depot i, we could assign c to depot j by ejecting the customer from the current solution for depot i and then injecting c into the current solution for depot j. More complicated operations could be handled in a similar manner. One drawback to this approach is that each `VRP` object currently has a copy of the entire distance matrix, but this would become an issue only for very large instances with many depots.

Finally, we mention some performance statistics and enhancements for our code. For each local search operator, *VRPH* keeps track of the number of times the *evaluate* and *move* methods are called. Even for small problems, the *evaluate* method is called millions of times when running the RTR metaheuristic and only a very small percentage of these calls result in actual solution modifications, typically because the proposed move that is evaluated would lead to an infeasible solution. For example, running *vrp_rtr* on the 199-node Christofides_5.vrp instance, using all seven local search operators, the code reports statistics on the number of times the *evaluate* and *move* methods were called. These statistics are shown in Table 7.

While over 300 million calls to the *evaluate* methods were made during this run (which took less than 20 s), only about 0.15% as many calls were made to the *move* method. This is largely due to the fact that many of the potential solution modifications

Table 7 Statistics describing the number of calls to the *evaluate* and *move* methods for the seven local search operators

Local search operator (move)	Number of *move* calls	Number of *evaluate* calls
One-point	122,152	25,352,386
Two-point	19,622	19,879,621
Three-point	2,908	34,828,448
Two-opt	252,346	101,823,601
Three-opt	35,444	3,647,704
Or-opt	21,691	54,179,840
Cross-exchange	2,644	75,589,328
Total	456,807	315,300,928

that are evaluated lead to infeasible solutions. Additionally, by using a code profiler to see where the time is spent, we find that nearly half of the total execution time is spent in the *evaluate* methods for the two-opt, Or-opt, and cross-exchange operators. Furthermore, the *move* methods collectively account for less than 1% of the running time of the algorithm.

Based on the number of calls and the percentage of time spent in the *evaluate* methods, it is clear that the implementation of these operators could be improved by either speeding up the *evaluate* methods or by intelligently reducing the number of calls. We have taken some care in our implementation of the *evaluate* methods by trying to quickly identify infeasible moves. Implementing some of the ideas from [19] to improve the efficiency of the operators in *VRPH* could be another way to more efficiently construct and search the neighborhoods.

7 Conclusions

VRPH provides open source software that offers implementations of several algorithms for generating good solutions to instances of the classical VRP. We have demonstrated how to integrate *VRPH* with both *SYMPHONY*'s exact VRP solver and mixed-integer programming solvers. The code also offers a uniform interface that users can extend. We hope that this library will be useful to researchers studying the VRP and that it will provide a flexible tool to implement new metaheuristic algorithms and to address new, more complex variants of the classical VRP.

Acknowledgments We thank the anonymous reviewers and associate editor for their suggestions that improved the paper. We thank Ted Ralphs for providing advice on integrating *VRPH* with *SYMPHONY*.

References

1. Applegate, D., Bixby, R., Chvátal, V., Cook, W., Espinoza, D., Goycoolea, M., Helsgaun K.: The Concorde source code. http://www.tsp.gatech.edu/concorde.html (2010)
2. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: The Traveling Salesman Problem: A Computational Study. Princeton University Press, Princeton, NJ (2006)

3. Bräysy, O., Gendreau, M.: VRPTW, part I: Route construction and local search algorithms. Transp. Sci. **39**, 104–118 (2005)
4. Bräysy, O., Gendreau, M.: VRPTW, part II: Metaheuristics. Transp. Sci. **39**, 119–139 (2005)
5. Christofides, N., Eilon, S.: An algorithm for the vehicle dispatching problem. Oper. Res. Q. **20**, 309–318 (1969)
6. Christofides, N., Mingozzi, A., Toth, P.: The vehicle routing problem. In: Christofides, N., Mingozzi, A., Toth, P., Sandi, C. (eds.) Combinatorial Optimization, pp. 315–338. Wiley, Chichester, UK (1979)
7. COIN-OR. Open Solver Interface (OSI). https://projects.coin-or.org/Osi/ (2010)
8. Dueck, G.: New optimization heuristics: the great-deluge algorithm and the record-to-record travel. J. Comput. Phys. **104**, 86–92 (1993)
9. Fukasawa, R., Longo, H., Lysgaard, J., Poggi, D., Reis, M., Uchoa, E., Werneck, R.: Robust branch-and-cut-and-price for the capacitated vehicle routing problem. Math. Program. **106**(3), 491–511 (2006)
10. Gendreau, M., Potvin, J-Y., Bräysy, O., Hasle, G., Løkketangen, A.: Metaheuristics for the vehicle routing problem and its extensions: a categorized bibliography. In: Golden, B., Raghavan, S., Wasil, E. (eds.) The Vehicle Routing Problem: Latest Advances and New Challenges, pp. 143–169. Springer, New York (2008)
11. Glover, F., Taillard, E.: A user's guide to tabu search. Ann. Oper. Res. **41**(1), 1–28 (1993)
12. GLPK, The GNU Linear Programming Kit. http://www.gnu.org/software/glpk/ (2010)
13. Golden, B., Wasil, E., Kelly, J., Chao, I.-M.: The impact of metaheuristics on solving the vehicle routing problem: algorithms, problem sets, and computational results. In: Crainic, T., Laporte, G. (eds.) Fleet Management and Logistics, pp. 33–56. Kluwer, Boston (1998)
14. Groër, C.: Parallel and serial algorithms for vehicle routing problems. Ph.D thesis, University of Maryland, College Park, MD (2008)
15. Groër, C.: The *VRPH* software. http://sites.google.com/site/vrphlibrary/ (2010)
16. Hassin, R., Keinan, A.: Greedy heuristics with regret, with application to the cheapest insertion algorithm for the TSP. Oper. Res. Lett. **36**(2), 243–246 (2008)
17. Helsgaun, K.: An effective implementation of the Lin-Kernighan traveling salesman heuristic. Euro. J. Oper. Res. **126**(1), 106–130 (2000)
18. Helsgaun, K.: The LKH source code. http://www.akira.ruc.dk/~keld/research/LKH/ (2010)
19. Kindervater, G., Savelsbergh, M.: Vehicle routing: handling edge exchanges. In: Aarts, E., Lenstra, J.K. (eds.) Local Search in Combinatorial Optimization, pp. 337–360. Princeton University Press, Princeton, NJ (2003)
20. Kirkpatrick, S., Gelatt, C., Vecchi, M.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)
21. Kytöjoki, J., Nuortio, T., Bräysy, O., Gendreau, M.: An efficient variable neighborhood search heuristic for very large scale vehicle routing problems. Comput. Oper. Res. **47**(2), 329–336 (2005)
22. Le Bouthillier, A., Crainic, T.: A cooperative parallel meta-heuristic for the vehicle routing problem with time windows. Comput. Oper. Res. **32**, 1685–1708 (2005)
23. Li, F., Golden, B., Wasil, E.: Very large-scale vehicle routing: new test problems, algorithms, and results. Comput. Oper. Res. **32**, 1165–1179 (2005)
24. Lin, S., Kernighan, B.: An effective heuristic algorithm for the traveling salesman problem. Oper. Res. **21**, 2245–2269 (1973)
25. Lodi, A., Punnen, A.: TSP software. In: Gutin, G., Punnen, A. (eds.) The Traveling Salesman Problem and its Variations, pp. 737–749. Kluwer, Dordrecht (2002)
26. Lysgaard, J.: CVRPSP: A package of separation routines for the capacitated vehicle routing problem. Working Paper 03–04 (2004)
27. Mester, D., Bräysy, O.: Active guided evolution strategies for the large scale vehicle routing problem with time windows. Comput. Oper. Res. **32**, 1593–1614 (2005)
28. Mester, D., Bräysy, O.: Active-guided evolution strategies for large-scale vehicle routing problems. Comput. Oper. Res. **34**, 2964–2975 (2007)
29. Nagata, Y., Bräysy, O.: Efficient local search limitation strategies for vehicle routing problems. In: Hemert, J., Cotta, C. (eds.) EvoCOP, Volume 4972 of Lecture Notes in Computer Science, pp. 48–60. Springer, Berlin (2008)
30. Nagata, Y., Bräysy, O.: Edge assembly-based memetic algorithm for the capacitated vehicle routing problem. Networks **54**, 205–215 (2009)
31. PLPlot: The PLPlot software package. http://plplot.sourceforge.net/ (2010)

32. Prins, C.: A GRASP evolutionary local search hybrid for the vehicle routing problem. In: Pereira, F., Tavares, J. (eds.) Bio-Inspired Algorithms for the Vehicle Routing Problem, pp. 35–53. Springer, Berlin (2009)
33. Ralphs, T.: Parallel branch and cut for capacitated vehicle routing. Parallel Comput. **29**, 607–620 (2003)
34. Ralphs, T., Guzelsoy, M., Mahajan, A.: The SYMPHONY source code. https://projects.coin-or.org/SYMPHONY (2010)
35. Ralphs, T., Kopman, L., Pulleyblank, W., Trotter, L.: On the capacitated vehicle routing problem. Math. Program. **94**, 343–359 (2003)
36. Reinelt, G.: TSPLIB—a traveling salesman problem library. ORSA J. Comput. **3**, 376–384 (1991)
37. Rochat, Y., Taillard, E.: Probabilistic diversification and intensification in local search for vehicle routing. J. Heuristics **1**, 147–167 (1995)
38. Taillard, E.: Parallel iterative search methods for vehicle routing problems. Networks **23**, 661–676 (1993)
39. Taillard, E.: VRP benchmarks. http://mistic.heig-vd.ch/taillard/problemes.dir/vrp.dir/vrp.html (1993)
40. Yellow, P.C.: A computational modification to the savings method of vehicle scheduling. Oper. Res. Q. **21**, 281–293 (1970)