A Matheuristic Approach to the Pickup and Delivery Problem with Time Windows

Carlo S. Sartori(✉) and Luciana S. Buriol

Instituto de Informática,
Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
{cssartori,buriol}@inf.ufrgs.br

Abstract. In this work, the Pickup and Delivery Problem with Time Windows is studied. It is a combinatorial optimization problem, in which the objective is to construct the best set of vehicle routes while respecting side constraints, such as precedence between locations to be visited, and the time to service them. To tackle this problem, a matheuristic based on Iterated Local Search method is proposed, with an embedded Set Partitioning Problem that is iteratively solved to recombine routes of previously found solutions. Results indicate the approach works well for a standard benchmark set of instances from the literature. A number of new best-known solutions has been found.

Keywords: Matheuristic · Pickup and delivey problem
Time windows · Iterated local search

1 Introduction

The *Vehicle Routing Problem* (VRP) is a well-known combinatorial optimization problem used to model scenarios on transportation and logistics, with significant economic importance [10]. Its objective is to build the best set of vehicle routes to service a set of requests, such that all side constraints are satisfied. Given the number of real-world applications, several models and variations have been proposed to best describe each particular scenario.

One such variation is the *Pickup and Delivery Problem* [20], in which requests are pairs of pickup and delivery locations. Goods have to be transported from the pickup location to the corresponding delivery location, by the same vehicle route. Constraints include the pickup to be serviced before its delivery, and the maximum capacity of vehicles, which should not be exceeded.

When locations have to be serviced within a given period, the problem is known as *Pickup and Delivery Problem with Time Windows* (PDPTW) [8]. In this case, vehicles cannot start service before the beginning, or after the end of the time window of a given location.

In this work, the PDPTW is considered. It is a $\mathcal{NP}$-Hard problem, which generalizes both the *Classical Vehicle Routing Problem* and the *Vehicle Routing Problem with Time Windows*. Additionally, it can be applied to various real-world

R. Cerulli et al. (Eds.): ICCL 2018, LNCS 11184, pp. 253–267, 2018.
https://doi.org/10.1007/978-3-030-00898-7_16

scenarios, such as product delivery, bulk product transportation, dial-a-ride problems, courier services, airline scheduling, bus routing, and logistics and maintenance support [15]. Thus, it is reasonable to consider that efficient methods to solve the PDPTW can be used for related problems in transportation.

This work proposes the study of a hybrid method, using metaheuristic and exact solutions of a *Set Partitioning Problem* (SPP) to solve the PDPTW. Such combination is referred in the literature as *matheuristic* [5]. The metaheuristic components are based on the work of [7,19], and include the *Adaptive Guided Ejection Search* and the *Large Neighborhood Search*. They are embedded into an *Iterated Local Search* framework [13] with a SPP solver. Moreover, a tuning procedure is performed to decide the best parameter values and components to be used. Experimental results show this approach performs well when compared to a *state-of-the-art* method.

The remainder of the article goes as follows. In Sect. 2 the PDPTW is formally defined, and basic notations are introduced. A literature review is presented in Sect. 3. Our proposed method is detailed in Sect. 4, and experiments and results are discussed in Sect. 5. The work is concluded in Sect. 6.

2 Problem Definition

An instance of the PDPTW is defined on a graph $G = (V, E)$, where V is the set of $n = |V|$ vertices and E the set of edges. Let $P \subset V$ be the set of pickup locations, and set $D \subset V$ be the delivery locations, $P \cap D = \emptyset$, and the set of requests is $R = P \cup D$. There is a single depot denoted by 0, and $V = \{0\} \cup R$. A request is a pair (p, d), where each $p \in P$ has only one paired $d \in D$. Edges (i, j) have an associated cost c_{ij}, and time t_{ij} to travel between locations i and j.

Every location $i \in V$ has a time window of the form $[e_i, l_i]$, where e_i is the earliest time service can start at location i, and l_i is the latest time. There is also an associated waiting time w_i, or how long the vehicle takes to complete the service at that location. By definition, the time window of the depot has values $e_0 = 0$, and l_0 as the maximum time a route can take.

A homogeneous fleet of vehicles is available at the depot, from where they start and end their routes. Every vehicle has an associated maximum capacity Q, which is the maximum amount of load it can carry at once. Each location has a demand, or the total amount of goods to be picked up or delivered. Given any pickup and delivery pair (p, d), their demands q are such that $q_d = -q_p$, and pickups are non-negative, $q_p \geq 0$. In other words, everything that has been collected has to be dropped at the corresponding location.

A solution to the PDPTW is a set of routes $s = \{r_1, \ldots, r_m\}$. Each route is a sequence of locations to be visited, starting and ending at the depot, denoted by $r = \{v_0, v_1, \ldots, v_h, v_{h+1}\}$, where $v_k \in R, k = 1, \ldots, h$, $v_0 = v_{h+1} = 0$, and each location is visited at most once. A solution is feasible if all requests are serviced exactly once, and all routes respect the side constraints.

For any request (p, d), if the pickup location p belongs to the route, then the delivery location d must also belong to the same route. Additionally, the pickup

must always precede the delivery in the path. These are known as pairing and precedence constraints, respectively.

Accumulated load carried by a vehicle up to the k-th location is given by $a_{v_k} = \sum_{i=0}^{k} q_{v_i}$. Capacity constraints state that $a_{v_k} \leq Q, \forall k = 1, \ldots, h$. A vehicle leaves and returns to the depot empty, so $a_{v_0} = a_{v_{h+1}} = 0$.

Time window constraints are defined in terms of b_{v_k}, the time a vehicle starts service at location v_k. The start of service at the next location in route is given by $b_{v_{k+1}} = \max(e_{v_{k+1}}, b_{v_k} + w_{v_k} + t_{v_k,v_{k+1}})$. In other words, vehicles may arrive before the beginning of a time window, but must wait until service can start. It is required that $b_{v_k} \leq l_{v_k}, \forall k = 0, \ldots, h+1$. By definition, $b_{v_0} = 0$, that is, routes start as soon as possible.

The objective function to be minimized is evaluated according to the lexicographic order of its terms and is given by:

$$f(s) = (\ |s|, \sum_{r \in s} C(r)\ )$$

The first term minimizes the number of routes or vehicles used. Ties are broken in the evaluation by the second term, which is the accumulated cost of all routes in solution s. The cost of a route r is the sum of all edges from the input graph that belong to r and is denoted by $C(r)$.

3 Related Works

A survey of both exact and heuristic solution methods for pickup and delivery problems can be found in the work of Parragh et al. [16]. The authors explain how pickup and delivery problems can be studied as two different groups. One is the non-paired variant, where vehicles must deliver products from the depot to some customers, and collect products from customers to the depot. Another variant is the paired, where products must be transported between locations, which is precisely how the PDPTW is defined.

Due to its difficulty, the most common approach applied to solve the problem is metaheuristic. Nanry and Barnes [15] proposed the first metaheuristic to address the multiple vehicle version of the PDPTW. Their method was a Reactive Tabu Search, and defined three standard local search neighborhood movements: shift a request between routes, swap two requests between routes, and relocate a request within its route.

Li and Lim [11] proposed the current standard benchmark set of instances for the problem. The authors applied a Tabu-Embedded Simulated Annealing procedure to solve the problem, with the same neighborhoods of [15].

However, a method that has been successfully applied to the PDPTW is *Large Neighborhood Search* (LNS). Bent and Van Hentenryck [4] proposed a two-stage algorithm to solve the problem, where the first stage used *Simulated Annealing* to reduce the number of vehicles, and the second stage performed LNS to reduce the cost. Likewise, Ropke and Pisinger [19] applied a two-stage *Adaptive Large Neighborhood Search*, where both phases used the same LNS algorithm with

different weights on the objective function. The latter improved almost half of the best-known solutions of the standard benchmark instances [11].

A particular case of the PDPTW, where only vehicle minimization was taken into account, was studied by Nagata and Kobayashi [14]. The method was called *Guided Ejection Search* (GES), and is based on the idea of randomly removing a route of a solution, and trying to reinsert all its requests into the remaining routes, possibly ejecting requests to make space for them. Their results showed the method was efficient in reducing the number of routes, obtaining new results for the standard benchmark set.

Curtois et al. [7] proposed a hybridization of LNS and a modification of the GES heuristic, called *Adaptive Guided Ejection Search* (AGES). The method was able to improve many instances of the standard benchmark set, and can be considered the current *state-of-the-art* algorithm to solve the PDPTW. This algorithm is the base for the method presented in this paper.

Among exact approaches, branch-and-price, and branch-and-cut are the main used algorithms. Applications of such methods include [3,8,18]. The algorithm of Baldacci et al. [3] was able to solve to optimality all instances from the set previously proposed by [18]. Although, none of the exact methods is able to cope with medium and large sized instances, such as the ones from [11] benchmark set, corroborating the use of heuristics.

Regarding matheuristics, it has been shown that they can be an effective method to solve combinatorial optimization problems [5]. Applications to VRPs include the variations with time windows [1], cross-docking [9], and to solve a number of routing problems [24]. Additionally, a similar method to the one presented in this work has been proposed to the dial-a-ride problem [17], although the heuristic components are not the same, and the objective of the problems differs slightly. A survey on matheuristics applied to VRPs can be found in [2].

All cited matheuristic methods are SPP based, which works as a column generation, where each column is a valid VRP route. Routes are generated by running a metaheuristic and storing the routes of new local optimum solutions on a pool to be used by the SPP model. Even though solutions for many of the referred works are of high quality, there is no guarantee of optimality.

4 Proposed Algorithm

Our algorithm is based on that of Curtois et al. [7]. It combines AGES and LNS to generate solutions as proposed in [7] and extends the approach to iteratively recombine routes of previous local minima through a SPP model.

Those components are embedded into an ILS framework, to allow perturbation of solutions, and acceptance of different local minima to continue the search. It is henceforth referred as IGLS and described in Algorithm 1. The current solution is denoted by s, and the best found by s^*.

In line 1 an initial solution is generated by a greedy constructive algorithm. A pool $\mathcal{P}$ of routes is initialized in line 2 and is used to create the associated SPP model. Line 3 initializes the perturbation size z_p, which changes dynamically

during the execution. Then, the main loop (lines 4–19) is repeated until a given stopping condition is reached. The best solution s^* is returned in line 21.

Algorithm 1 IGLS: ILS+GES+LNS+SPP

Input: Instance with graph $G = (V, E)$; set $R \subset V$ is the set of requests
Parameters: α, A, M_g, Z_g, p_{shift}, K, M_l, L, p_{shaw}
1: $s, s^* \leftarrow$ initial_solution()
2: $\mathcal{P} \leftarrow$ initialize_pool$(s, \mathcal{P})$
3: $z_p \leftarrow \lceil \alpha \cdot |R| \rceil$
4: **repeat**
5: $s \leftarrow$ AGES$(s,\ M_g,\ Z_g,\ p_{shift})$
6: $s \leftarrow$ LNS$(s,\ K,\ M_l,\ L,\ p_{shaw})$
7: $\mathcal{P} \leftarrow$ update_pool$(s,\ \mathcal{P})$
8: $s^p \leftarrow$ solve_SPP$(\mathcal{P})$
9: **if** $f(s^p) < f(s^*)$ **then**
10: $s^* \leftarrow s^p$
11: $z_p \leftarrow \lceil \alpha \cdot |R| \rceil$
12: $count \leftarrow 0$
13: **else**
14: $s \leftarrow$ accept_solution$(s,\ s^*,\ count,\ iter)$
15: $s \leftarrow$ perturb$(s,\ z_p,\ p_{shift})$
16: $z_p \leftarrow \min(count \cdot \lceil \alpha \cdot |R| \rceil, \lceil A \cdot |R| \rceil)$
17: $count \leftarrow count + 1$
18: **end if**
19: $iter \leftarrow iter + 1$
20: **until** stopping condition
21: **return** s^*

At each iteration *iter* of the loop, the AGES is executed to reduce the number of routes in solution s. Next, the LNS heuristic is used to reduce the cost of the solution s. Lines 5–6 update the pool of routes, and call the SPP model to solve the problem over the new pool $\mathcal{P}$, returning the best combination of routes s^p. If the objective function $f(s^p)$ is *strictly* better than $f(s^*)$, solution s^p is accepted in line 10, else a perturbation is performed. In line 14, Solution s^* is chosen to be perturbed with probability $(iter - count)/iter$, otherwise solution s is chosen. The intention is to intensify the search when few iterations without improvement have passed but to diversify the search when too many iterations have been executed without any update of the solution s^*.

When a new best solution is found, the counter *count* of iterations without improvement, and the size of the perturbation z_p, are reset (lines 11–12). If s^p does not improve s^*, *count* is incremented, and z_p is increased to $count \cdot \lceil \alpha \cdot |R| \rceil$, up to a maximum value $\lceil A \cdot |R| \rceil$ (lines 16–17). That is, larger perturbations are performed whenever IGLS seems trapped in a local minimum.

While the two-stage algorithms of [4,19] first execute one vehicle minimization phase, and then a cost reduction phase, the proposed method, IGLS, and that of [7], perform both stages at every iteration. This procedure leads to a more intensive search, especially regarding vehicle minimization, since after reducing the total cost there may be more opportunities to reduce the number of routes. Results obtained by the two latter methods corroborate this hypothesis.

4.1 Greedy Constructive Algorithm

In order to create an initial solution, a simple greedy constructive algorithm based on [23] is used in line 1 of IGLS. The procedure creates one route at a time, and at each iteration, it tries to insert all unrouted requests into the current route, into all possible positions within it. The request and position, which are both feasible and minimize the total cost are chosen. If no request can be inserted in the current route, a new one is created. This continues until there are no more unrouted requests, or a request cannot be inserted anywhere.

This constructive algorithm does not take into account a maximum number of available vehicles, since the next steps of the algorithm may significantly reduce the number of routes. If a feasible solution cannot be built with this procedure, even when every request is serviced by a separated route, the instance is not feasible due to capacity or time window constraints.

4.2 Adaptive Guided Ejection Search (AGES)

The AGES heuristic [7], in line 5 of IGLS, is the component most responsible for vehicle minimization. At first, a random route r is removed from the current solution s. All requests that belong to r are inserted into a stack, for Last-In-First-Out (LIFO) ordering. This generates a partial solution s'.

Next, a request u is removed from the stack and inserted in a random position in s', from every possible insertion of u in the partial solution. If request u was successfully reinserted in s', then the algorithm moves to the next request in the stack. Otherwise, if there is no possible insertion position for u in s', the procedure tries to open space for u in s' by ejecting some requests.

Requests are selected to be ejected from s' based on a simple heuristic. For every request $x \in R$, a penalty counter is associated, denoted ρ_x. Whenever a request u is not able to be reinserted into the partial solution, requiring others to be ejected, ρ_u is increased. This means requests that are hard to be reinserted will probably have a high penalty counter. Thus, it seems reasonable to assume that such requests should remain where they currently are.

Then, the ejection heuristic chooses a number k of requests to remove, minimizing the sum of their penalties. In other words, it chooses k requests that are more likely to be reinserted later. Due to the large number of possibilities to choose when k grows, the heuristic only tries for $k = 1$, and if no space could be opened for request u, for $k = 2$. If it still cannot reinsert u, the current AGES iteration is aborted. The k removed requests are inserted in the stack.

After an ejection procedure and the following reinsertion of u, the resulting solution is perturbed using the procedure detailed in Sect. 4.4. The number of AGES perturbation movements is fixed, and denoted by Z_g. A counter $count_g$ is increased for every perturbation movement, and reset whenever a new partial solution with a smaller number of unassigned requests is found.

If at a given iteration the stack is empty, a route has been successfully removed from the original solution s. Then, s' is a full solution and $f(s') < f(s)$. In such case, the heuristic tries to remove another route. The procedure

terminates when $count_g > M_g$ perturbation movements were executed without finding s' with less unrouted requests. This stopping condition tries to keep AGES running for as long as it appears to make progress.

4.3 Large Neighborhood Search (LNS)

Large Neighborhood Search has been shown to perform well on VRPs in general. It differs from the standard local search because it changes large portions of the solution, instead of a few components. In the PDPTW, for example, it has been used to remove a possibly large number of requests and reinsert them [19], while local search moves only moved one or two requests at a time [11,15].

The LNS used in line 6 of IGLS is the same of [7]. It has two removal heuristics: random removal [19], and shaw removal [21]. In the first, requests are randomly removed from solution s. In the second, requests are removed according to a relatedness measure, which takes into account the distance, demand and current service time of the requests. This relatedness between two requests is estimated with the same function proposed in [19], including the weights of each term. Requests considered related to already removed requests are more likely to be chosen. Though, the first request is selected at random.

In every iteration one of the two heuristics is selected, the shaw with probability p_{shaw}. The number of removed requests is chosen randomly in the interval defined by $[K, |R|/2]$, where K is a parameter. It is reasonable to expect that the more intelligent shaw removal has a higher probability of being chosen.

In order to reinsert the requests removed, the regret heuristic is used [19]. This heuristic uses a look-ahead and estimates how much would be lost if a request was not inserted in its best route, by considering its k best routes for insertion in the current solution. This way, requests that can only be inserted in a few routes will be inserted first, minimizing regrets on the decision. The look-ahead size k is chosen randomly from $\{1, 2, 3, 4, 5, |s|\}$.

A partial solution s' is generated from the original solution s when requests are removed. If some requests cannot be reinserted into s', the procedure is aborted and it continues with s, whereas if all requests were reinserted generating new solution s'', two options arise. When $f(s'') < f(s)$, s'' is accepted and the search continues from it, while if $f(s'') \geq f(s)$, an acceptance criterion based on *Late Acceptance Hill Climbing* (LAHC) [6] is applied. A list of size L is kept with values of visited solutions, and when to verify if a given solution s'' should be accepted, the algorithm checks previously stored values to decide. With LAHC, LNS may accept non-improving solutions to continue the search, allowing it to move out from certain basin of attractions more easily.

The LNS stopping conditions differ from the ones used in [7]. It ends after a number M_l of iterations without improvement, or after $10,000$ iterations overall, whichever happens first. A strict maximum number of iterations has been imposed because, in larger instances, the method spends too much time in the LNS when it would be preferable to continue and further reduce the number of vehicles. Thus, it avoids long unnecessary computation times, and as the results seem to indicate it has no negative influence on the final solution quality.

4.4 Perturbation

Solutions are perturbed by two neighborhood movements: random shift and random swap. A number of maximum moves is given as a parameter for the perturbation. Just as in [14], if a given movement cannot be applied, because would lead to an infeasible solution, it is aborted, and the next one is to be tried. Each movement selects one of two neighborhoods with a given probability. The probability of selecting random shift is given by p_{shift}.

In the random shift, two routes r_1 and r_2 are randomly picked from s. A random request u is removed from r_1 and inserted at a random feasible position in route r_2. While in the random swap, two routes r_1 and r_2 are randomly picked from s. A random request u_1 is removed from r_1, and another random request u_2 is removed from r_2. Then, u_1 is inserted in its best position in route r_2, and the same follows for u_2 in route r_1.

This procedure is used for both AGES and as the perturbation of the overall ILS in line 15 of IGLS. Although, perturbation sizes are different.

4.5 Set Partitioning Problem (SPP)

Vehicle Routing Problems can be formulated as a SPP in Integer Linear Programming form, as follows. Let $\mathcal{R}$ be the set of all feasible routes in a given VRP problem. Denote by a binary value $\lambda_{ir}, i \in V, r \in \mathcal{R}$ whether node i belongs to route r. Binary variable y_r assumes 1 when route r is used in solution, and zero otherwise. Thus,

$$\begin{aligned} \min \quad & \sum_{r \in \mathcal{R}} C(r) y_r \\ \text{s.t.} \quad & \sum_{r \in \mathcal{R}} \lambda_{ir} y_r = 1, \; i = 1, ..., n-1 \\ & y_r \in \{0, 1\}, \quad \forall r \in \mathcal{R} \end{aligned}$$

The objective function is to minimize the summed costs of all selected routes. The only constraint is that each request should be visited only once. Given that, the PDPTW can be formulated as a SPP as long as the routes in $\mathcal{R}$ respect all constraints defined in Sect. 2. In order to minimize the number of routes, a large constant value is assigned to every route, as the cost to schedule a vehicle.

However, the size of set $\mathcal{R}$ is exponential on the instance size, and so impractical to use. A common procedure is to somehow generate only routes that are of interest to find reasonable solutions [2]. In the case of matheuristics, this is done employing a (meta)heuristic. Although, there is no guarantee of optimality.

In this work, we denote the set $\mathcal{R}$ as $\mathcal{P}$, also called *pool of routes*. It contains routes that belong to local minima found during the algorithm search. At each iteration of IGLS, routes of the new local minimum solution are added to $\mathcal{P}$ in line 7, and the model is solved in line 8.

Computationally, the set $\mathcal{P}$ is a map, and the key is a set of requests attended by the mapped route. If two routes r_1 and r_2 have the same set of serviced requests, regardless of their visit order, and $C(r_1) < C(r_2)$, the pool will only store route r_1, since it serves the same requests with lower cost. This way the

number of stored routes is reduced, and so is memory consumption. In fact, the pool of routes remained small enough for the SPP to be solved up to optimality at every iteration with little impact on computation times.

5 Computational Experiments

This section presents and discusses the computational results of our proposed algorithm. A replication of the method of [7] is also analyzed. All experiments have been done using the standard set of benchmark instances of [11], which is available online and maintained by SINTEF [22].

There are 354 instances separated into six groups according to the number of locations in the input graph. These sizes are: 100, 200, 400, 600, 800, and 1000. Each size is divided into groups depending on how locations are distributed, they can be: clustered (`LC`), random (`LR`), or a combination of both (`LRC`). Instances are also grouped based on the size of the planning horizon into two types: (1) with a shorter horizon; and (2) with a longer horizon.

For each instance, we have run the algorithms multiple times, due to their stochastic components. Instances of sizes 100, 200, 400, and 600 were executed ten times each, with maximum running time per execution of 300, 900, 900, and 1800 seconds, respectively. Larger instances of sizes 800 and 1000 were executed only five times, with 3600 seconds each. These running times were used by [7]. Separate runs used different random seeds.

All implementations were done in `C++`, and compiled using `g++` with `-O3` optimization flag. The associated SPP is solved using CPLEX 12.6.2 API. A computer equipped with an Intel i7 930 @ 2.8 GHz processor, 12 GB of RAM, and Ubuntu 16.04 LTS operating system was used to perform the experiments. The algorithms and CPLEX were run in single thread mode.

Due to the limited space, we were not able to provide full results for each instance. Though, complete tables are available upon request.

5.1 Reimplementation of State-of-the-Art

A replication of the algorithm proposed in [7] was needed to better support the findings in our work because the IGLS is based on the main components of the former, i.e., AGES and LNS. Additionally, results reported by [7] contained information only for a single execution of the algorithm. As the method comprises many stochastic components, results from a single execution can be misleading for certain conclusions, and hard to compare with, since there is usually a deviation in results between two separate runs.

Henceforth, the method of [7] is denoted by the acronym CLSQL[1]. Its approach has been implemented following the original article, and the same set of parameter values has been used. We denote our reimplementation as R-CLSQL.

Table 1 presents the results from our replication. It is usual in the literature of the PDPTW to present results as the accumulated values of all instances for

[1] Initials from the last name of the authors.

Table 1. Results of the replication. CLSQL is one run per instance, and R-CLSQL is the average of 10 or 5 runs as previously informed.

Inst. Size	CLSQL			R-CLSQL			
	$\#V$	Cost	t(s)[a]	$\#V$	σ_v	Cost	t(s)
100	402	58,163.27	300	*402.0*	± 0.00	*58,089.89*	310
200	601	186,158.61	900	601.3	± 1.12	192,649.45	929
400	1142	447,627.43	900	1152.4	± 8.59	470,814.57	959
600	1643	935,948.36	1800	1653.5	±11.14	975,073.23	1909
800	2146	1,551,495.36	3600	2146.2	±12.73	1,617,631.40	3783
1000	2634	2,310,830.27	3600	*2629.6*	±14.98	*2,374,610.91*	3883

[a]: results reported in [7] using an Intel Xeon E5-1620 @ 3.5 GHz

each size to summarize the final results. We present in the same way. Column *Inst. Size* contains the six sizes from the instances of [11]. For each size and algorithm, column $\#V$ presents the accumulated number of vehicles, σ_v the standard deviation of the number of vehicles between runs, and *Cost* the cost of the solutions. Computational times used are given in column *t(s)*, in seconds.

Results show that, on average, the solutions of CLSQL and our replication are similar. In fact, for sizes 100 and 1000 (in italic), our average results are better than the original work. Remaining sizes are worse, but can still be considered equivalent due to the standard deviation on the number of vehicles. For example, size 100 has zero σ_v, meaning the method always reached the best number of vehicles, however for size 600 the deviation is about 11 vehicles, which accounts for the 10 vehicles of difference reached by our replication compared to CLSQL. Moreover, the best results reported in Table 4 show that R-CLSQL reached solutions equivalent to the ones reported by the original work at least once.

We intended to verify if R-CLSQL achieved similar results to the original work. Given the reported results and deviations, we consider the replication successful and continue to use R-CLSQL in comparisons with IGLS.

5.2 Component Selection and Parameter Tuning of IGLS

In order to better decide the components to be used by the IGLS, as well as its parameter values, a tuning procedure has been performed using the tool `irace` [12]. It relies on statistical tests to choose values for each parameter, obtaining a configuration with good results, on average.

One of the main questions to arise when proposing a matheuristic method such as the IGLS is whether the mathematical programming component is statistically significant. Even though it is possible to consider simple tests comparing a version with, and another without the SPP module, it could be the case that such component is only actually useful when combined with other values of the parameters. Thus, the tuning procedure is used to both verify if the SPP is significant, and with which configuration it seems to work well.

Table 2. Tuned parameters and their respective values

	Parameter	Values	
Notation	Description	Range	Best
`spp`	use SPP model	{`true`,`false`}	`true`
`ls`	use Local Search	{`true`,`false`}	`false`
`gen-sol`	how to generate initial solution	{`LS`, `greedy`}	`greedy`
α	initial ILS perturbation multiplier	[0.0,5.0]	2.47
A	maximum ILS perturbation multiplier	[0.0,5.0]	4.13
p_{shift}	probability of performing shift in perturbation	[0.0,1.0]	0.58
M_g	maximum number of AGES iterations	$\{10^4, 10^5, 10^6\}$	10^6
Z_g	perturbation size for AGES	{10,100,1000}	100
K	minimum number of requests to remove in LNS	[1,4]	3
M_l	number of iterations with no improvement in LNS	[600,1500]	928
L	LAHC list size	[500,2500]	1309
p_{shaw}	probability of performing shaw removal in LNS	[0.0,1.0]	0.71

Table 2 presents the parameters tuned with `irace`. The first three rows are components to be used by the algorithm, rather than actual values. These components include the SPP, the use of local search method (implemented as proposed in [7]) after AGES, and the initial solution generator (where LS refers to the already mentioned local search). Column *Notation* is the expression to denote the parameter, and column *Description* briefly explains where each parameter is used. Columns *Range* and *Best* are the range of values available for each parameter, and the best value found by `irace`, respectively.

A total of 2000 experiments were performed by `irace`, using a randomly selected subset of instances. It is worth mentioning that the five elite configurations returned by `irace`, which have no statistical difference, had the parameter `spp` set to `true`. Then, it is reasonable to say that the SPP has a positive impact on solution quality, and it surpasses any computational cost of being used. The best values reported are used in all experiments of IGLS.

5.3 Numerical Results of IGLS

For the PDPTW, comparing solely with the best-known solutions (BKS) published in [22] can be misleading. These results are from various methods, many from proprietary algorithms, or with no peer-reviewed publication, to which there is neither information on how they were obtained, nor on how much time it was needed. Because of that, the proposed algorithm is compared to the method of CLSQL, which can be considered the current *state-of-the-art*.

Table 3 compares results of our replication R-CLSQL and the IGLS regarding average solution quality. Columns are the same as in Table 1. Improved solutions are highlighted in bold.

On average, results of R-CLSQL and IGLS are the same for instances of size 100, with slightly smaller solution cost for IGLS. For instances of sizes 200, 400, 600, and 800, the average solution was better for IGLS, with differences as

Table 3. Comparison between average results of R-CLSQL and IGLS

Inst. Size	R-CLSQL				IGLS			
	$\#V$	σ_v	Cost	t(s)	$\#V$	σ_v	Cost	t(s)
100	402.0	± 0.00	58,089.89	310	**402.0**	± 0.00	**58,080.54**	302
200	601.3	± 1.12	192,649.45	929	**600.6**	± 0.52	**189,189.25**	905
400	1152.4	± 8.59	470,814.57	959	**1143.8**	± 4.43	**464,123.57**	919
600	1653.5	±11.14	975,073.23	1909	**1647.2**	± 7.31	**967,684.34**	1861
800	2146.2	±12.73	1,588,410.73	3783	**2143.5**	± 9.58	**1,597,189.41**	3692
1000	**2629.6**	±14.98	**2,374,610.91**	3883	2631.6	±16.18	2,374,504.33	3784

little as 0.7 (200), up to 8.6 vehicles (400). However, for the set of 1000 requests, R-CLSQL had a better performance, with 2.0 fewer vehicles. It indicates that on average our approach works better for instances of small and medium sizes. Although, according to the deviation in the number of vehicles, IGLS can be considered competitive even for the largest set of instances.

Note that if comparisons had been made using the original information for a single execution of CLSQL, the conclusions would be different. Our proposed method would have had better results for sizes 100, 200, 800 and 1000, which is only partially true if we consider the average behavior of the algorithms.

Nevertheless, we compare the best results found by each algorithm among their multiple executions. Also, for informative purposes, we compare these results to the BKS values available at SINTEF[2] [22]. Table 4 compares such results. Values are no longer the accumulated average, but the accumulated value of the best solutions found.

Table 4. Comparison between best solutions of R-CLSQL and IGLS in 10 or 5 runs

Inst. Size	BKS		R-CLSQL			IGLS		
	$\#V$	Cost	$\#V$	Cost	t(s)	$\#V$	Cost	t(s)
100	402	58,059.55	402	58,059.55	310	402	58,059.55	302
200	600	183,848.47	600	186,300.36	927	600	184,557.16	905
400	1133	438,718.64	1142	449,091.70	960	1138	447,767.02	913
600	1628	897,494.59	1642	939,861.14	1917	1639	930,144.54	1856
800	2116	1,519,213.78	2134	1,576,713.11	3772	2133	1,563,062.74	3704
1000	2572	2,231,988.31	2615	2,327,553.52	3885	2617	2,315,764.64	3803

Both algorithms, R-CLSQL and IGLS, were able to reach the best-known solutions for all instances of size 100 at least once. Although, for all the other instance sizes, some solutions could not be achieved by any of the methods. Moreover, the comparison between the best solutions found by the two algorithms agrees with the comparison of the average values. IGLS has better solutions for sizes 200, 400, 600 and 800, while R-CLSQL dominates 1000. The only

[2] According to the solutions published until April 16, 2018.

Table 5. New best-known solutions

Instance		BKS		CLSQL		IGLS			
Name	*Size*	*#V*	Cost	*#V*	Cost	*#V*	Cost	gap(%)	t(s)
LC1_2_8	200	19	3367.48	19	3397.65	19	3354.27	−0.39	900
LR1_2_4	200	10	3030.03	10	3044.69	10	3027.06	−0.10	902
LR1_2_7	200	12	3543.69	12	3550.61	12	3543.36	−0.01	900
LR1_2_8	200	9	2759.44	9	2814.32	9	2759.32	−0.01	901
LR2_2_8	200	2	2455.87	2	2586.42	2	2450.47	−0.22	902
LR2_2_9	200	3	3924.82	3	3924.82	3	3922.11	−0.07	901
LR2_2_10	200	3	3274.96	3	3274.96	3	3254.83	−0.61	904
LRC2_2_7	200	4	3018.05	4	3057.23	4	3016.53	−0.05	903
LC2_4_3	400	12	4412.75	12	4418.88	12	4407.71	−0.11	911
LRC1_4_3	400	24	7828.75	24	7856.72	24	7819.90	−0.11	900
LRC1_4_4	400	19	5806.20	19	5841.95	19	5804.47	−0.03	902
LRC2_4_2	400	10	7308.24	10	7605.61	10	7214.99	−1.28	902
LRC2_4_3	400	8	6505.71	8	6576.48	8	6483.48	−0.34	902
LRC2_4_5	400	10	7416.87	10	7462.66	10	7404.23	−0.17	901
LR2_6_2	600	9	23255.40	9	23255.40	9	22310.56	−4.06	1808
LR2_6_3	600	7	19183.41	7	19183.41	7	18337.46	−4.41	1806
LRC1_6_1	600	52	18293.94	52	18312.60	52	18293.72	−0.01	1805
LRC1_6_2	600	43	16624.01	43	17063.21	43	16576.53	−0.26	1825
LRC1_6_3	600	36	14041.72	36	14115.00	36	13987.02	−0.39	1802
LRC1_6_8	600	33	15862.32	33	15919.78	33	15812.61	−0.31	2000
LRC2_6_1	600	16	14782.39	16	14892.18	16	14665.50	−0.79	1810
LRC2_6_6	600	12	15315.05	12	17149.19	12	15200.75	−0.75	1800
LRC1_8_7	800	50	29948.00	51	28705.17	50	29378.53	−1.90	3609

exception is size 100, where they reach the same solution. On the other hand, comparing the best solutions of IGLS to the single execution of CLSQL leads to the conclusion that our method outperforms the latter in all instance sizes.

A total of 23 new BKS solutions have been found by the IGLS method, using a diversified set of parameters. These solutions have been validated and published by SINTEF [22]. Table 5 presents for each new solution, the name of the instance, and its size. Also, for each result presented, columns $\#V$ and *Cost* are the number of vehicles and the cost of the solution, respectively. We present the previous BKS published at SINTEF, the best solution reported by CLSQL [7], and the new result found by IGLS. Because all new solutions improve the cost, but not the number of vehicles, we present the perceptual improvement of the cost in column *gap(%)*, computed as $100 \cdot (S - BKS)/BKS$, where S is our new solution cost, and BKS the previous one.

The improvements achieved with the new BKS range from as little as 0.1%, up to more than 4.0% in the total cost of a solution. It further confirms that the use of a SPP model can help a method reach good solutions, even for problems with many side-constraints as the PDPTW.

6 Conclusion and Future Work

This work proposed a matheuristic approach based on Set Partitioning to solve the Pickup and Delivery Problem with Time Windows. The method combines Adaptive Guided Ejection Search, Large Neighborhood search and exact solutions of the SPP, in an Iterated Local Search framework. Both AGES and LNS had already been shown to work very well for the PDPTW, but their combination with a SPP solver had not been previously tested.

The study showed the addition of SPP can boost the results, and it seems to work very well on small and medium-sized instances, while it remains competitive for the largest ones. Comparisons with a *state-of-the-art* algorithm further indicate this, and a number of new best-known solutions has been found for a well-known benchmark set.

However, certain research paths need more investigation. It has been noted that in many instances the AGES seemed to use too much unnecessary computational time at every iteration when it could no longer reduce the number of routes. A more refined adaptive system could allow for this time to be spent by the LNS phase instead, trying to reduce the total cost. As for the LNS, many iterations where a single request could not be reinserted were wasted. Allowing some degree of infeasibility in LNS could help avoid these cases and to explore certain regions of the search space that would otherwise remain untouched. Also, the current use of the SPP is quite simple. Although it has performed well, it could be interesting to add routes of every local minimum of the LNS, but it would require an efficient pool size management since much more routes would be added per ILS iteration. At last, the component selection used during the tuning is far from exhaustive, and using an automatic algorithm design [12] for this problem could lead to even better results.

Acknowledgment. This work was partially supported by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) and FAPERGS (Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul). In addition, the authors acknowledge the valuable contributions of the two anonymous reviewers.

References

1. Alvarenga, G.B., Mateus, G.R., De Tomi, G.: A genetic and set partitioning two-phase approach for the vehicle routing problem with time windows. Comput. Oper. Res. **34**(6), 1561–1584 (2007)
2. Archetti, C., Speranza, M.G.: A survey on matheuristics for routing problems. EURO J. Comput. Optim. **2**(4), 223–246 (2014)
3. Baldacci, R., Bartolini, E., Mingozzi, A.: An exact algorithm for the pickup and delivery problem with time windows. Oper. Res. **59**(2), 414–426 (2011)
4. Bent, R., Van Hentenryck, P.: A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. Comput. Oper. Res. **33**(4), 875–893 (2006)

5. Boschetti, M.A., Maniezzo, V., Roffilli, M., Bolufé Röhler, A.: Matheuristics: optimization, simulation and control. In: Blesa, M.J., Blum, C., Di Gaspero, L., Roli, A., Sampels, M., Schaerf, A. (eds.) HM 2009. LNCS, vol. 5818, pp. 171–177. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04918-7_13
6. Burke, E.K., Bykov, Y.: A late acceptance strategy in hill-climbing for exam timetabling problems. In: PATAT 2008 Conference, Montreal, Canada (2008)
7. Curtois, T., Landa-Silva, D., Qu, Y., Laesanklang, W.: Large neighbourhood search with adaptive guided ejection search for the pickup and delivery problem with time windows. EURO J. Transp. Logist., pp. 1–42 (2017)
8. Dumas, Y., Desrosiers, J., Soumis, F.: The pickup and delivery problem with time windows. Eur. J. Oper. Res. **54**(1), 7–22 (1991)
9. Grangier, P., Gendreau, M., Lehuédé, F., Rousseau, L.M.: A matheuristic based on large neighborhood search for the vehicle routing problem with cross-docking. Comput. Oper. Res. **84**, 116–126 (2017)
10. Laporte, G.: What you should know about the vehicle routing problem. Nav. Res. Logist. (NRL) **54**(8), 811–819 (2007)
11. Li, H., Lim, A.: A metaheuristic for the pickup and delivery problem with time windows. Int. J. Artif. Intell. Tools **12**(02), 173–186 (2003)
12. López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Oper. Res. Perspect. **3**, 43–58 (2016)
13. Lourenço, H.R., Martin, O.C., Stützle, T.: Iterated local search: framework and applications. In: Gendreau, M., Potvin, J.Y. (eds.) Handbook of Metaheuristics. International Series in Operations Research & Management Science, vol. 146, pp. 363–397. Springer, Boston (2010). https://doi.org/10.1007/978-1-4419-1665-5_12
14. Nalepa, J., Blocho, M.: Enhanced guided ejection search for the pickup and delivery problem with time windows. In: Nguyen, N.T., Trawiński, B., Fujita, H., Hong, T.-P. (eds.) ACIIDS 2016. LNCS (LNAI), vol. 9621, pp. 388–398. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49381-6_37
15. Nanry, W.P., Barnes, J.W.: Solving the pickup and delivery problem with time windows using reactive tabu search. Transp. Res. Part B Methodol. **34**(2), 107–121 (2000)
16. Parragh, S.N., Doerner, K.F., Hartl, R.F.: A survey on pickup and delivery problems. J. für Betriebswirtschaft **58**(1), 21–51 (2008)
17. Parragh, S.N., Schmid, V.: Hybrid column generation and large neighborhood search for the dial-a-ride problem. Comput. Oper. Res. **40**(1), 490–497 (2013)
18. Ropke, S., Cordeau, J.F.: Branch-and-cut-and-price for the pickup and delivery problem with time windows. Transp. Sci. **43**(3), 267–286 (2009)
19. Ropke, S., Pisinger, D.: An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transp. Sci. **40**(4), 455–472 (2006)
20. Savelsbergh, M.W., Sol, M.: The general pickup and delivery problem. Transp. Sci. **29**(1), 17–29 (1995)
21. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49481-2_30
22. SINTEF: Li & lim benchmark instances (2008). https://www.sintef.no/projectweb/top/pdptw/li-lim-benchmark/. Accessed 18 April 2018
23. Solomon, M.M.: Algorithms for the vehicle routing and scheduling problems with time window constraints. Oper. Res. **35**(2), 254–265 (1987)
24. Subramanian, A., Uchoa, E., Ochi, L.S.: A hybrid algorithm for a class of vehicle routing problems. Comput. Oper. Res. **40**(10), 2519–2531 (2013)