Heuristics for the Robust Vehicle Routing Problem with Time Windows

Simen Braaten[1] Ola Gjønnes[1] Lars Magnus Hvattum[2] Gregorio Tirado[3]

[1]Department of Industrial Economics and Technology Management, Norwegian University of Science and Technology

[2]Faculty of Logistics, Molde University College, Norway

[3]Universidad Complutense de Madrid, Spain

July 20, 2016

Abstract

Uncertainty is frequently present in logistics and transportation, where vehicle routing problems play a crucial role. However, due to the complexity inherent in dealing with uncertainty, most research has been devoted to deterministic problems. This paper considers a robust version of the vehicle routing problem with hard time windows, in which travel times are uncertain. A budget polytope uncertainty set describes the travel times, to limit the maximum number of sailing legs that can be delayed. This makes sure that improbable scenarios are not considered, while making sure that solutions are immune to delays on a given number of sailing legs. Existing exact methods are only able to solve small instances of the problem and can be computationally demanding. With the aim of solving large instances with reduced running times, this paper proposes an efficient heuristic based on adaptive large neighborhood search. The computational study performed on instances with different uncertainty levels compares and analyzes the performance of four versions of the heuristic and shows how good quality solutions can be obtained within short computational times.

Keywords: Robust optimization, metaheuristic, uncertainty, travel time.

1 Introduction

Vehicle routing problems play a crucial role in logistics and transportation, and many different variants have been widely studied in the literature during the last decades (Toth and Vigo, 2014). Most of this research has been devoted to deterministic problems, where all the parameters related to the problem data are known with certainty beforehand. However, in real life it is common to observe uncertainty, which decision makers must take into consideration.

The motivation of the optimization problem faced in this paper stems from the maritime shipping industry, where routing decisions are complicated by frequent delays that may severely affect route plans and delivery times. Weather conditions, port congestion or strikes, and mechanical problems are possible reasons for ship delays. As a consequence of the high operating cost of the ships, a pure cost minimization perspective means that no buffers are imbedded in the schedules, and any delays imply a need for reoptimizing the schedules. Given the massive importance of maritime transportation to the world's economy (Christiansen et al., 2013), improving the efficiency of this sector has significant positive effects, and facilitates further growth.

This paper considers a *robust vehicle routing problem with time windows* (RVRPTW) that arises in this context of the shipping industry. The robustness implies that the solution should be immune to uncertainty in data. The relevance of introducing robust optimization solutions was stressed by Ben-Tal and Nemirovski (2000), who concluded that, in real-world applications of linear programming, the possibility that a small uncertainty in the data can make the optimal solution meaningless from a practical viewpoint, cannot be ignored. In our case, the travel times of the vessels will be taken as uncertain parameters, considering several possible sets of combinations of delays. This leads to a more realistic setting, in which the proposed solutions will remain feasible even for realizations of the travel times that differ from the expectation.

Agra et al. (2012, 2013) developed exact solution methods for the RVRPTW, but these methods are quite time consuming and can only be used to solve small instances. The main contribution of this paper is an efficient heuristic, based on adaptive large neighborhood search (ALNS), that is able to provide very good quality solutions for large RVRPTW instances. To design the heuristic, the concept of *programming by optimization* (Hoos, 2012) is used. That is, the heuristic is implemented by adding a large range of search components, possibly with many parameters that influence their individual behavior. Instead of manually testing the effectiveness of each component, or selecting components by trial and error, an automated algorithmic optimization procedure is used to determine the best functioning combination of search components. To this end, the irace software for parameter tuning is used (Lopez-Ibanez et al., 2011). Additionally, we test four different versions of the heuristic, to evaluate the effect of three important design choices: one version of the heuristic does not include a proposed preprocessing phase, another version excludes all local search components, and a third version uses a more common move acceptance rule. The main version, on the other hand, includes both preprocessing, local search, and an alternative move acceptance rule.

The remainder of this paper is structured as follows. The next section defines the problem at hand, and introduces its mathematical formulation. Section 3 presents a brief review of relevant literature. The methodology proposed to solve the problem, including preprocessing, destruction, insertion and local search heuristics, an efficient feasibility check method, and an overarching ALNS heuristic, is introduced in Section 4. A detailed computational study is presented in Section 5, and finally, some concluding remarks and suggestions for future work are discussed in Section 6.

2 Problem description

The RVRPTW is a generalization of the well-known *vehicle routing problem with time windows* (VRPTW). The RVRPTW results from a special case in maritime shipping, where goods are transported between given pairs of loading and unloading ports. Whenever the goods are transported using full vessel loads, the problem can be modelled using a graph where nodes represent the loading operation, the movement to the unloading port, and the unloading operation. Additional nodes represent the starting points of empty vessels and the artificial final positions of the vessels after completing their itineraries. Arcs between nodes represent travel from the previous unloading operation (or the starting position) to the next loading operation (or the artificial final position). There is a time window associated to the loading operation of each transportation task, and the set of vessels is typically heterogeneous, with vessels varying in size and speed.

Full load maritime pick-up and delivery problems therefore reduce to instances of an asymmetric, heterogeneous fleet VRPTW. Let us further consider the possibility of delays happening between the loading operation and the arrival at the next loading operation. It is important to hedge against these delays, and the shipping company may want a robust schedule such that even when a few delays happen, the vessels' itineraries are still not in violation of any time windows. This leads to the RVRPTW.

A mathematical formulation of the RVRPTW was first proposed in (Agra et al., 2012). We present a slightly adapted mathematical model to clarify the problem description. To model the uncertainty in travel times in the presence of hard time windows, a step-wise (layered) formulation is used. Let us denote the set of nodes by N, using i and j to denote general nodes, and let N^* be the subset of N that excludes the origin o and destination d. The set of arcs is denoted as A and contains pairs of nodes, (i, j). The set of vessels is called V with elements v. Now we can assign to each vessel v a cost c_{ijv} for traversing edge (i, j), and to each node i a time window $[a_i, b_i]$. Then x_{ijv} are binary decision variables that take the value 1 if vessel v uses the edge (i, j).

To account for the robustness and the time windows, some additional notation is required. First, to simplify the notion of visiting the nodes in an order, we introduce steps s from 0 and up to some upper bound $\bar{S}$ on the length of a path, for example $\bar{S} = |N|$. These steps are a different interpretation of the layers used in (Agra et al., 2012). With these in place, we define $\mathcal{A}^S$ as the set of valid edge-step combinations. For example, in the 0'th step, only edges going out from o can be used, as we always need to start in the origin. Thus, for the 0'th step, $\mathcal{A}^S$ only contains 3-tuples such as $(o, i, 0)$, meaning that a vessel can go from the origin to node i in step 0. This allows us to consider the binary decision variable y_{ijvs}, which is 1 if and only if vessel v goes from i to j as the s'th step. Second, to handle the robustness, we introduce the set $\mathcal{T}_v^\Gamma$ of delay patterns p that a vessel v can face, each with a maximum of Γ delays. With this, we can define t_{ijvp} as the travel time for v along (i, j) in delay pattern p. The proposed model is the following:

Sets

N - Nodes
N^* - $N \setminus \{o, d\}$
A - Arcs
$\mathcal{A}^S$ - Allowable arc/step combinations
V - Vessels
$\mathcal{T}_v^\Gamma$ - Delay patterns for v

Indices

i, j - Node
v - Vessel
s - Step
p - Delay pattern

Parameters

$\bar{S}$ - Maximal number of steps
c_{ijv} - Cost for vessel v of travelling edge (i, j)
t_{ijvp} - Travel time for vessel v of travelling edge (i, j) facing delay pattern p
o - Origin node
d - Destination node
a_i - Time window opening at node i
b_i - Time window closing at node i

Decision variables

x_{ijv} - Flag variable for vessel v travelling edge (i, j)
y_{ijvs} - Flag variable for vessel v travelling edge (i, j) as its step number s

Objective function and restrictions

$$\min \quad z = \sum_{v \in V} \sum_{(i,j) \in A} c_{ijv} x_{ijv} \tag{1}$$

$$\text{s.t.} \quad \sum_{v \in V} \sum_{(i,j) \in A} x_{ijv} = 1, \qquad i \in N^*, \tag{2}$$

$$\sum_{(j,i,s-1) \in \mathcal{A}^S} y_{jiv,s-1} - \sum_{(i,j,s) \in \mathcal{A}^S} y_{ijvs} = \begin{cases} -1 & \text{if } (i = o) \\ 1 & \text{if } (i = d \quad \text{and} \quad s = \bar{S}) \\ 0 & \text{otherwise} \end{cases}, \qquad \begin{matrix} 1 \leq s \leq \bar{S}, \\ i \in N_s, v \in V, \end{matrix} \tag{3}$$

$$\sum_{(i,j,s) \in \mathcal{A}^S} y_{ijvs} = x_{ijv}, \qquad (i,j) \in A, v \in V, \tag{4}$$

$$\sum_{\substack{(i,j) \in A, \\ (i,j,s_1) \in \mathcal{A}^S}} a_i y_{ijvs_1} + \sum_{\substack{s = s_1, \ldots, s_2 - 1: \\ (i,j) \in A}} \sum t_{ijvp} y_{ijvs} \leq \sum_{\substack{(i,j) \in A: \\ (i,j,s_2) \in \mathcal{A}^S}} b_j y_{ijvs_2}, \qquad \begin{matrix} 1 \leq s_1 < s_2 < \bar{S}, \\ v \in V, p \in \mathcal{T}_v^\Gamma, \end{matrix} \tag{5}$$

$$x_{ijv} \in \{0,1\}, \qquad (i,j) \in A, v \in V, \tag{6}$$

$$y_{ijvs} \in \{0,1\}, \qquad (i,j,s) \in \mathcal{A}^S, v \in V. \tag{7}$$

The total travel cost is minimized in the objective function (1), while constraints (2) ensure that all nodes are visited. Constraints (3) control the conservation of flows, and constraints (4) bind the general edge variables together with the step specific ones. Lastly, constraints (5) ensure that each node is visited within its time window, no matter which delay pattern in $\mathcal{T}_v^\Gamma$ occurs.

3 Literature review

This section reviews the relevant literature regarding robust routing problems. Adulyasak and Jaillet (2014) point out that robust solutions are often made so that they can withstand any realization of the uncertain travel times. This creates a potential for very conservative solutions. Given, for example, a transportation problem where a delay is almost unambiguously worse than no delay, it will often result in a deterministic model that simply contains worst case realizations (Ben-Tal et al., 2009). The possibly over-conservative solutions will tend to have an unnecessary high price of robustness, so that the increase in cost from taking uncertainty into account does not reflect the gain in actual robustness (Bertsimas and Sim, 2004). Because of this, research on stochastic routing problems have had a tendency to consider other ways of dealing with the uncertainty.

One natural way to handle the uncertainty is to optimize the expected value. Kenyon and Morton (2003) notes that this also reduces to a deterministic problem in some cases, and that this is a very common and reasonable way to handle uncertainty in many different problems.

However, other formulations approach the issue from a different angle. In a *chance constrained* model, every feasible solution needs to have at most some predetermined chance of failing. A typical failing criterion in our context is to violate a time window, so a chance constrained formulation could be to restrict the probability of missing any time windows to less than 5% (Kenyon and Morton, 2003).

Recent contributions concerning robust problems have often introduced *budgeted uncertainty* (Bertsimas and Sim, 2004). Budgeted uncertainty is a way to limit the realizations of uncertainty that needs to be considered, excluding the often very improbable, or even impossible, event that all variables reach their worst case value simultaneously (Bertsimas and Sim, 2004). In our case we consider budgeted uncertainty to limit the maximum number of trips that can be delayed.

The deterministic VRPTW has been very well studied over the last decades (Bräysy and Gendreau, 2005), but many authors have also approached VRPs with a multitude of stochastic attributes (Dror and Trudeau, 1986; Gendreau et al., 1996; Stewart Jr. and Golden, 1983), with a special focus primarily on stochastic demand (Souyris et al., 2013). Even though this has not led to a significant amount of research on robust solutions of VRP instances with stochastic travel times, there are some recent contributions in this area. Sungur et al. (2008) and Gounaris et al. (2013) proposed different robust exact methods to the vehicle routing problem with demand uncertainty, comparing their performance and analyzing the potential benefits of such robust approaches. A tutorial containing robust models for VRPs with different uncertain parameters, such as costs, demand, time, and customers, can be found in Ordóñez (2010), including some computational results and two applications: large scale emergencies and courier deliveries. Recently, Solano-Charris et al. (2015) considered a robust VRP with uncertain arc costs, represented by a set of discrete scenarios. They use a lexicographic min-max criterion as the objective function: the worst cost over all scenarios must be minimized, and ties are broken using the other scenarios. The authors present a mixed integer linear formulation of the problem and several heuristics.

To the best of our knowledge, the robust VRPTW with budgeted uncertainty, as described in this paper, has only been discussed by Agra et al. (2012, 2013), who presented exact solution methods. Agra et al. (2013) solved instances with up to 50 customers, but the solution time for larger instances occasionally exceed their time limit of half an hour. A similar problem, the robust VRP with deadlines and stochastic demand and travel time, was solved by Lee et al. (2012). However, the uncertainty in travel times is handled in a different way, and their solution strategies are not able to solve instances with more than 50 customers either, spending more than an hour on problems with 25 customers.

4 Heuristics

The proposed heuristic is implemented in a modular fashion, allowing different concepts to be combined, or single components to be analysed in their own right. The top level does parsing, preprocessing and construction of initial solutions, whereas the metaheuristic layer takes these solutions and improves them. This section describes all of these modules, which are embedded together to form the final framework.

4.1 Main framework

The general framework of the proposed solution method is illustrated in Algorithm 1. Before any computation is done, the instance to solve is cloned, so that it can be preprocessed (if selected) without changing the original. Then, every insertion heuristic available constructs a solution to the clone, by trying to insert all the nodes into an initially empty solution. Each solution is assigned an adjusted value, so as to select which solutions should be kept. This adjusted value is simply the objective value of the solution if every node is assigned, and if the solution has unassigned nodes, the adjusted value is a large number M plus the number of unassigned nodes. This measure gives preference to feasible solutions, and for infeasible solutions, it gives preference to those with few unassigned nodes. The solutions with the smallest adjusted values are selected as candidates.

At this point, the ALNS comes into play, being made aware of only the unprocessed instance, to avoid the search being stuck with an instance that has accidentally been made unsolvable during preprocessing. The framework hands it one candidate after the other, and starts the local timer for each of them. An ALNS iteration involves three steps: First, the selection of a single destruction heuristic, and a number k of nodes to remove. Second, after the destruction heuristic has completed, a single insertion heuristic is selected to insert as many of the unassigned nodes as possible. Third, if the new solution is accepted, a local search is performed. These three steps are repeated until the iteration limit is reached, or the local or global timer runs out.

Algorithm 1 Main framework

input: *instance* - instance data,
params - ALNS parameters,
$\mathbb{C}$ - a set of construction heuristics,
$\mathbb{P}$ - a set of preprocessing heuristics,
τ - time limit per solution,
$\tau^{overall}$ - time limit for the solution procedure.

```
1: procedure The framework
2:     instance ← parseFile(data)
3:     instanceClone ← instance
4:     for all (preprocessing heuristics p in ℙ) do
5:         instanceClone ← p.RemoveEdges(instanceClone)                    # Modifying the clone
6:     constructedSolutions ← ℂ.CreateSolutions(instanceClone)             # Constructing solutions to the clone
7:     solutions ← constructedSolutions.BestCandidates()
8:     for all (constructed solutions s in solutions) or until τ^overall do
9:         solution ← ALNS(s, instance, τ, params)                         # Improve solutions for the original instance
10:        if (solution is better than bestSolution) then
11:            bestSolution ← solution
12:    return bestSolution
```

The local timer decides, to a large degree, how many candidates are evaluated. Given a fixed global time limit, longer local time limits give the ALNS the opportunity to work for a long time on a few candidates, intensifying the search. A short local time limit means that additional candidates will be tested, thereby achieving greater diversification.

The main elements that come into play within this framework are described in the following sections. Preprocessing heuristics are introduced in Section 4.2, insertion and destruction heuristics in Sections 4.3 and 4.4, respectively, and local search heuristics in Section 4.5. An algorithm to efficiently check the feasibility of a solution is proposed in Section 4.6, whereas the details of the ALNS are described in Section 4.7.

4.2 Preprocessing heuristics

In what follows we present different methods for removing edges prior to the construction of routes, with the aim of improving the likelihood of getting good results when creating initial solutions. Some of the preprocessing heuristics simply remove edges that would never be part of a feasible solution. These are called required preprocessing heuristics, and are always run as the instance is parsed. Other options allow removing edges that are technically possible, but seem unlikely to be part of a good solution. Toth and Vigo (2003) showed that edges that intuitively seem bad, are seldom part of an optimal solution. The aim of these options is to reduce the size of the feasible neighbourhoods, allowing a search through them to finish faster. These optional preprocessing heuristics might actually make the problem unsolvable, but solutions constructed based on only the most promising edges can still help the solution procedure in the search of a good solution.

Long edge removal

The simplest optional preprocessing heuristic eliminates an edge (i, j) if it takes more time to traverse than some proportion γ^l of the longest travel time in the problem.

Time window closes too early

This required preprocessing eliminates edges that connect nodes with incompatible time windows. Time windows are said to be incompatible if the from-node opens so late that it is impossible to reach the to-node in time, that is, if the time of opening, a_i, of the time window of the from-node, added to the worst case travel and service time for a given vessel, $\hat{t}_{ijv}$, is greater than the latest allowed arrival, b_j, of the to-node, the edge (i, j) for vessel v is removed. Any delay pattern with a delay on this edge will violate the time window. Since the feasible routes should be able to handle every delay pattern, this makes it impossible for the edge to be part of any feasible route, as long as $\Gamma \geq 1$. Thus, this heuristic only removes edges that cannot appear in any feasible solution, and is always used by default.

Edges that imply relatively long waiting time

One of the ways simple construction heuristics may get stuck creating infeasible solutions, is by using cheap edges directly from nodes with early time windows to nodes with late time windows. Since these are feasible, and contribute relatively little to the cost of a route, they might seem attractive for a greedy heuristic. However, this creates solutions where vessels end up visiting some nodes a lot earlier than necessary, and spends so much time waiting for the time window to open, that they never have time to visit nodes that close at earlier times. In order to reduce

how often this happens, one could simply disallow edges that lead to very long *waiting times*, defined as the difference $w_{ij} = a_j - (b_i + \hat{t}_{ijv})$.

To eliminate the edges that cause such long waiting time, two alternatives are proposed. The simplest option removes an edge if it implies a waiting time longer than some proportion γ^w of the longest waiting time, which is a criterion similar to the one used by *Long edge removal*. The other option consists of removing an edge if the difference between the time windows of its nodes is greater than some multiplier ν^w times the travel time of the edge. The idea behind this is that, if the distance between the time windows is many times bigger than the travel time, this will lead to a relatively long wait. The reference point from node i is b_i, the latest possible departure from i, and thus the edge is eliminated if $\nu^w t_{ijv} \leq a_j - b_i$. Both of these options are optional preprocessing heuristics.

Merging nodes

Another optional preprocessing is to merge nodes i and j that intuitively seem like they should be visited immediately after one another. This decision is based on the relative difference between the nodes, in distance and time windows, and the loss of flexibility by combining them. Three conditions are considered when deciding if two nodes are deemed compatible to be merged or not.

i) The distance between them is shorter than a proportion γ^m of the longest edge in the problem, $t_{ijv} \leq \gamma^m t_{max}$.

ii) To maintain a large degree of flexibility, a merge is penalized if j's time window closes first, meaning that going from i to j forces the vessel to arrive at and leave i earlier than strictly necessary, and thereby restricting some flexibility in the solution. This is measured by $\Pi^f = \pi^f \max\{0, (b_i - b_j)\}$, where π^f is the flexibility loss penalty factor.

iii) Third, a merge is also penalized if j's time window opens much later than i's closes, since it does not seem a good idea to merge nodes that should be visited at completely different times, even if they are geographically close to each other. This is measured by $\Pi^w = \pi^w \max\{0, (a_j - b_i)\}$, with π^w being the window distance penalty factor.

In order for i and j to be merged, we require $\Pi^f + \Pi^w \leq \alpha^m (b_i - a_i)$, that is, the combined penalty of flexibility loss and window distance should be less than some proportion α^m of i's time window width. This really only means that both of the penalty terms need to be less than $\alpha^m (b_i - a_i)$, as only one of them will have a nonzero value at any time.

4.3 Insertion heuristics

Insertion heuristics start with a given empty or partial solution and a list of unassigned nodes, and then decide which nodes to insert and where to place them. They use a wide array of different decision rules, focusing on travel cost, flexibility, or time window constraints, or trying to reconcile these three considerations. These insertion heuristics are described in what follows.

Greedy append

Inspired by the nearest neighbour heuristic presented in Solomon (1987), this heuristic calculates which unassigned node is cheapest to append at the end of one of the current routes and perform this insertion. It consciously ignores time window constraints, attempting to create a route as short, or cheap, as possible, as a starting point for the local search heuristics to improve. This heuristic has been implemented with both the mean travel time and the travel cost of edges as cost measure.

K-regret

The K-regret decision rule (Pisinger and Ropke, 2005) is a variant of a greedy insertion heuristic, trying to simulate some foresight. It uses a certain cost measure c^r to find the K best insertions for each node and then calculates the regret-value for each node. The regret value could informally be described as how much you will regret not placing this node right now. If we let c^r_{ip} denote the cost of the p'th best insertion for node i, the regret value $r_i = c^r_{iK} - c^r_{i1}$ shows how much you stand to lose if i does not get placed in one of its $K-1$ best insertions. Nodes with large regret-values can quickly become very expensive to insert if their optimal positions are occupied and thus they should be inserted first. K-regret has been implemented using travel cost, mean travel time and worst case travel time as cost measures.

Solomon insertion

Inspired by a general insertion heuristic described by Solomon (1987), this is a shell heuristic that needs two cost measures, c_1 and c_2, as parameters. Using these, the heuristic separates the two important steps of insertion: position selection and node selection. First, for each unassigned node, c_1 is used to evaluate its possible insertion positions, and determine the best one. Then, when every node is assigned a position at which to be potentially inserted, c_2 is used to evaluate these different insertions, and select the one to actually perform. Travel cost and time measures have been used as c_1, c_2, or both, obtaining different versions of the heuristic.

Fill the gap

This is a novel insertion heuristic that looks for large gaps between consecutive time windows in a route with the aim of making the solution time-window-feasible. The size of a gap between two subsequent nodes i and j for a vessel v, g_{ijv}, is calculated as the difference between the closing of the time window of j, b_j, and the latest possible departure from i, b_i. Thus, $g_{ijv} = b_j - b_i$. Routes with large gaps often have a potential for visiting more nodes, so the heuristic iteratively inserts nodes in the greatest time window gaps it can find. It is the node that increases the cost of the route the least, given that it is a feasible insertion, that is used to fill the gap.

4.4 Destruction heuristics

Insertion heuristics can be used to insert unassigned nodes into a partial initial solution, but they must be used in combination with destruction heuristics if no unassigned nodes exist (and

thus the initial solution is complete). The idea is to follow a ruin and recreate-paradigm, where each search step consists of unassigning (destroying) and then reassigning (inserting) nodes.

The simplest approach is to unassign a collection of nodes randomly. However, in what follows we present some more elaborate methods that try to select the nodes to be unassigned to benefit the insertion heuristics that are applied afterwards.

Remove k

These destruction heuristics use different cost measures to decide which k nodes should be removed from the solution, leading to several alternatives that are presented below.

- **Remove k random nodes.** Simply choose k nodes at random and remove them from their route.
- **Remove the k nodes giving the greatest cost saving.** If i is the predecessor of j and k its successor, the cost saved, Δc, after unassigning j is $\Delta c = c_{ik} - (c_{ij} + c_{jk})$.
- **Remove the k nodes giving the greatest saving in travelled time.** Since the travelled time is not necessarily proportional to the cost, travelled time could be used instead of cost in order to decide which nodes to unassign.
- **Remove the k nodes giving the greatest saving in potential delay,** where the potential delay is the difference between the expected travel time and the worst case travel time.
- **Remove the k most expensive edges.** Similar rules are used for the k **longest edges**, the k **edges with longest potential delay**, and the k **cheapest edges**, where an edge is removed by simply unassigning the corresponding from-node. It seems counterintuitive to remove the cheapest edges from a solution, but this is used as a way to strengthen the diversification of the solution, as suggested by Pisinger and Ropke (2005). The same applies to removing the k **shortest edges**, and the k **edges with the shortest potential delay**.

Remove k nodes from the same route

Choose one of the routes at random and then randomly remove k nodes from it. This heuristic differs from the general Remove k-heuristics mentioned above in that it only considers a single route, and typically removes a lower number of nodes. The idea in this case is to maintain much of the structure in the solution, and to focus the changes in a small area.

Make k feasible gaps

This heuristic is used when, after a certain iteration, there are unassigned nodes that could not be inserted in any of the routes. In this case, the idea is to make k feasible gaps to try to diversify the search by increasing the probability of inserting those nodes. The heuristic iteratively chooses one of the unassigned nodes at random, and removes the first node which,

when removed, makes it feasible to insert the randomly chosen one. This is repeated until k nodes are removed, and thus there are k gaps where at least one of the unassigned nodes can be inserted.

Note: Calculation of k

In all the destruction heuristics described above, the value of k is selected as a proportion of the number of nodes in the problem. This proportion is randomly selected between 0 and an upper bound that develops as a linear function of the number of iterations since the last improvement. This upper bound is always at least some number $\bar{q}_{min} \in [0, 1]$. Thereafter, as the number of iterations approaches the limit L^{NI} on the number of non-improving iterations, the upper bound grows linearly toward its upper limit $\bar{q}_{max}$. The proportion q of nodes to remove is chosen randomly in $(0, \bar{q}_{min} + \frac{n^{NI}}{L^{NI}}(\bar{q}_{max} - \bar{q}_{min}))$, where n^{NI} is the number of non-improving iterations, and L^{NI} is the limit on such iterations. From this we get $k = \lfloor q \cdot |N| \rfloor$ for all except the same route-heuristic, where $|N|$ is replaced by the number of elements in the route in question. The idea behind this is to make the search stay in a rather close neighbourhood immediately after finding a new best solution, but to gradually allow it to take more drastic steps, potentially unassigning ever more nodes, as it is running out of iterations.

4.5 Local search heuristics

There exists a multitude of local search heuristics for routing problems, as shown by Vidal et al. (2013). In what follows, we describe the variations of local search heuristics that have been implemented in the proposed framework, having been selected taking into account the particularities of the problem at hand.

Consecutive two-point move

A simple local search heuristic for the VRP is the two-point move (Groër et al., 2010). This rule looks for two nodes in a route that will save cost when being reordered. We have chosen to modify the heuristic, so that it only tries to reorder every pair of consecutive nodes, taking into account that two consecutive nodes in a route have a large probability of having time windows that are close to each other's. Nevertheless, this leads to a very small neighbourhood compared to many other heuristics.

1-interchange

Defined by Osman (1993), a slightly more complex variant is the 1-interchange heuristic. Here, node i will take j's place in vessel w's route and vice versa. This is a more flexible heuristic that can switch any two nodes between two routes, but most node pairs by far will have incompatible time window constraints, and are therefore impossible to switch. This means that a relatively large amount of checking is usually done before finding a feasible move.

Interroute 2-opt

Vidal et al. (2013) present a local search heuristic for the VRP that splits two routes in two by removing an edge, and then connects one of the parts from the first route with one of the parts of the second route, naming this the 2-opt*. We have implemented a minor modification of this heuristic, as the routes of the problem studied in this paper have an origin and a destination. This implies that not every combination of reattaching the routes is possible, since the routes have to follow the initial travel direction. The proposed interroute 2-opt heuristic looks for partial routes that should be serviced by different vessels, and when compatible points are found in two routes, the vessels simply switch nodes if this is cost beneficial, and complete each other's routes from that point out. The heuristic also looks for opportunities to make the opposite move, making two vessels switch the first parts of their routes, but finish as before.

4.6 Feasibility check

The process of checking whether a solution is feasible or not is crucial in the development of the proposed algorithm and could potentially be very time consuming, given that it must be checked whether each route can withstand every possible delay pattern. In order to make this feasibility check efficiently, a dynamic programming algorithm, inspired by Agra et al. (2013), is proposed. It checks whether all nodes are assigned and, if so, whether each route is feasible, which is clearly the major step here.

To reduce the number of times a solution has to be assessed, Vidal et al. (2013) observed that the successful algorithms for Multi Attribute VRPs in many cases stored information on previously evaluated routes. In the RVRPTW it may be very beneficial to store such information, as the delays make the process of evaluating a solution even more burdensome than in the deterministic case. For this reason, the routes are saved in a tree structure, so throughout this section, the word *node* refers to a tree node in such a structure. There is room for confusion, as a customer is represented as a node in the graph in our problem, but in this section, the word *customer* consistently refers to actual stops in a route. Although each node is connected to a unique customer, each customer could be represented by many nodes, one for each time it has appeared in a route.

In Figure 1, the root node represents the origin, and for each customer visited directly from the origin (first in a route), there is a first-layer branch. These branches are called children of the root node. Each node has a list $NextStops$ which contains the customers that have immediately followed the node's customer in a route, and a list $Children$ which contains the corresponding child tree nodes, one for each customer in $NextStops$. As we visit customers in orders we have not yet seen, we create new nodes in the tree, so that each route can be found as a path from the root and down to a leaf node. This means that in Figure 1, for example, we have assessed, among others, routes starting with $origin \rightarrow 1 \rightarrow 2$, $origin \rightarrow 1 \rightarrow 5 \rightarrow ...$, and so on.

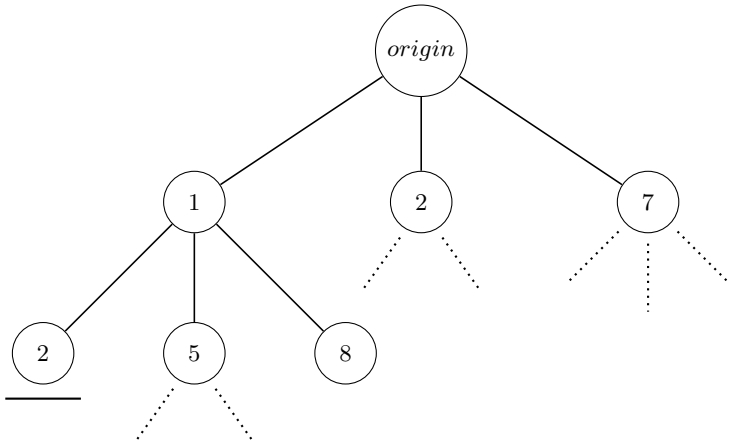

Figure 1: A part of a feasibility tree.

In order to introduce the delays, a set of labels is assigned to each tree node, corresponding to the different possible arrival times to the customer. Each label consists of the number of delays experienced so far, when service starts at the customer, and whether or not that is a feasible time to start the service. In addition, from the service start time, a departure time can be deduced. If a node has an infeasible label the tree is cut. This is also shown in Figure 1, where the tree is cut after $origin \rightarrow 1 \rightarrow 2$ because it has an infeasible label. From this it can be deduced that no route that starts out like that can ever be feasible, because there exists a delay pattern such that the time window at customer 2 cannot be met.

The algorithm proposed to check the feasibility of a route, which may be called by any of the proposed heuristics, is shown in Algorithm 2, where the existing tree is traversed, starting in the origin and iteratively looking for a node that represents the next customer in the route. This is done until one out of three cases happen. First, an infeasible node might be found, in which case the route in question is infeasible, and the algorithm halts. Second, the end of the route might be reached without finding an infeasible node, which means that the route is feasible, and the algorithm returns. Third, the bottom of the tree might be reached without having finished the route yet, and the algorithm continues by appending additional nodes to the tree.

Algorithm 2 Feasibility check: Checking a route

```
input: route - the route whose feasibility should be decided,
       instance - data related to an instance.
       feasibilityTree - the feasibility tree.
 1: procedure CheckRoute
 2:     currentNode ← feasibilityTree.RootNode          # Start at the tree node corresponding to the origin
 3:     nextCustomer ← first customer after origin in route
 4:     while (nextCustomer is in currentNode.NextStops) do
 5:         if (currentNode.IsFeasible) then
 6:             nextNode ← currentNode.Children.Get(nextCustomer)          # Child node for nextCustomer
 7:             currentNode ← nextNode                                     # Continue to that child node
 8:             if (currentNode.Customer = destination) then
 9:                 return true                                  # We have reached the end of the route
10:             nextCustomer ← the subsequent customer in route
11:         else
12:             return false
13:     end-while
14:     repeat          # Arriving here means we reached the bottom of the tree, but still have customers left
15:         nextNode ← TreeNode(nextCustomer)          # Create a tree node corresponding to nextCustomer
16:         nextNode.Labels ← ExtendLabels(currentNode, nextNode)
17:         RemoveDominatedLabels(nextNode)
18:         currentNode.NextStops.Add(nextCustomer)              # Record a move directly to nextCustomer
19:         currentNode.Children.Add(nextNode)                 # Add the corresponding child node to the tree
20:         currentNode ← nextNode                                         # Continue to that child node
21:         nextCustomer ← subsequent customer in route
22:         if (any of the labels of currentNode are infeasible) then
23:             currentNode.IsFeasbile ← false
24:             return false
25:     until end of route
26:     return true
```

In the third case, the tree is extended by iterating through the rest of the route, creating a new node in the tree for each customer, and saving it as a child of the preceding node, while recording the customer as a *NextStop*. While doing this, it is also necessary to *extend* the labels of the preceding node to the new node. Algorithm 3 demonstrates how this extension is performed. Each label in the preceding node is read, and then the arrival time of the vessel at the new customer is calculated. If the label has not considered the Γ available delays yet, another label is created, which uses the delayed travel time when calculating a new arrival time, and thus one more label with an incremented delay count is created. If any of the two new labels imply a violation of the time window, it is marked as infeasible, and then the labels are attached to the new node.

Algorithm 3 Extending the labels from one tree node to the next

input: $currentNode$ - the feasibility tree node of the current customer,
$nextNode$ - the feasibility tree node of the next customer.

procedure ExtendLabels
  $currentCustomer \leftarrow currentNode.Customer$
  $nextCustomer \leftarrow nextNode.Customer$
  for all (labels l in $currentNode.Labels$) **do**
    $arrivalTime \leftarrow l.DepartureTime$+TravelTime($currentCustomer, nextCustomer$)
    $serviceStart \leftarrow$ Max($nextCustomer.TWopening, arrivalTime$)
    $departureTime \leftarrow serviceStart$+ ServiceTime($nextCustomer$)
    if ($serviceStart > nextCustomer.TWclose$) **then**
      $feasible \leftarrow$ false # The vessel arrives after nextCustomers time window has closed
    else
      $feasible \leftarrow$ true
    $newLabel \leftarrow$($serviceStart, departureTime, l.Delays, feasible$)
    $newLabels$.Add($newlabel$)
    if ($l.Delays < \Gamma$) **then** # There is a possibility of adding delay to the trip
      $arrivalTime \leftarrow newlabel.ArrivalTime$+Delay($currentCustomer, nextCustomer$)
      $delays \leftarrow l.Delays + 1$
      $serviceStart \leftarrow$ Max($nextCustomer.TWopening, arrivalTime$)
      $departureTime \leftarrow serviceStart$+ ServiceTime($nextCustomer$)
      if ($serviceStart > nextCustomer.TWclose$) **then**
        $feasible \leftarrow$ false
      else
        $feasible \leftarrow$ true
      $delayedLabel \leftarrow$($serviceStart, departureTime, delays, feasible$)
      $newLabels$.Add($delayedLabel$)
  return $newLabels$

To reduce the time and space consumption of the heuristic, an evaluation of the labels of the nodes is conducted to remove dominated labels. A label l_1 is dominated if, despite being delayed on at least as many trips as another label l_2, it has a service start time no later than the service start time of l_2. In this case, l_1 will never extend into worse cases than l_2, neither having more delays to spend, nor being later having spent the same, and can be removed. As a result, it is not necessary to use memory space storing l_1, or spend time trying to extend it later. Finally, once the end of the route is reached, if no infeasible label was found the route is concluded to be feasible.

4.7 Adaptive large neighbourhood search

The ruin and recreate (R&R) paradigm (Schrimpf et al., 2000) is the basis for the proposed heuristic, with search steps consisting of unassigning and reassigning a relatively large number of nodes of the solutions. In every iteration, one destruction heuristic is selected to remove nodes from the current solution, and then an insertion heuristic is selected to repair it by trying to reinsert all of the nodes that are unassigned. These operations are performed within an adaptive large neighbourhood search (ALNS) heuristic, as described in what follows.

The potential of the R&R paradigm in the ALNS was already shown in Pisinger and Ropke (2005). Their implementation follows the R&R and simply destroys and rebuilds the solutions,

but we have supplemented it with an iterative local search step. In our proposal, the ruin and recreate operations will be performed by using the destruction and insertion operators presented in Sections 4.4-4.3 and the local search with the neighborhoods described in Section 4.5. Furthermore, ALNS is based on the idea that certain heuristics will work better on certain instances of a problem, at the same time as varying the neighbourhoods avoids getting stuck at local optima for a long time. The process usually builds upon different destruction and reconstruction heuristics to achieve a wide variety of neighbourhoods, continuously changing. Scores π_h are kept for each heuristic h, so that the heuristics that perform best can be identified. When a new iteration starts, a roulette wheel selection is used to decide which heuristics to use, where the heuristic's probability of being chosen is proportional to its score, simply $\pi_h / \sum_k \pi_k$.

As a result, one of the most important challenges relating to the implementation of ALNS is figuring out how to update the scores of the heuristics. One option is to use as the score of an iteration the percentage improvement of that iteration. That is, if a destruction heuristic and a construction heuristic together achieve a 3 % improvement, they each get 3 points. A similar reduction in quality deducts 3 points. These scores are either summed from the start, or averaged over the last k iterations. Another interesting alternative is proportional scoring. Under this scoring rule, every heuristic gets its score multiplied by the improvement during the round when it is used. Say, for example that the heuristics improved the solution by 3 %. Then each of the three get their score multiplied by $100/(100-3)$ (essentially divided by 0.97). In addition, we also propose what we called *harsh* and *generous* scoring. These are similar to the last k rule, in that they only consider a subset of all the scores a heuristic has achieved. The harsh scoring rule only deducts points when heuristics perform badly, the idea being that a single heuristic should not get a high score early and then be selected every time thereafter. The generous scoring rule is the opposite and only awards points to well performing heuristics, never deducting points at all. The thought behind this is that no heuristics should be completely eliminated, so as to maintain a level of diversification throughout the run.

To add to this, we have modified the way the scores impact the probability of being chosen. We have supplemented the standard proportional probabilities with *rank scaling* and *Boltzman scaling*. In the first of these two, the probability of being selected is only decided by how you rank in the order of heuristics. That is, the highest scoring heuristic is given probability p_1 of being selected, regardless of its score, while the second highest gets p_2, and so on. These probabilities can be linearly ($p_r - p_{r+1} = c$) or exponentially ($p_r / p_{r+1} = c$) decreasing. When using rank scaling, one hopes to standardize the relations between different heuristics. When heuristics score very evenly, rank scaling gives a larger benefit to the ones that have actually, if only barely, performed over average. When the heuristics end up getting very uneven scores, rank scaling will reduce the deviation and give the low scoring heuristics a somewhat larger chance of being selected. Boltzman selection, on the other hand, rescales the scores according to e^{π_h / t^b}, where t^b is a temperature, similar to simulated annealing (SA) (Kirkpatrick et al., 1983).With this, a high initial temperature makes the difference in scores matter less, while a gradually lowering temperature makes it exponentially more likely for the best scoring heuristic to be selected. This has many of the same benefits as SA, gradually changing the search from a highly diversifying one and to a more intensifying one.

The acceptance rule used to guide the search process is also important. Pisinger and Ropke

(2005) used an acceptance rule similar to the one typically used in SA. However, our proposal is based on the record-to-record travel (R2R) presented in (Li et al., 2007). An R2R local search simply accepts a move that is not worsening the objective value more than some threshold, eliminating the temperature and randomness of the SA. The standard versions of both SA and R2R continue the search from the current solution when the proposed move is not accepted. At this point we propose an alternative, consisting of rolling back the solution to the best known solution (the record) instead when the proposed move is not accepted. Using this scheme, the algorithm repeatedly starts out from the record, and either finds a new record to save, goes above its threshold and is reset, or uses up its iterations and stops. With this simple rule, the heuristic is able to diversify by accepting solutions that are worsening, even at late stages during execution. In addition, it might be stricter than SA in the early stages, focusing more on intensification, and not accepting moves to severely worsening solutions.

Combining all the ideas introduced above, Algorithm 4 shows how the proposed ALNS works.

Algorithm 4 Adaptive large neighbourhood search (ALNS)

input: *solution* - a constructed solution (feasible or infeasible),
$\mathbb{D}$ - a set of destruction heuristics,
$\mathbb{I}$ - a set of insertion heuristics,
$\mathbb{L}$ - a set of local search heuristics,
L^{NI} - maximum number of non-improving iterations,
ScoringRule - a rule for scores heuristics based on performance,
ScalingRule - a rule for scaling scores into probabilities.

1: **procedure** ALNS
2: $scores \leftarrow 1.0$ for each heuristic # How well each heuristic has performed
3: $n^{NI} \leftarrow 0$ # Number of non-improving iterations
4: **while** $(n^{NI} < L^{NI})$ **do**
5: $probabilities \leftarrow$ ScalingRule($scores$)
6: $d \in \mathbb{D}, i \in \mathbb{I} \leftarrow$ SelectHeuristics($probabilities$)
7: $solution \leftarrow d$.Destroy($solution$)
8: $solution \leftarrow i$.InsertUnassignedNodes($solution$)
9: $solution \leftarrow \mathbb{L}$.IterativeLocalSearch($solution$)
10: $scores \leftarrow$ScoringRule($scores, improvement$)
11: $n^{NI} \leftarrow n^{NI} + 1$
12: **if** best solution yet **then**
13: $bestSolution \leftarrow solution$
14: $n^{NI} \leftarrow 0$ # Reset non-improving iterations
15: **return** *bestSolution*

The heuristic framework as a whole is implemented following principles of *programming by optimization* (Hoos, 2012). This means that design choices are delayed as much as possible: instead of trying to manually figure out which components are the useful, this is left as an automated exercise using an appropriate tuning tool. Nevertheless, with the aim of evaluating the effect of several important components of the ALNS of Algorithm 4 on their own, we will

consider three additional versions of the heuristic: one without the preprocessing phase (NoPP), another one without the local search (NoLS), and another one with the standard acceptance rule returning to the previous solution instead of to the best known when the proposed move is not accepted (StdR2R). In the computational study presented in the next section the performance of these four heuristics is discussed and analyzed.

5 Computational Study

This section describes the computational experience carried out to evaluate the performance of the proposed heuristics. All the computational tests presented below were performed on computers with an Intel(R) Core(TM) i7-3770 3.4GHz PC with 16GB installed RAM running Windows 7. The exact solutions were found by running the path inequalities model from (Agra et al., 2013) in Xpress Mosel 3.6.0 using Xpress Optimizer version 26.01.04. The heuristic was implemented using C# 4.0 on .NET framework 4.0, compiled to x64 Release executables.

The problem instances used in the computational study are first described in Section 5.1, whereas the details regarding the tuning of the parameters of the heuristics are provided in Section 5.2 and the main results obtained are analyzed in Section 5.3.

5.1 Instances

The performed computational study is based on three sets of instances. The first set (S_{50}) consists of the same instances as the ones used for testing in (Agra et al., 2012, 2013), which have been used to test the performance of the proposed heuristic against the exact solution based on the mathematical model and also to evaluate the impact of some of the components of this heuristic. The instances from S_{50} have between 10 and 50 customers. The second set (S_{100}) is made up of instances with 100 customers, which have been used to evaluate the performance of the heuristic when applied to larger instances. Last, the third (S_{tune}) is a set of instances that range from ten to 100 customers, which have been used as a learning set to tune the parameters of the algorithms. The two latter sets are generated using the same instance generator as used by Agra et al. (2012, 2013), and we have applied the same pattern for the number of vessels and the steps of Γ as they did.

5.2 Parameter tuning

To tune the parameters of the proposed heuristics we have used the irace software (Lopez-Ibanez et al., 2011). The four heuristic versions considered have been tuned independently using this software. Being an automatic tool for parameter tuning, irace maintains a population of possible parameter configurations, running the implementation with each configuration on instance after instance. After all the configurations have solved an instance, irace uses a statistical test to determine if any of the configurations are significantly worse than the others. Any such configurations are replaced with new ones, generated based on the ones performing best so far. After running the implementation 1000 times, it stops and reports the six best configurations.

During this tuning (and also for the later testing), the global time limit of the framework was set to 300 seconds, while the local time limit was tuned. The global time limit was fixed to

restrict the framework to not use more time than what would be acceptable for use in practice. The parameter α^m introduced in 4.2 was also fixed, in order to restrict the degrees of freedom the tuning would have, as a variation on α^m in itself would be redundant if both π^f and π^w could also be varied at the same time. The remaining relevant parameters were left to be tuned by irace.

5.3 Results

The instances from S_{50} have been used to compare the proposed algorithm against the exact method $\mathcal{T} - PI$ by Agra et al. (2013). To get comparable results, $\mathcal{T} - PI$ has been run on the same computer as the heuristic, and as in (Agra et al., 2013), a time limit of 1800 seconds was set. In all experiments, the global time limit of the heuristic was set to 300 seconds. With these conditions, we obtained comparable data for most of S_{50}. However, on some of the largest instances of S_{50}, and for all of the instances in S_{100}, $\mathcal{T} - PI$ timed out. Therefore, the optimal solutions for the 100-customer instances could not be obtained. However, these instances with 100 customers are nevertheless useful to evaluate the performance of the heuristic on larger instances.

In addition to the comparison against the exact solution, we will use both S_{50} and S_{100} to evaluate the effect of several important components of the proposed heuristic, namely the preprocessing phase, the local search, and the acceptance rule consisting of coming back to the best known every time the current move is rejected. For this purpose, we solved the instances of S_{50} and S_{100} with several versions of the proposed heuristic: the original solution method as described in Algorithm 1 (ALNS) and modified versions without the preprocessing phase (NoPP), without the local search (NoLS), and with the standard acceptance rule returning to the previous solution instead of to the best known when the proposed move is not accepted (StdR2R).

Three different uncertainty levels have been considered. The results obtained on the instances from S_{50} and S_{100} are presented in Tables 1 – 3 for the low, medium, and high uncertainty level, respectively. The pattern in the instances is that the uncertainty level, Γ, varies in the same way as the number of vessels, $|V|$. As an example, let us consider the instances with cargo size, $|N|$, equal to twenty. Here, the number of vessels is one, three, or five, and following the pattern mentioned above, Γ will take on the same three values. Thus, for the instances with twenty cargos, the *Low* uncertainty level corresponds to one possible delay per route ($\Gamma = 1$), the *Medium* uncertainty level is analogous to three possible delays per route ($\Gamma = 3$), and the *High* uncertainty level corresponds to five possible delays per route ($\Gamma = 5$).

In Tables 1 – 3, the number of customers and vessels are given in the first two columns, while the solution time (in seconds) and the percentage gap to the best known solution of each method are given in columns 3 to 7 and the last four columns, respectively. Note that the results on instances from S_{100} are shown in the last three rows, whereas all the other results correspond to instances from S_{50}. The values presented in these tables for each pair of $|N|$ and $|V|$ are averages of values for five instances having the same properties, and the number of instances on which the mathematical program exceeded the time limit of 1800 seconds are marked in a parenthesis. The computational times of these runs count towards the average time. Finally, Table 4 presents the aggregated results of all three uncertainty levels.

From Tables 1 – 4, we see some patterns. For the smallest instances, all versions of the heuristic are able to consistently find the optimal solution. They do not beat the mathematical model on running time, but solving the instances is only matter of a few seconds. For the larger instances in this test set, most of the heuristic versions find solutions that are only marginally worse than the optimal solutions, but now the heuristics run much quicker than the exact solution. Notably, for some of the instances with 40 and 50 customers and all of the instances with 100 customers, the mathematical model times out at 1800 seconds, while the heuristics are able to return before their timeout at 300 seconds in many cases. The solutions generated by the heuristics are generally quite good, and notably, ALNS seems to perform the best for 100-customer instances and similarly to NoPP for instances with at most 50 customers. This indicates that the preprocessing is useful for medium and large instances, while it is less useful for smaller instances. It is also interesting to see that StdR2R seems to show the worst general performance and that noLS is performing consistently worse. This indicates that the effect of the new acceptance rule (coming back to the best known when a move is rejected) is positive, and that using local search generally provides significant improvements. Regarding running time, NoLS is, in general, the fastest method, which is not surprising due to the computational effort typically required by the local search. StdR2R seems to show the worst performance also regarding running time, confirming the importance of the move acceptance rule. Regarding the uncertainty levels, it seems that the running times of all algorithms slightly increase with the uncertainty level, but the quality of the solutions do not change significantly. Higher uncertainty levels lead to harder problems, but not by much.

6 Conclusions

Most of the existing research in transportation and logistics deals with deterministic problems. This is in disaccord with real life, where decision makers frequently deal with uncertain outcomes of future events. This paper focused on a problem from maritime shipping formulated as a robust vehicle routing problem with hard time windows, in which travel times are considered uncertain due to potential delays based on weather conditions, port congestion or strikes, or mechanical problems. The robustness criterion used implies that a solution must be feasible for any possible set of combinations of delays within a budget polytope uncertainty set.

The robust vehicle routing problem with time windows (RVRPTW) was previously formulated and solved using exact methods. These methods, however, are time consuming and unable to solve large realistic instances. In order to overcome this limitation, this paper has focused on the development of an efficient heuristic, based on adaptive large neighborhood search. Following the concept of programming by optimization, the heuristic was developed by implementing a large variety of search components, followed by the use of automatic parameter tuning tools to determine a final set of components and settings. The performance of the proposed heuristic and three simplified variants, in comparison with the best existing exact method, was evaluated through an extensive computational study.

For small instances, all versions of the heuristic are able to consistently find the optimal solutions, even though the mathematical model is faster. However, for medium sized instances, most of the heuristic versions found solutions that are very close to the optimal solutions, and

				Avg. solution time in seconds				Avg. gap to best known			
$\|N\|$	$\|V\|$	$(\mathcal{T}-PI)$		ALNS	NoLS	NoPP	StdR2R	ALNS	NoLS	NoPP	StdR2R
10	1	0		1	1	1	1	0.0 %	0.0 %	0.0 %	0.0 %
	3	0		3	3	3	6	0.0 %	0.0 %	0.0 %	0.0 %
	avg	0		2	2	2	4	0.0 %	0.0 %	0.0 %	0.0 %
20	1	0		3	6	5	8	0.0 %	0.0 %	0.0 %	0.0 %
	3	1		7	4	7	14	0.0 %	0.0 %	0.0 %	0.1 %
	5	0		14	6	11	24	0.0 %	0.5 %	0.0 %	0.3 %
	avg	1		8	5	8	15	0.0 %	0.2 %	0.0 %	0.1 %
30	1	2		10	27	23	34	0.0 %	0.0 %	0.0 %	0.0 %
	4	1		21	9	24	33	0.1 %	0.4 %	0.1 %	1.0 %
	7	4		40	13	51	62	0.1 %	0.7 %	0.0 %	1.2 %
	avg	2		24	17	32	43	0.0 %	0.4 %	0.0 %	0.7 %
40	1	1		25	62	58	76	0.0 %	0.0 %	0.0 %	0.0 %
	5	14		42	23	50	74	0.5 %	1.1 %	0.5 %	1.8 %
	9	397	(1)	77	23	120	142	0.4 %	2.3 %	0.3 %	2.0 %
	avg	137		48	36	76	97	0.3 %	1.1 %	0.3 %	1.3 %
50	1	14		54	127	128	164	0.0 %	0.0 %	0.0 %	0.0 %
	6	137		82	48	110	161	0.9 %	2.2 %	0.7 %	3.0 %
	11	780	(2)	188	38	267	285	1.2 %	3.8 %	1.1 %	3.1 %
	avg	311		108	71	168	203	0.7 %	2.0 %	0.6 %	2.0 %
100	11	NA	(5)	300	300	300	300	0.0 %	0.0 %	0.2 %	0.8 %
	21	NA	(5)	300	292	300	300	0.1 %	0.3 %	0.5 %	0.7 %
	avg	NA		300	294	300	300	0.1 %	0.3 %	0.4 %	0.7 %

Table 1: Low uncertainty level

			Avg. solution time in seconds				Avg. gap to best known			
$\|N\|$	$\|V\|$	$(\mathcal{T}-PI)$	ALNS	NoLS	NoPP	StdR2R	AT	NoLS	NoPP	StdR2R
20	1	0	3	7	7	9	0.0 %	0.0 %	0.0 %	0.0 %
	3	2	7	5	7	13	0.0 %	0.0 %	0.0 %	0.0 %
	5	1	11	6	11	19	0.0 %	0.2 %	0.0 %	0.6 %
	avg	1	7	6	8	14	0.0 %	0.1 %	0.0 %	0.2 %
30	1	1	17	35	36	45	0.0 %	0.0 %	0.0 %	0.0 %
	4	2	19	10	23	31	0.5 %	0.8 %	0.2 %	0.7 %
	7	10	51	12	49	56	0.0 %	1.0 %	0.3 %	1.2 %
	avg	4	29	19	36	44	0.2 %	0.6 %	0.1 %	0.6 %
40	1	2	42	94	92	115	0.0 %	0.0 %	0.0 %	0.0 %
	5	66	42	26	58	67	0.9 %	0.9 %	0.5 %	1.9 %
	9	387 (1)	81	23	110	127	0.6 %	2.5 %	0.4 %	2.0 %
	avg	152	55	48	87	103	0.5 %	1.1 %	0.3 %	1.3 %
50	1	31	99	185	195	219	0.0 %	0.0 %	0.0 %	0.0 %
	6	763 (2)	70	48	98	168	0.9 %	1.3 %	0.7 %	2.9 %
	11	1163 (3)	179	46	267	285	0.9 %	3.0 %	1.0 %	3.1 %
	avg	653	116	93	187	224	0.6 %	1.4 %	0.6 %	2.0 %
100	11	NA (5)	300	300	300	302	0.1 %	0.8 %	0.8 %	0.8 %
	21	NA (5)	300	279	300	300	0.1 %	0.6 %	0.6 %	1.0 %
	avg	NA	300	289	300	301	0.1 %	0.7 %	0.7 %	0.9 %

Table 2: Medium uncertainty level

				Avg. solution time in seconds				Avg. gap to best known			
$\|N\|$	$\|V\|$	$(\mathcal{T}-PI)$		ALNS	NoLS	NoPP	StdR2R	AT	NoLS	NoPP	StdR2R
10	1	0		1	1	1	1	0.0 %	0.0 %	0.0 %	0.0 %
	3	0		3	3	3	6	0.0 %	0.0 %	0.0 %	0.0 %
	avg	0		2	2	2	3	0.0 %	0.0 %	0.0 %	0.0 %
20	1	0		3	7	7	9	0.0 %	0.0 %	0.0 %	0.0 %
	3	2		6	4	7	13	0.0 %	0.0 %	0.0 %	0.0 %
	5	1		12	6	12	24	0.1 %	0.1 %	0.0 %	0.2 %
	avg	1		7	6	8	15	0.0 %	0.0 %	0.0 %	0.1 %
30	1	3		19	40	41	50	0.0 %	0.0 %	0.0 %	0.0 %
	4	2		19	11	22	31	0.2 %	0.6 %	0.0 %	0.9 %
	7	10		43	13	45	65	0.2 %	1.5 %	0.2 %	1.4 %
	avg	5		27	21	36	49	0.1 %	0.7 %	0.1 %	0.8 %
40	1	19		53	119	107	122	0.0 %	0.0 %	0.0 %	0.0 %
	5	73		42	25	50	76	1.0 %	1.9 %	1.1 %	2.0 %
	9	401	(1)	74	23	114	132	0.4 %	2.1 %	0.4 %	2.2 %
	avg	164		56	56	90	110	0.5 %	1.3 %	0.5 %	1.4 %
50	1	48		110	196	203	229	0.0 %	0.0 %	0.0 %	0.0 %
	6	757	(2)	76	43	93	154	1.1 %	2.1 %	0.8 %	3.0 %
	11	1147	(3)	187	43	259	282	0.7 %	2.9 %	1.0 %	2.4 %
	avg	651		124	94	185	222	0.6 %	1.7 %	0.6 %	1.8 %
100	11	NA	(5)	300	300	300	303	0.0 %	0.9 %	1.2 %	0.4 %
	21	NA	(5)	300	289	300	300	0.0 %	0.5 %	0.5 %	0.6 %
	avg			300	295	300	302	0.0 %	0.7 %	0.8 %	0.5 %

Table 3: High uncertainty level

	Avg. solution time in seconds				Avg. gap to best known			
	ALNS	NoLS	NoPP	StdR2R	ALNS	NoLS	NoPP	StdR2R
$10 \leq \|N\| \leq 50$	46	35	69	86	0.3 %	0.8 %	0.2 %	0.9 %
$\|N\| = 100$	300	292	300	301	0.1 %	0.6 %	0.7 %	0.7 %

Table 4: All instances

were generally faster than the mathematical model. Furthermore, the exact method failed to solve some of the instances with 40 and 50 customers, as well as all of the instances with 100 customers, within the given time limit, while the heuristics were able to provide good solutions for all of them.

Furthermore, the effect of three of the main components of the heuristic, namely the preprocessing phase, the local search, and the new acceptance rule, was also analyzed. The experiments indicate that the preprocessing could help for instances with at least 40 customers, while its importance is negligible for smaller instances. On the other hand, both the local search and the new acceptance rule showed to be quite effective and provided significant improvements without requiring additional computing time.

Acknowledgements

This research was carried out with financial support from the Government of Spain, grant MTM2015-65803-R, and the Government of Madrid, grant CASI-CAM S2013/ICE-2845.

References

Y. Adulyasak and P. Jaillet. Models and algorithms for stochastic and robust vehicle routing with deadlines. *Transportation Science*, 50:1–36, 2014.

A. Agra, M. Christiansen, R. Figueiredo, L.M. Hvattum, M. Poss, and C. Requejo. Layered formulation for the robust vehicle routing problem with time windows. *Lecture Notes in Computer Science*, 7422:249–260, 2012.

A. Agra, M. Christiansen, R. Figueiredo, L.M. Hvattum, M. Poss, and C. Requejo. The robust vehicle routing problem with time windows. *Computers and Operations Research*, 40:856–866, 2013.

A. Ben-Tal and A. Nemirovski. Robust solutions of linear programming problems contaminated with uncertain data. *Mathematical Programming*, 88(3):411–424, 2000.

A. Ben-Tal, L. El Ghaoui, and A. Nemirovski. *Robust optimization*. Princeton University Press, 2009.

D. Bertsimas and M. Sim. The price of robustness. *Operations Research*, 52(1):35–53, 2004.

O. Bräysy and M. Gendreau. Vehicle routing problem with time windows, Part I: Route construction and local search algorithms. *Transportation Science*, 39(1):104–118, 2005.

M. Christiansen, K. Fagerholt, B. Nygreen, and D. Ronen. Ship routing and scheduling in the new millennium. *European Journal of Operational Research*, 228:467–483, 2013.

M. Dror and P. Trudeau. Stochastic vehicle routing with modified savings algorithm. *European Journal of Operational Research*, 23(2):228–235, 1986.

M. Gendreau, G. Laporte, and R. Séguin. Stochastic vehicle routing. *European Journal of Operational Research*, 88(1):3–12, 1996.

C.E. Gounaris, W. Wiesemann, and C.A. Floudas. The robust capacitated vehicle routing problem under demand uncertainty. *Operations Research*, 61(3):677–693, 2013.

C. Groër, B. Golden, and E. Wasil. A library of local search heuristics for the vehicle routing problem. *Mathematical Programming Computation*, 2(2):79–101, 2010.

H.H. Hoos. Programming by optimization. *Communications of the ACM*, 55:70–80, February 2012.

A.S. Kenyon and D.P. Morton. Stochastic vehicle routing with random travel times. *Transportation Science*, 37(1):69–82, 2003.

S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

C. Lee, K. Lee, and S. Park. Robust vehicle routing problem with deadlines and travel time/demand uncertainty. *Journal of the Operational Research Society*, 63:1294–1306, 2012.

F. Li, B. Golden, and E. Wasil. A record-to-record travel algorithm for solving the heterogeneous fleet vehicle routing problem. *Computers and Operations Research*, 34:2734–2742, 2007.

M. Lopez-Ibanez, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace package: Iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Universite Libre de Bruxelles, Belgium, 2011.

F. Ordóñez. Robust vehicle routing. *TUTORIALS in Operations Research*, pages 153–178, 2010.

I.H. Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41(4):421–451, 1993.

D. Pisinger and S. Ropke. A general heuristic for vehicle routing problems. *Computers and Operations Research*, 34:2403–2435, 2005.

G. Schrimpf, J. Schneider, H. Stamm-Wilbrandt, and G. Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000.

E. Solano-Charris, C. Prins, and A.C. Santos. Local search based metaheuristics for the robust vehicle routing problem with discrete scenarios. *Applied Soft Computing*, 32:518–531, 2015.

M.M. Solomon. Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints. *Operations Research*, 35:254–265, 1987.

S. Souyris, C.E. Cortés, F. Ordóñez, and A. Weintraub. A robust optimization approach to dispatching technicians under stochastic service times. *Optimization Letters*, 7(7):1549–1568, 2013.

W.E. Stewart Jr. and B.L. Golden. Stochastic vehicle routing: A comprehensive approach. *European Journal of Operational Research*, 14(4):371–385, 1983.

I. Sungur, F. Ordónez, and M. Dessouky. A robust optimization approach for the capacitated vehicle routing problem with demand uncertainty. *IIE Transactions*, 40(5):509–523, 2008.

P. Toth and D. Vigo. The granular tabu search and its application to the vehicle-routing problem. *INFORMS Journal on Computing*, 15:333–346, 2003.

P. Toth and D. Vigo. *Vehicle routing: problems, methods, and applications*. Siam, 2014.

T. Vidal, T. G. Crainic, M. Gendreau, and C. Prins. Heuristics for multi-attribute vehicle routing problems: A survey and synthesis. *European Journal of Operational Research*, 231: 1–21, 2013.